Rethinking the Way We Do Templates in C++

Mateusz Pusz

September 17, 2019

Plan for the talk

1. User experience
2. New toys in a toolbox
3. Performance

USER EXPERIENCE

Physical Units library in a nutshell

```
// simple numeric operations
static_assert(10km / 2 == 5km);
```

Physical Units library in a nutshell

```
// simple numeric operations
static_assert(10km / 2 == 5km);
```

```
// unit conversions
static_assert(1h == 3600s);
static_assert(1km + 1m == 1001m);
```

Physical Units library in a nutshell

```
// simple numeric operations
static_assert(10km / 2 == 5km);
```

```
// unit conversions
static_assert(1h == 3600s);
static_assert(1km + 1m == 1001m);
```

```
// dimension conversions
static_assert(1km / 1s == 1000mps);
static_assert(2kmph * 2h == 4km);
static_assert(2km / 2kmph == 1h);

static_assert(1000 / 1s == 1kHz);

static_assert(10km / 5km == 2);
```

Toy example

```
/* velocity */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

Toy example

```
/* velocity */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

```
const auto kmph = avg_speed(/* 220 km */, /* 2 hours */);
std::cout << /* kmph */ << " km/h\n";
```

Toy example

```
/* velocity */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

```
const auto kmph = avg_speed(/* 220 km */, /* 2 hours */);
std::cout << /* kmph */ << " km/h\n";
```

```
const auto mph = avg_speed(/* 140 miles */, /* 2 hours */);
std::cout << /* mph */ << " mph\n";
```

Toy example

```
/* velocity */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

```
const auto kmph = avg_speed(/* 220 km */, /* 2 hours */);
std::cout << /* kmph */ << " km/h\n";
```

```
const auto mph = avg_speed(/* 140 miles */, /* 2 hours */);
std::cout << /* mph */ << " mph\n";
```

- **Compile time safety** to make sure that the result is of a correct dimension

Toy example

```
/* velocity */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

```
const auto kmph = avg_speed(/* 220 km */, /* 2 hours */);
std::cout << /* kmph */ << " km/h\n";
```

```
const auto mph = avg_speed(/* 140 miles */, /* 2 hours */);
std::cout << /* mph */ << " mph\n";
```

- **Compile time safety** to make sure that the result is of a correct dimension
- Support for **multiple units and unit prefixes**

Toy example

```
/* velocity */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

```
const auto kmph = avg_speed(/* 220 km */, /* 2 hours */);
std::cout << /* kmph */ << " km/h\n";
```

```
const auto mph = avg_speed(/* 140 miles */, /* 2 hours */);
std::cout << /* mph */ << " mph\n";
```

- Compile time safety to make sure that the result is of a correct dimension
- Support for multiple units and unit prefixes
- No runtime overhead
  - no additional intermediate conversions
  - as fast as a custom code implemented with **`double`**s

User experience: Compilation: Boost.Units

```
constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d, bu::quantity<bu::si::time> t)
{ return d * t; }
```

User experience: Compilation: Boost.Units

```
constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d, bu::quantity<bu::si::time> t)
{ return d * t; }
```

GCC-8

```
error: could not convert ‘boost::units::operator*(const boost::units::quantity<Unit1, X>&,
const boost::units::quantity<Unit2, Y>&) [with Unit1 = boost::units::unit<boost::units::list<boost::units::dim
<boost::units::length_base_dimension, boost::units::static_rational<1> >, boost::units::dimensionless_type>,
boost::units::homogeneous_system<boost::units::list<boost::units::si::meter_base_unit,
boost::units::list<boost::units::scaled_base_unit<boost::units::cgs::gram_base_unit, boost::units::scale<10,
boost::units::static_rational<3> > >, boost::units::list<boost::units::si::second_base_unit,
boost::units::list<boost::units::si::ampere_base_unit, boost::units::list<boost::units::si::kelvin_base_unit,
boost::units::list<boost::units::si::mole_base_unit, boost::units::list<boost::units::si::candela_base_unit,
boost::units::list<boost::units::angle::radian_base_unit, boost::units::list<boost::units::angle::steradian_base_unit,
boost::units::dimensionless_type> > > > > > > > > >; Unit2 = boost::units::unit<boost::units::list<boost::units::dim
<boost::units::time_base_dimension, boost::units::static_rational<1> >, boost::units::dimensionless_type>,
boost::units::homogeneous_system<boost::units::list<boost::units::si::meter_base_unit,
boost::units::list<boost::units::scaled_base_unit<boost::units::cgs::gram_base_unit, boost::units::scale<10,
boost::units::static_rational<3> > >, boost::units::list<boost::units::si::second_base_unit, boost::units::list
<boost::units::si::ampere_base_unit, boost::units::list<boost::units::si::kelvin_base_unit, boost::units::list
<boost::units::si::mole_base_unit, boost::units::list<boost::units::si::candela_base_unit, boost::units::list
<boost::units::angle::radian_base_unit, boost::units::list<boost::units::angle::steradian_base_unit,
boost::units::dimensionless_type> > > > > > > > > >; X = double; Y = double; typename
boost::units::multiply_typeof_helper<boost::units::quantity<Unit1, X>, boost::units::quantity<Unit2, Y> >::type =
...
```

User experience: Compilation: Boost.Units

```
constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d, bu::quantity<bu::si::time> t)
{ return d * t; }
```

GCC-8 (CONTINUED)

```
...
boost::units::quantity<boost::units::unit<boost::units::list<boost::units::dim<boost::units::length_base_dimension,
boost::units::static_rational<1> >, boost::units::list<boost::units::dim<boost::units::time_base_dimension,
boost::units::static_rational<1> >, boost::units::dimensionless_type> >, boost::units::homogeneous_system
<boost::units::list<boost::units::si::meter_base_unit, boost::units::list<boost::units::scaled_base_unit
<boost::units::cgs::gram_base_unit, boost::units::scale<10, boost::units::static_rational<3> > >,
boost::units::list<boost::units::si::second_base_unit, boost::units::list<boost::units::si::ampere_base_unit,
boost::units::list<boost::units::si::kelvin_base_unit, boost::units::list<boost::units::si::mole_base_unit,
boost::units::list<boost::units::si::candela_base_unit, boost::units::list<boost::units::angle::radian_base_unit,
boost::units::list<boost::units::angle::steradian_base_unit, boost::units::dimensionless_type> > > > > > > > >,
void>, double>](t)' from 'quantity<unit<list<[...],list<dim<[...],static_rational<1>>,[...]>>,[...],[...]>,[...]>'
to 'quantity<unit<list<[...],list<dim<[...],static_rational<-1>>,[...]>>,[...],[...]>,[...]>'
     return d * t;
            ~~^~~
```

User experience: Compilation: Boost.Units

```
constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d, bu::quantity<bu::si::time> t)
{ return d * t; }
```

CLANG-7

```
error: no viable conversion from returned value of type 'quantity<unit<list<[...], list<dim<[...],
static_rational<1, [...]>>, [...]>>, [...]>, [...]>' to function return type 'quantity<unit<list<[...], list<dim<[...],
static_rational<-1, [...]>>, [...]>>, [...]>, [...]>'
    return d * t;
           ^~~~
```

User experience: Compilation: Boost.Units

```
constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d, bu::quantity<bu::si::time> t)
{ return d * t; }
```

CLANG-7

```
error: no viable conversion from returned value of type 'quantity<unit<list<[...], list<dim<[...],
static_rational<1, [...]>>, [...]>>, [...]>, [...]>' to function return type 'quantity<unit<list<[...], list<dim<[...],
static_rational<-1, [...]>>, [...]>>, [...]>, [...]>'
    return d * t;
           ^~~~
```

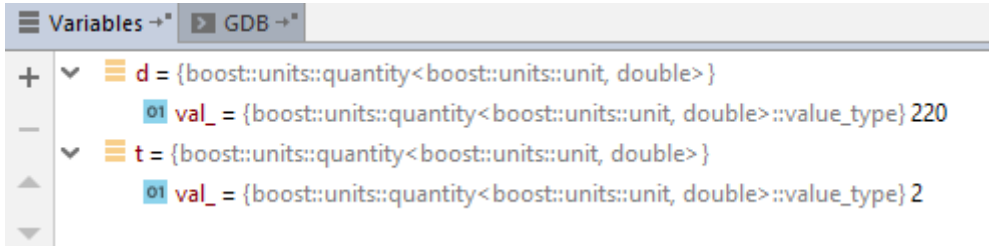
Sometimes a shorter error message is not necessarily better ;-)

User experience: Debugging: Boost.Units

```
35  template<typename LengthSystem, typename TimeSystem>
36  constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>> d,
37                           bu::quantity<bu::unit<bu::time_dimension, TimeSystem>> t)
38  {
39      return d / t;
40  }
41
```

User experience: Debugging: Boost.Units

```
35  template<typename LengthSystem, typename TimeSystem>
36  constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>> d,
37                           bu::quantity<bu::unit<bu::time_dimension, TimeSystem>> t)
38  {
39      return d / t;
40  }
41
```



User experience: Debugging: Boost.Units

```
35  template<typename LengthSystem, typename TimeSystem>
36  constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>> d,
37                           bu::quantity<bu::unit<bu::time_dimension, TimeSystem>> t)
38  {
39      return d / t;
40  }
41
```

```
Breakpoint 1, avg_speed<boost::units::heterogeneous_system<boost::units::heterogeneous_system_impl
<boost::units::list<boost::units::heterogeneous_system_dim<boost::units::si::meter_base_unit, boost::units::static_rational<1> >,
boost::units::dimensionless_type>, boost::units::list<boost::units::dim<boost::units::length_base_dimension,
boost::units::static_rational<1> >, boost::units::dimensionless_type>, boost::units::list<boost::units::scale_list_dim
<boost::units::scale<10, boost::units::static_rational<3> > >, boost::units::dimensionless_type> > >,
boost::units::heterogeneous_system<boost::units::heterogeneous_system_impl<boost::units::list
<boost::units::heterogeneous_system_dim<boost::units::scaled_base_unit<boost::units::si::second_base_unit,
boost::units::scale<60, boost::units::static_rational<2> > >, boost::units::static_rational<1> >,
boost::units::dimensionless_type>, boost::units::list<boost::units::dim<boost::units::time_base_dimension,
boost::units::static_rational<1> >, boost::units::dimensionless_type>, boost::units::dimensionless_type> > > (d=..., t=...) at
velocity_2.cpp:39
39          return d / t;
```

User experience: Debugging: Boost.Units

```
35  template<typename LengthSystem, typename TimeSystem>
36  constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>> d,
37                           bu::quantity<bu::unit<bu::time_dimension, TimeSystem>> t)
38  {
39      return d / t;
40  }
41
```

```
(gdb) ptype d
type = class boost::units::quantity<boost::units::unit<boost::units::list<boost::units::dim<boost::units::length_base_dimension,
boost::units::static_rational<1, 1> >, boost::units::dimensionless_type>, boost::units::heterogeneous_system
<boost::units::heterogeneous_system_impl<boost::units::list<boost::units::heterogeneous_system_dim
<boost::units::si::meter_base_unit, boost::units::static_rational<1, 1> >, boost::units::dimensionless_type>,
boost::units::list<boost::units::dim<boost::units::length_base_dimension, boost::units::static_rational<1, 1> >,
boost::units::dimensionless_type>, boost::units::list<boost::units::scale_list_dim<boost::units::scale<10, static_rational<3>>>,
boost::units::dimensionless_type> > >, void>, double> [with Unit = boost::units::unit<boost::units::list<boost::units::dim
<boost::units::length_base_dimension, boost::units::static_rational<1, 1> >, boost::units::dimensionless_type>,
boost::units::heterogeneous_system<boost::units::heterogeneous_system_impl<boost::units::list
<boost::units::heterogeneous_system_dim<boost::units::si::meter_base_unit, boost::units::static_rational<1, 1> >,
boost::units::dimensionless_type>, boost::units::list<boost::units::dim<boost::units::length_base_dimension,
boost::units::static_rational<1, 1> >, boost::units::dimensionless_type>, boost::units::list<boost::units::scale_list_dim
<boost::units::scale<10, static_rational<3> > >, boost::units::dimensionless_type> > >, void>, Y = double] {
...
```

Boost.Units: Design

```
template<typename Dim, typename System, typename Enable = void>
class unit;

template<typename Unit, typename Rep = double>
class quantity;
```

Boost.Units: Design

```
template<typename Dim, typename System, typename Enable = void>
class unit;

template<typename Unit, typename Rep = double>
class quantity;
```

```
struct length_base_dimension : base_dimension<length_base_dimension, -9> {};
struct time_base_dimension : base_dimension<time_base_dimension, -7> {};

using velocity_dimension = derived_dimension<length_base_dimension, 1, time_base_dimension, -1>::type;
```

Boost.Units: Design

```
template<typename Dim, typename System, typename Enable = void>
class unit;

template<typename Unit, typename Rep = double>
class quantity;
```

```
struct length_base_dimension : base_dimension<length_base_dimension, -9> {};
struct time_base_dimension : base_dimension<time_base_dimension, -7> {};

using velocity_dimension = derived_dimension<length_base_dimension, 1, time_base_dimension, -1>::type;
```

```
namespace si {
  using system = make_system<meter_base_unit, kilogram_base_unit, second_base_unit, ampere_base_unit,
                             kelvin_base_unit, mole_base_unit, candela_base_unit,
                             angle::radian_base_unit, angle::steradian_base_unit>::type;
}
```

Boost.Units: Design

```
template<typename Dim, typename System, typename Enable = void>
class unit;

template<typename Unit, typename Rep = double>
class quantity;
```

```
struct length_base_dimension : base_dimension<length_base_dimension, -9> {};
struct time_base_dimension : base_dimension<time_base_dimension, -7> {};

using velocity_dimension = derived_dimension<length_base_dimension, 1, time_base_dimension, -1>::type;
```

```
namespace si {
  using system = make_system<meter_base_unit, kilogram_base_unit, second_base_unit, ampere_base_unit,
                             kelvin_base_unit, mole_base_unit, candela_base_unit,
                             angle::radian_base_unit, angle::steradian_base_unit>::type;
}
```

```
using velocity = unit<velocity_dimension, si::system>;
```

User experience: Compilation: NHolthaus Units

```
constexpr units::velocity::meters_per_second_t avg_speed(units::length::meter_t d, units::time::second_t t)
{ return d * t; }
```

User experience: Compilation: NHolthaus Units

```
constexpr units::velocity::meters_per_second_t avg_speed(units::length::meter_t d, units::time::second_t t)
{ return d * t; }
```

GCC-8

```
error: static assertion failed: Units are not compatible.
   static_assert(traits::is_convertible_unit<UnitFrom, UnitTo>::value, "Units are not compatible.");
                 ^~~~~~
```

User experience: Compilation: NHolthaus Units

```
constexpr units::velocity::meters_per_second_t avg_speed(units::length::meter_t d, units::time::second_t t)
{ return d * t; }
```

GCC-8

```
error: static assertion failed: Units are not compatible.
   static_assert(traits::is_convertible_unit<UnitFrom, UnitTo>::value, "Units are not compatible.");
                 ^~~~~~
```

`static_assert` is often not the best solution

- does not influence the overload resolution process
- for some compilers does not provide enough context

User experience: Compilation: NHolthaus Units

```
constexpr units::velocity::meters_per_second_t avg_speed(units::length::meter_t d, units::time::second_t t)
{ return d * t; }
```

CLANG-7

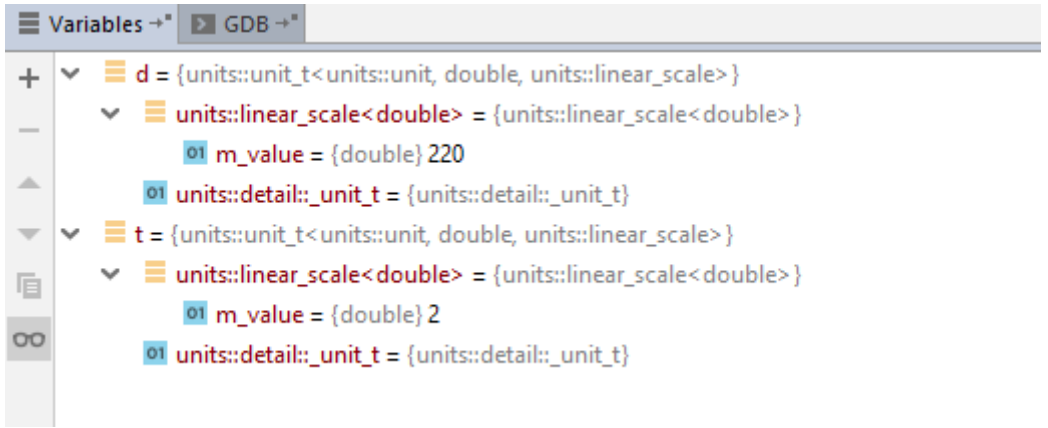
```
error: static_assert failed due to requirement 'traits::is_convertible_unit<unit<ratio<1, 1>, base_unit<ratio<1, 1>,
ratio<0, 1>, ratio<1, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1> >, ratio<0, 1>,
ratio<0, 1> >, unit<ratio<1, 1>, base_unit<ratio<1, 1>, ratio<0, 1>, ratio<-1, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1>,
ratio<0, 1>, ratio<0, 1>, ratio<0, 1> >, ratio<0, 1>, ratio<0, 1> > >::value' "Units are not compatible."
                static_assert(traits::is_convertible_unit<UnitFrom, UnitTo>::value, "Units are not compatible.");
                ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

User experience: Debugging: NHolthaus Units

```
23  template<typename Length, typename Time>
24  constexpr auto avg_speed(Length d, Time t)
25  {
26      static_assert(units::traits::is_length_unit<Length>::value);
27      static_assert(units::traits::is_time_unit<Time>::value);
28      const auto v = d / t;
29      static_assert(units::traits::is_velocity_unit<decltype(v)>::value);
30      return v;
31  }
```

User experience: Debugging: NHolthaus Units

```
23  template<typename Length, typename Time>
24  constexpr auto avg_speed(Length d, Time t)
25  {
26      static_assert(units::traits::is_length_unit<Length>::value);
27      static_assert(units::traits::is_time_unit<Time>::value);
28      const auto v = d / t;
29      static_assert(units::traits::is_velocity_unit<decltype(v)>::value);
30      return v;
31  }
```



User experience: Debugging: NHolthaus Units

```
23  template<typename Length, typename Time>
24  constexpr auto avg_speed(Length d, Time t)
25  {
26      static_assert(units::traits::is_length_unit<Length>::value);
27      static_assert(units::traits::is_time_unit<Time>::value);
28      const auto v = d / t;
29      static_assert(units::traits::is_velocity_unit<decltype(v)>::value);
30      return v;
31  }
```

```
Breakpoint 1, avg_speed<units::unit_t<units::unit<std::ratio<1000, 1>, units::unit<std::ratio<1>,
units::base_unit<std::ratio<1> > >, std::ratio<0, 1>, std::ratio<0, 1> > >, units::unit_t<units::unit<std::ratio<60>,
units::unit<std::ratio<60>, units::unit<std::ratio<1>, units::base_unit<std::ratio<0, 1>, std::ratio<0, 1>,
std::ratio<1> > > > > > >
(d=..., t=...) at velocity.cpp:28
28          const auto v = d / t;
```

User experience: Debugging: NHolthaus Units

```
23  template<typename Length, typename Time>
24  constexpr auto avg_speed(Length d, Time t)
25  {
26      static_assert(units::traits::is_length_unit<Length>::value);
27      static_assert(units::traits::is_time_unit<Time>::value);
28      const auto v = d / t;
29      static_assert(units::traits::is_velocity_unit<decltype(v)>::value);
30      return v;
31  }
```

```
(gdb) ptype d
type = class units::unit_t<units::unit<std::ratio<1000, 1>, units::unit<std::ratio<1, 1>, units::base_unit<std::ratio<1, 1>,
std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>,
std::ratio<0, 1> >, std::ratio<0, 1>, std::ratio<0, 1> >, std::ratio<0, 1>, std::ratio<0, 1> >, double, units::linear_scale>
[with Units = units::unit<std::ratio<1000, 1>, units::unit<std::ratio<1, 1>, units::base_unit<std::ratio<1, 1>, std::ratio<0, 1>,
std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1> >,
std::ratio<0, 1>, std::ratio<0, 1> >, std::ratio<0, 1>, std::ratio<0, 1> >, T = double] : public units::linear_scale<T>,
private units::detail::_unit_t {
...
```

NHolthaus Units: Design

```
template<class Meter = detail::meter_ratio<0>,
         class Kilogram = std::ratio<0>,
         class Second = std::ratio<0>,
         class Radian = std::ratio<0>,
         class Ampere = std::ratio<0>,
         class Kelvin = std::ratio<0>,
         class Mole = std::ratio<0>,
         class Candela = std::ratio<0>,
         class Byte = std::ratio<0>>
struct base_unit;

template<class Conversion, class BaseUnit, class PiExponent = std::ratio<0>, class Translation = std::ratio<0>>
struct unit;

#ifndef UNIT_LIB_DEFAULT_TYPE
#   define UNIT_LIB_DEFAULT_TYPE double
#endif

template<class Units, typename T = UNIT_LIB_DEFAULT_TYPE, template<typename> class NonLinearScale = linear_scale>
class unit_t;
```

NHolthaus Units: Design

```
template<class Meter = detail::meter_ratio<0>,
         class Kilogram = std::ratio<0>,
         class Second = std::ratio<0>,
         class Radian = std::ratio<0>,
         class Ampere = std::ratio<0>,
         class Kelvin = std::ratio<0>,
         class Mole = std::ratio<0>,
         class Candela = std::ratio<0>,
         class Byte = std::ratio<0>>
struct base_unit;

template<class Conversion, class BaseUnit, class PiExponent = std::ratio<0>, class Translation = std::ratio<0>>
struct unit;

#ifndef UNIT_LIB_DEFAULT_TYPE
#   define UNIT_LIB_DEFAULT_TYPE double
#endif

template<class Units, typename T = UNIT_LIB_DEFAULT_TYPE, template<typename> class NonLinearScale = linear_scale>
class unit_t;
```

```
using velocity_t = units::unit<std::ratio<1>,
                               units::base_unit<std::ratio<1>, std::ratio<0>, std::ratio<1>, std::ratio<0>,
                                                std::ratio<0>, std::ratio<0>, std::ratio<0>, std::ratio<0>,
                                                std::ratio<0>>,
                               std::ratio<0>, std::ratio<0>>;
```

A need to modernize our toolbox

- For most template metaprogramming libraries *compile-time errors are rare*

A need to modernize our toolbox

- For most template metaprogramming libraries *compile-time errors are rare*
- It is expected that engineers working with a physical units library **will experience compile-time errors very often**
  - generating compile-time errors for invalid calculation is the *main reason to create such a library*

A need to modernize our toolbox

- For most template metaprogramming libraries *compile-time errors are rare*
- It is expected that engineers working with a physical units library **will experience compile-time errors very often**
  - generating compile-time errors for invalid calculation is the *main reason to create such a library*

In case of the physical units library we have to rethink the way we do template metaprogramming!

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process
- As a result **end users** get **huge types in error messages**

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process
- As a result **end users** get **huge types in error messages**

EXAMPLE

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
using kilometre_per_hour = derived_unit<velocity, kilometre, hour>;
```

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process
- As a result **end users** get **huge types in error messages**

EXAMPLE

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
using kilometre_per_hour = derived_unit<velocity, kilometre, hour>;
```

```
void foo(quantity<kilometre_per_hour>(90));
```

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process
- As a result **end users** get **huge types in error messages**

EXAMPLE

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
using kilometre_per_hour = derived_unit<velocity, kilometre, hour>;
```

```
void foo(quantity<kilometre_per_hour>(90));
```

```
[with Q = units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1, 1>,
   units::exp<units::base_dim_time, 1, -1> >, units::ratio<5, 18> >, double>]
```

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process
- As a result **end users** get **huge types in error messages**

EXAMPLE

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
using kilometre_per_hour = derived_unit<velocity, kilometre, hour>;
```

```
void foo(quantity<kilometre_per_hour>(90));
```

```
[with Q = units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1, 1>,
   units::exp<units::base_dim_time, 1, -1> >, units::ratio<5, 18> >, double>]
```

It is a pity that we still do not have **strong typedefs** in the C++ language :-(

Inheritance as a workaround

```
struct velocity : make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>> {};
```

Inheritance as a workaround

```
struct velocity : make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>> {};
```

- **Similarly** to strong typedefs
  - *strong types* that do not vanish during compilation process
  - member functions of a base class *taking the base class type as an argument* will still *work with a child class type* provided instead (i.e. **op==**)

Inheritance as a workaround

```
struct velocity : make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>> {};
```

- **Similarly** to strong typedefs
  - *strong types* that do not vanish during compilation process
  - member functions of a base class *taking the base class type as an argument* will still *work with a child class type* provided instead (i.e. **op==**)
- **Alternatively** to strong typedefs
  - do not automatically inherit *constructors and assignment operators*
  - member functions of a base class *returning the base class type* will still *return the same base type for a child class instance*

Type substitution problem

```
Velocity auto v = 10m / 2s;
```

Type substitution problem

```
Velocity auto v = 10m / 2s;
```

```
template<typename U1, typename Rep1, typename U2, typename Rep2>
[[nodiscard]] constexpr Quantity operator/(const quantity<U1, Rep1>& lhs,
                                           const quantity<U2, Rep2>& rhs);
```

Type substitution problem

```
Velocity auto v = 10m / 2s;
```

```
template<typename U1, typename Rep1, typename U2, typename Rep2>
[[nodiscard]] constexpr Quantity operator/(const quantity<U1, Rep1>& lhs,
                                           const quantity<U2, Rep2>& rhs);
```

How to form a **`velocity`** dimension child class type from division of **`length`** by **`time`**?

Downcasting facility

BASE CLASS

```
template<typename BaseType>
struct downcast_base {
  using base_type = BaseType;
};
```

```
template<Exponent... Es>
struct dimension : downcast_base<dimension<Es...>> {};
```

Downcasting facility

BASE CLASS

```
template<typename BaseType>
struct downcast_base {
  using base_type = BaseType;
};
```

```
template<Exponent... Es>
struct dimension : downcast_base<dimension<Es...>> {};
```

DOWNCASTABLE

```
template<typename T>
concept Downcastable =
    requires {
      typename T::base_type;
    } &&
    std::derived_from<T, downcast_base<typename T::base_type>>;
```

Downcasting facility

HELPER ALIASES

```
template<Downcastable T>
using downcast_from = T::base_type;

template<Downcastable T>
using downcast_to = std::type_identity<T>;
```

Downcasting facility

HELPER ALIASES

```
template<Downcastable T>
using downcast_from = T::base_type;

template<Downcastable T>
using downcast_to = std::type_identity<T>;
```

DOWNCASTING TRAITS

```
template<Downcastable T>
struct downcasting_traits : downcast_to<T> {};

template<Downcastable T>
using downcasting_traits_t = downcasting_traits<T>::type;
```

Downcasting facility

EXAMPLE

```
template<Exponent... Es>
struct dimension : downcast_base<dimension<Es...>> {};
```

Downcasting facility

EXAMPLE

```
template<Exponent... Es>
struct dimension : downcast_base<dimension<Es...>> {};
```

```
struct velocity : make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>> {};
template<>
struct downcasting_traits<downcast_from<velocity>> : downcast_to<velocity> {};
```

User experience: Compilation

BEFORE

```
[with D = units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1, 1>,
                  units::exp<units::base_dim_time, 1, -1> >, units::ratio<5, 18> >, double>]
```

User experience: Compilation

BEFORE

```
[with D = units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1, 1>,
                  units::exp<units::base_dim_time, 1, -1> >, units::ratio<5, 18> >, double>]
```

AFTER

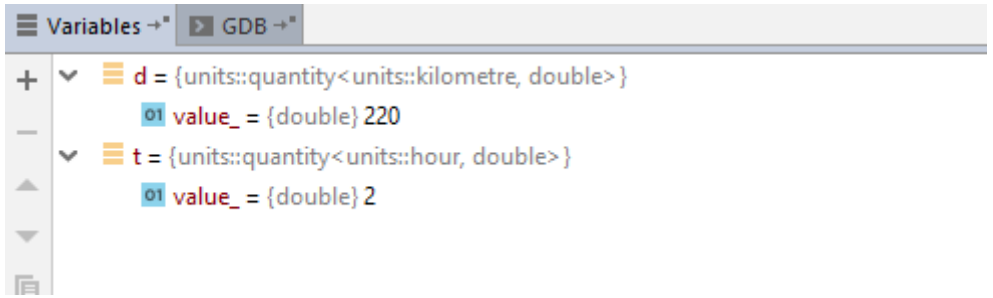
```
[with D = units::quantity<units::kilometre_per_hour, double>]
```

User experience: Debugging

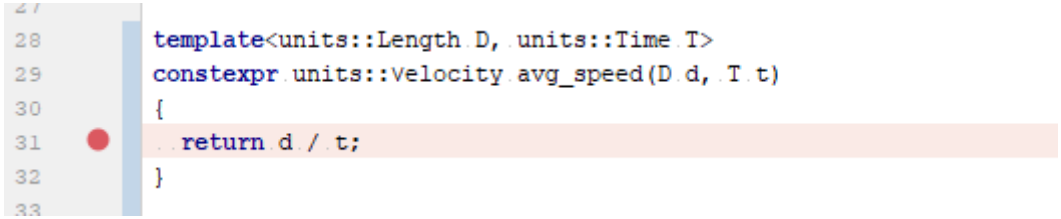
```
28  template<units::Length D, units::Time T>
29  constexpr units::Velocity avg_speed(D d, T t)
30  {
31      return d / t;
32  }
```

User experience: Debugging

```
28  template<units::Length D, units::Time T>
29  constexpr units::Velocity avg_speed(D d, T t)
30  {
31      return d / t;
32  }
```



User experience: Debugging



```
28  template<units::Length D, units::Time T>
29  constexpr units::Velocity avg_speed(D d, T t)
30  {
31      return d / t;
32  }
```

```
Breakpoint 1, avg_speed<units::quantity<units::kilometre, double>,
                        units::quantity<units::hour, double> >
(d=..., t=...) at velocity.cpp:31
31      return d / t;
```

User experience: Debugging

```
28  template<units::Length D, units::Time T>
29  constexpr units::Velocity avg_speed(D d, T t)
30  {
31      return d / t;
32  }
```

```
(gdb) ptype d
type = class units::quantity<units::kilometre, double>
[with U = units::kilometre, Rep = double] {
...
```

NEW TOYS IN A TOOLBOX

Traditional implementation of `std::ratio`

```
template<intmax_t Num, intmax_t Den = 1>
struct ratio {
  static constexpr intmax_t num = Num * static_sign<Den>::value / static_gcd<Num, Den>::value;
  static constexpr intmax_t den = static_abs<Den>::value / static_gcd<Num, Den>::value;
  using type = ratio<num, den>;
};
```

Traditional implementation of `std::ratio`

```
template<intmax_t Num, intmax_t Den = 1>
struct ratio {
  static constexpr intmax_t num = Num * static_sign<Den>::value / static_gcd<Num, Den>::value;
  static constexpr intmax_t den = static_abs<Den>::value / static_gcd<Num, Den>::value;
  using type = ratio<num, den>;
};
```

```
template<intmax_t Pn>
struct static_sign : integral_constant<intmax_t, (Pn < 0) ? -1 : 1> {};
```

Traditional implementation of `std::ratio`

```
template<intmax_t Num, intmax_t Den = 1>
struct ratio {
  static constexpr intmax_t num = Num * static_sign<Den>::value / static_gcd<Num, Den>::value;
  static constexpr intmax_t den = static_abs<Den>::value / static_gcd<Num, Den>::value;
  using type = ratio<num, den>;
};
```

```
template<intmax_t Pn>
struct static_sign : integral_constant<intmax_t, (Pn < 0) ? -1 : 1> {};
```

```
template<intmax_t Pn>
struct static_abs : integral_constant<intmax_t, Pn * static_sign<Pn>::value> {};
```

Traditional implementation of `std::ratio`

```
template<intmax_t Num, intmax_t Den = 1>
struct ratio {
  static constexpr intmax_t num = Num * static_sign<Den>::value / static_gcd<Num, Den>::value;
  static constexpr intmax_t den = static_abs<Den>::value / static_gcd<Num, Den>::value;
  using type = ratio<num, den>;
};
```

```
template<intmax_t Pn, intmax_t Qn>
struct static_gcd : static_gcd<Qn, (Pn % Qn)> {};

template<intmax_t Pn>
struct static_gcd<Pn, 0> : integral_constant<intmax_t, static_abs<Pn>::value> {};

template<intmax_t Qn>
struct static_gcd<0, Qn> : integral_constant<intmax_t, static_abs<Qn>::value> {};
```

Traditional implementation of `std::ratio_multiply`

```
namespace detail {

  template<typename R1, typename R2>
  struct ratio_multiply_impl {
  private:
    static constexpr intmax_t gcd1 = static_gcd<R1::num, R2::den>::value;
    static constexpr intmax_t gcd2 = static_gcd<R2::num, R1::den>::value;
  public:
    using type = ratio<safe_multiply<(R1::num / gcd1), (R2::num / gcd2)>::value,
                       safe_multiply<(R1::den / gcd2), (R2::den / gcd1)>::value>;
    static constexpr intmax_t num = type::num;
    static constexpr intmax_t den = type::den;
  };

}

template<typename R1, typename R2>
using ratio_multiply = detail::ratio_multiply_impl<R1, R2>::type;
```

constexpr-based implementation of ratio

```
template<typename T>
[[nodiscard]] constexpr T abs(T v) noexcept { return v < 0 ? -v : v; }
```

```
template<std::intmax_t Num, std::intmax_t Den = 1>
struct ratio {
  static constexpr std::intmax_t num = Num * (Den < 0 ? -1 : 1) / std::gcd(Num, Den);
  static constexpr std::intmax_t den = abs(Den) / std::gcd(Num, Den);

  using type = ratio<num, den>;
};
```

`constexpr`-based implementation of `ratio`

```
template<typename T>
[[nodiscard]] constexpr T abs(T v) noexcept { return v < 0 ? -v : v; }
```

```
template<std::intmax_t Num, std::intmax_t Den = 1>
struct ratio {
  static constexpr std::intmax_t num = Num * (Den < 0 ? -1 : 1) / std::gcd(Num, Den);
  static constexpr std::intmax_t den = abs(Den) / std::gcd(Num, Den);

  using type = ratio<num, den>;
};
```

- *Better code reuse* between run-time and compile-time programming
- *Less instantiations* of class templates

constexpr-based implementation of ratio_multiply

```
namespace detail {

  template<typename R1, typename R2>
  struct ratio_multiply_impl {
  private:
    static constexpr std::intmax_t gcd1 = std::gcd(R1::num, R2::den);
    static constexpr std::intmax_t gcd2 = std::gcd(R2::num, R1::den);
  public:
    using type = ratio<safe_multiply(R1::num / gcd1, R2::num / gcd2),
                       safe_multiply(R1::den / gcd2, R2::den / gcd1)>;
    static constexpr std::intmax_t num = type::num;
    static constexpr std::intmax_t den = type::den;
  };

}

template<Ratio R1, Ratio R2>
using ratio_multiply = detail::ratio_multiply_impl<R1, R2>::type;
```

P0732 P1185 Class Types in Non-Type Template Parameters

- Allow *non-union class types* to appear in non-type template parameters (**NTTP**)
- Require that types used as such, have **strong structural equality**
  - a type **T** is having strong structural equality if *each subobject recursively has defaulted* **==** *and none of the subobjects are floating point types*

P0732 P1185 Class Types in Non-Type Template Parameters

- Allow *non-union class types* to appear in non-type template parameters (**NTTP**)
- Require that types used as such, have **strong structural equality**
  - a type **T** is having strong structural equality if *each subobject recursively has defaulted* **==** *and none of the subobjects are floating point types*

C++17

```
template<size_t seconds>
class fixed_timer { /* ... */ };
```

C++20

```
template<std::chrono::seconds seconds>
class fixed_timer { /* ... */ };
```

NTTP support and controversies

- gcc is the only compiler supporting NTTP for now
  - standard compliant *support in gcc-9.2*
- *QoI and compile-time performance* might improve over time

NTTP support and controversies

- gcc is the only compiler supporting NTTP for now
  - standard compliant *support in gcc-9.2*
- *QoI and compile-time performance* might improve over time
- There are some *controversies regarding the NTTP* feature (P1837)
  - Identity and Equality are not the same thing

NTTP support and controversies

- gcc is the only compiler supporting NTTP for now
  - standard compliant *support in gcc-9.2*
- *QoI and compile-time performance* might improve over time
- There are some *controversies regarding the NTTP* feature (P1837)
  - Identity and Equality are not the same thing

NTTP support in the Physical Units library will not be merged to a master branch before the ISO C++ Committee meeting in Belfast

NTTP-based implementation of `ratio`

```
struct ratio {
  std::intmax_t num;
  std::intmax_t den;

  explicit constexpr ratio(std::intmax_t n, std::intmax_t d = 1) :
    num(n * (d < 0 ? -1 : 1) / std::gcd(n, d)),
    den(abs(d) / std::gcd(n, d))
  {
  }

  [[nodiscard]] constexpr bool operator==(const ratio&) = default;

  // ...
};
```

NTTP-based implementation of `ratio_multiply`

```
struct ratio {
  // ...
  [[nodiscard]] friend constexpr ratio operator*(const ratio& lhs, const ratio& rhs)
  {
    const std::intmax_t gcd1 = std::gcd(lhs.num, rhs.den);
    const std::intmax_t gcd2 = std::gcd(rhs.num, lhs.den);
    return ratio(safe_multiply(lhs.num / gcd1, rhs.num / gcd2),
                 safe_multiply(lhs.den / gcd2, rhs.den / gcd1));
  }









};
```

NTTP-based implementation of `ratio_multiply`

```
struct ratio {
  // ...
  [[nodiscard]] friend constexpr ratio operator*(const ratio& lhs, const ratio& rhs)
  {
    const std::intmax_t gcd1 = std::gcd(lhs.num, rhs.den);
    const std::intmax_t gcd2 = std::gcd(rhs.num, lhs.den);
    return ratio(safe_multiply(lhs.num / gcd1, rhs.num / gcd2),
                 safe_multiply(lhs.den / gcd2, rhs.den / gcd1));
  }

  [[nodiscard]] friend consteval ratio operator*(std::intmax_t n, const ratio& rhs)
  {
    return ratio(n) * rhs;
  }

  [[nodiscard]] friend consteval ratio operator*(const ratio& lhs, std::intmax_t n)
  {
    return lhs * ratio(n);
  }
};
```

NTTP in action

```
// US customary units
struct yard : unit<length, ratio(9'144, 10'000)> {};
template<> struct downcasting_traits<downcast_from<yard>> : downcast_to<yard> {};

struct foot : unit<length, yard::ratio / 3> {};
template<> struct downcasting_traits<downcast_from<foot>> : downcast_to<foot> {};

struct inch : unit<length, foot::ratio / 12> {};
template<> struct downcasting_traits<downcast_from<inch>> : downcast_to<inch> {};

struct mile : unit<length, 1'760 * yard::ratio> {};
template<> struct downcasting_traits<downcast_from<mile>> : downcast_to<mile> {};
```

NTTP in action

BEFORE

```
template<typename U1, typename Rep1, typename U2, typename Rep2>
  requires (std::ratio_divide<typename U1::ratio, typename U2::ratio>::den == 1)
[[nodiscard]] quantity<downcasting_traits_t<unit<dimension_divide_t<typename U1::dimension, typename U2::dimension>,
                                                std::ratio_divide<typename U1::ratio, typename U2::ratio>>>,
                      std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<U1, Rep1>& lhs,
                    const quantity<U2, Rep2>& rhs);
```

NTTP in action

BEFORE

```
template<typename U1, typename Rep1, typename U2, typename Rep2>
  requires (std::ratio_divide<typename U1::ratio, typename U2::ratio>::den == 1)
[[nodiscard]] quantity<downcasting_traits_t<unit<dimension_divide_t<typename U1::dimension, typename U2::dimension>,
                                                std::ratio_divide<typename U1::ratio, typename U2::ratio>>>,
                       std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<U1, Rep1>& lhs,
                    const quantity<U2, Rep2>& rhs);
```

AFTER

```
template<typename U1, typename Rep1, typename U2, typename Rep2>
  requires ((U1::ratio / U2::ratio).den == 1)
[[nodiscard]] quantity<downcasting_traits_t<unit<dimension_divide_t<typename U1::dimension, typename U2::dimension>,
                                                U1::ratio / U2::ratio>>,
                       std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<U1, Rep1>& lhs,
                    const quantity<U2, Rep2>& rhs);
```

NTTP in action

BEFORE

```
template<typename U1, typename Rep1, typename U2, typename Rep2>
  requires (std::ratio_divide<typename U1::ratio, typename U2::ratio>::den == 1)
[[nodiscard]] quantity<downcasting_traits_t<unit<dimension_divide_t<typename U1::dimension, typename U2::dimension>,
                                                std::ratio_divide<typename U1::ratio, typename U2::ratio>>>,
                      std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<U1, Rep1>& lhs,
                    const quantity<U2, Rep2>& rhs);
```

NTTP in action

BEFORE

```
template<typename U1, typename Rep1, typename U2, typename Rep2>
  requires (std::ratio_divide<typename U1::ratio, typename U2::ratio>::den == 1)
[[nodiscard]] quantity<downcasting_traits_t<unit<dimension_divide_t<typename U1::dimension, typename U2::dimension>,
                                                std::ratio_divide<typename U1::ratio, typename U2::ratio>>>,
                      std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<U1, Rep1>& lhs,
                    const quantity<U2, Rep2>& rhs);
```

AFTER

```
template<typename U1, typename Rep1, typename U2, typename Rep2>
  requires ((U1::ratio / U2::ratio).den == 1)
[[nodiscard]] quantity<downcasting_traits_t<unit<U1::dimension / U2::dimension, U1::ratio / U2::ratio>>,
                      std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<U1, Rep1>& lhs,
                    const quantity<U2, Rep2>& rhs);
```

NTTP in action

```
struct exp {
  base_dimension base;
  int num;
  int den;
  constexpr exp(base_dimension b, int n, int d = 1): base(b), num(n), den(d) {}
};
```

NTTP in action

```
struct exp {
  base_dimension base;
  int num;
  int den;
  constexpr exp(base_dimension b, int n, int d = 1): base(b), num(n), den(d) {}
};
```

```
struct dimension {
  std::vector<exp> exponents;  // compile-time vector
  constexpr dimension(std::initializer_list<exp> init);
  [[nodiscard]] friend constexpr dimension operator*(const dimension& lhs, const dimension& rhs);
  [[nodiscard]] friend constexpr dimension operator/(const dimension& lhs, const dimension& rhs);
};
```

NTTP in action

```
struct exp {
  base_dimension base;
  int num;
  int den;
  constexpr exp(base_dimension b, int n, int d = 1): base(b), num(n), den(d) {}
};
```

```
struct dimension {
  std::vector<exp> exponents;  // compile-time vector
  constexpr dimension(std::initializer_list<exp> init);
  [[nodiscard]] friend constexpr dimension operator*(const dimension& lhs, const dimension& rhs);
  [[nodiscard]] friend constexpr dimension operator/(const dimension& lhs, const dimension& rhs);
};
```

```
inline constexpr dimension velocity = { exp(base_dim_length, 1), exp(base_dim_time, -1) };
```

NTTP in action

```
struct exp {
  base_dimension base;
  int num;
  int den;
  constexpr exp(base_dimension b, int n, int d = 1): base(b), num(n), den(d) {}
};
```

```
struct dimension {
  std::vector<exp> exponents;  // compile-time vector
  constexpr dimension(std::initializer_list<exp> init);
  [[nodiscard]] friend constexpr dimension operator*(const dimension& lhs, const dimension& rhs);
  [[nodiscard]] friend constexpr dimension operator/(const dimension& lhs, const dimension& rhs);
};
```

```
inline constexpr dimension velocity = { exp(base_dim_length, 1), exp(base_dim_time, -1) };
```

exp and **dimension** values are only used as NTTP so those types and their implementation should exist only during compile-time

P1073 Immediate functions

```
struct exp {
  base_dimension base;
  int num;
  int den;
  consteval exp(base_dimension b, int n, int d = 1): base(b), num(n), den(d) {}
};
```

```
struct dimension {
  std::vector<exp> exponents;  // compile-time vector
  consteval dimension(std::initializer_list<exp> init);
  [[nodiscard]] friend consteval dimension operator*(const dimension& lhs, const dimension& rhs);
  [[nodiscard]] friend consteval dimension operator/(const dimension& lhs, const dimension& rhs);
};
```

```
inline consteval dimension velocity = { exp(base_dim_length, 1), exp(base_dim_time, -1) };
```

P1073 Immediate functions

```
struct exp {
  base_dimension base;
  int num;
  int den;
  consteval exp(base_dimension b, int n, int d = 1): base(b), num(n), den(d) {}
};
```

```
struct dimension {
  std::vector<exp> exponents;  // compile-time vector
  consteval dimension(std::initializer_list<exp> init);
  [[nodiscard]] friend consteval dimension operator*(const dimension& lhs, const dimension& rhs);
  [[nodiscard]] friend consteval dimension operator/(const dimension& lhs, const dimension& rhs);
};
```

```
inline consteval dimension velocity = { exp(base_dim_length, 1), exp(base_dim_time, -1) };
```

C++20 does not allow us yet to create **`consteval`** objects :-(

Class Types in Non-Type Template Parameters

Usage of class types as non-type template parameters (NTTP) might be *one of the most significant C++ improvements in template metaprogramming* during the last decade

How do you feel about such an interface?

```
void* foo(void* t);
```

How do you feel about such an interface?

```
void* foo(void* t);
```

```
template<typename T>
auto foo(T&& t);
```

How do you feel about such an interface?

```
void* foo(void* t);
```

```
template<typename T>
auto foo(T&& t);
```

```
auto foo = [](auto&& t){ /* ... */ };
```

How do you feel about such an interface?

```
void* foo(void* t);
```

```
template<typename T>
auto foo(T&& t);
```

```
auto foo = [](auto&& t){ /* ... */ };
```

Unconstrained template parameters are a **`void*`** of C++

Concepts

- Class/Function/Variable/Alias templates, and non-template functions (typically members of class templates) may be associated with a constraint

Concepts

- **Class**/**Function**/**Variable**/**Alias** templates, and **non-template functions** (typically members of class templates) may be **associated with a constraint**
- Constraint specifies **the requirements on template arguments**
  - can be used *to select the most appropriate function overloads and template specializations*

Concepts

- **Class**/**Function**/**Variable**/**Alias** templates, and **non-template functions** (typically members of class templates) may be **associated with a constraint**
- Constraint specifies **the requirements on template arguments**
  - can be used *to select the most appropriate function overloads and template specializations*
- *Named sets* of such requirements are called **concepts**

Concepts

- **Class**/**Function**/**Variable**/**Alias** templates, and **non-template functions** (typically members of class templates) may be **associated with a constraint**
- Constraint specifies **the requirements on template arguments**
  - can be used *to select the most appropriate function overloads and template specializations*
- *Named sets* of such requirements are called **concepts**
- **Concept**
  - is a *named predicate*
  - evaluated *at compile time*
  - a *part of the interface of a template*

Concept example

```
template<typename T>
concept Velocity = QuantityOf<T, velocity>;
```

```
template<Velocity V>
constexpr price calc_fine(V speed);
```

Concept example

```
template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && std::same_as<typename T::dimension, D>;

template<typename T>
concept Velocity = QuantityOf<T, velocity>;
```

```
template<Velocity V>
constexpr price calc_fine(V speed);
```

Concept example

```
template<typename T>
concept Dimension =
    std::is_empty_v<T> &&
    detail::is_dimension<downcast_from<T>>;

template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && std::same_as<typename T::dimension, D>;

template<typename T>
concept Velocity = QuantityOf<T, velocity>;
```

```
template<Velocity V>
constexpr price calc_fine(V speed);
```

Concept example

```
template<typename T>
concept Quantity = detail::is_quantity<T>;

template<typename T>
concept Dimension =
    std::is_empty_v<T> &&
    detail::is_dimension<downcast_from<T>>;

template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && std::same_as<typename T::dimension, D>;

template<typename T>
concept Velocity = QuantityOf<T, velocity>;
```

```
template<Velocity V>
constexpr price calc_fine(V speed);
```

Not just a syntactic sugar for `enable_if` and `void_t`

Not just a syntactic sugar for `enable_if` and `void_t`

- Constraining *function template return types*

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

Not just a syntactic sugar for `enable_if` and `void_t`

- Constraining *function template return types*

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

- Constraining the *deduced types* of user's variables

```
const Velocity auto speed = avg_speed(220.km, 2.h);
```

Not just a syntactic sugar for `enable_if` and `void_t`

- Constraining *function template return types*

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

- Constraining the *deduced types* of user's variables

```
const Velocity auto speed = avg_speed(220.km, 2.h);
```

- Constraining *class template parameters* without the need to introduce new class template parameters

```
template<Dimension D, Ratio R = ratio<1>>
  requires (R::num * R::den > 0)
struct unit;
```

Concept categories

Concept categories

PREDICATES

```
template<class T>
concept signed_integral =
  integral<T> &&
  is_signed_v<T>;
```

Concept categories

PREDICATES

```
template<class T>
concept signed_integral =
  integral<T> &&
  is_signed_v<T>;
```

CAPABILITIES

```
template<class T>
concept swappable =
  requires(T& a, T& b) {
    ranges::swap(a, b);
  };
```

Concept categories

ABSTRACTIONS

```
template<class I>
concept bidirectional_iterator =
  forward_iterator<I> &&
  derived_from<ITER_CONCEPT(I), bidirectional_iterator_tag> &&
  requires(I i) {
    { --i } -> same_as<I&>;
    { i-- } -> same_as<I>;
  };
```

Concept categories

FAMILY OF INSTANTIATIONS

```
template<typename T>
concept Ratio = is_ratio<T>;
```

Concept categories

FAMILY OF INSTANTIATIONS

```
template<typename T>
inline constexpr bool is_ratio = false;

template<intmax_t Num, intmax_t Den>
inline constexpr bool is_ratio<ratio<Num, Den>> = true;

template<typename T>
concept Ratio = is_ratio<T>;
```

Concept categories

FAMILY OF INSTANTIATIONS

```
template<typename T>
inline constexpr bool is_ratio = false;

template<intmax_t Num, intmax_t Den>
inline constexpr bool is_ratio<ratio<Num, Den>> = true;

template<typename T>
concept Ratio = is_ratio<T>;
```

USAGE EXAMPLES

```
template<Ratio R1, Ratio R2>
using ratio_multiply = detail::ratio_multiply_impl<R1, R2>::type;
```

Concept categories

FAMILY OF INSTANTIATIONS

```
template<typename T>
inline constexpr bool is_ratio = false;

template<intmax_t Num, intmax_t Den>
inline constexpr bool is_ratio<ratio<Num, Den>> = true;

template<typename T>
concept Ratio = is_ratio<T>;
```

USAGE EXAMPLES

```
template<Ratio R1, Ratio R2>
using ratio_multiply = detail::ratio_multiply_impl<R1, R2>::type;
```

```
template<Unit U, Scalar Rep = double>
class quantity;
```

User experience: Compilation

```
constexpr units::Velocity auto avg_speed(units::Length auto d, units::Time auto t)
{
  return d * t;
}
```

User experience: Compilation

```
constexpr units::Velocity auto avg_speed(units::Length auto d, units::Time auto t)
{
  return d * t;
}
```

```
example.cpp: In instantiation of ‘constexpr units::Velocity avg_speed(D, T)
    [with D = units::quantity<units::kilometre>; T = units::quantity<units::hour>]’:
example.cpp:49:49:   required from here
example.cpp:34:14: error: placeholder constraints not satisfied
    34 |   return d * t;
       |              ^
include/units/dimensions/velocity.h:34:16: note: within ‘template<class T> concept units::Velocity<T>
    [with T = units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1, 1>,
                units::exp<units::base_dim_time, 1, 1> >, units::ratio<3600000, 1> >, double>]’
    34 |   concept Velocity = Quantity<T> && std::same_as<typename T::dimension, velocity>;
       |           ^~~~~~~~
include/stl2/detail/concepts/core.hpp:37:15: note: within ‘template<class T, class U> concept std::same_as<T, U>
    [with T = units::dimension<units::exp<units::base_dim_length, 1, 1>, units::exp<units::base_dim_time, 1, 1> >;
          U = units::velocity]’
    37 |  META_CONCEPT same_as = meta::Same<T, U> && meta::Same<U, T>;
       |               ^~~~~~~
...
```

User experience: Compilation

```
constexpr units::Velocity auto avg_speed(units::Length auto d, units::Time auto t)
{
  return d * t;
}
```

```
include/meta/meta_fwd.hpp:224:18: note: within ‘template<class T, class U> concept meta::Same<T, U>
    [with T = units::dimension<units::exp<units::base_dim_length, 1, 1>, units::exp<units::base_dim_time, 1, 1> >;
          U = units::velocity]’
   224 |     META_CONCEPT Same =
       |                  ^~~~
include/meta/meta_fwd.hpp:224:18: note: ‘meta::detail::barrier’ evaluated to false
include/meta/meta_fwd.hpp:224:18: note: within ‘template<class T, class U> concept meta::Same<T, U>
    [with T = units::velocity;
          U = units::dimension<units::exp<units::base_dim_length, 1, 1>, units::exp<units::base_dim_time, 1, 1> >]’
include/meta/meta_fwd.hpp:224:18: note: ‘meta::detail::barrier’ evaluated to false
```

User experience: Compilation

```
constexpr units::Velocity auto avg_speed(units::Length auto d, units::Time auto t)
{
  return d * t;
}
```

```
include/meta/meta_fwd.hpp:224:18: note: within ‘template<class T, class U> concept meta::Same<T, U>
    [with T = units::dimension<units::exp<units::base_dim_length, 1, 1>, units::exp<units::base_dim_time, 1, 1> >;
          U = units::velocity]’
   224 |     META_CONCEPT Same =
       |                  ^~~~
include/meta/meta_fwd.hpp:224:18: note: ‘meta::detail::barrier’ evaluated to false
include/meta/meta_fwd.hpp:224:18: note: within ‘template<class T, class U> concept meta::Same<T, U>
    [with T = units::velocity;
          U = units::dimension<units::exp<units::base_dim_length, 1, 1>, units::exp<units::base_dim_time, 1, 1> >]’
include/meta/meta_fwd.hpp:224:18: note: ‘meta::detail::barrier’ evaluated to false
```

Concepts support in C++ compilers is still experimental and largely based on Concepts TS. We can expect even better diagnostics in the future.

Benefits of using C++ Concepts

Benefits of using C++ Concepts

1. Clearly state the design intent of the interface of a class/function template

Benefits of using C++ Concepts

1. Clearly state the design intent of the interface of a class/function template
2. Embedded in a template signature

Benefits of using C++ Concepts

1. Clearly **state the design intent** of the interface of a class/function template
2. **Embedded in a template signature**
3. **Simplify and extend SFINAE**
   - *disabling specific overloads* for specific constraints (compared to **`std::enable_if`**)
   - constraints *based on dependent member types/functions existence* (compared to **`void_t`**)
   - *no dummy template parameters* allocated for SFINAE needs
   - constraining *function return types and deduced types of user's variables*

Benefits of using C++ Concepts

1. Clearly **state the design intent** of the interface of a class/function template
2. **Embedded in a template signature**
3. **Simplify and extend SFINAE**
   - *disabling specific overloads* for specific constraints (compared to **`std::enable_if`**)
   - constraints *based on dependent member types/functions existence* (compared to **`void_t`**)
   - *no dummy template parameters* allocated for SFINAE needs
   - constraining *function return types and deduced types of user's variables*
4. Greatly **improve error messages**
   - raise *compilation error* about failed compile-time contract *before instantiating a template*
   - no more errors from *deeply nested implementation details of a function template*

PERFORMANCE

Compile-time benchmarking with Metabench

- Started in 2016 by *Louis Dionne*

Compile-time benchmarking with Metabench

- Started in 2016 by *Louis Dionne*
- *CMake module* that simplifies compile-time microbenchmarking

Compile-time benchmarking with Metabench

- Started in 2016 by *Louis Dionne*
- *CMake module* that simplifies compile-time microbenchmarking
- Can be used *to benchmark precise parts* of a C++ file, such as the instantiation of a single function

Compile-time benchmarking with Metabench

- Started in 2016 by *Louis Dionne*
- *CMake module* that simplifies compile-time microbenchmarking
- Can be used *to benchmark precise parts* of a C++ file, such as the instantiation of a single function
- http://metaben.ch *compares the performance* of several MPL algorithms implemented in various libraries

A short intro to Metabench: CMake

```
metabench_add_dataset(metabench.data.ratio.create.std_ratio
    all_std_ratio.cpp.erb "[10, 50, 100, 250, 500, 750, 1000, 1500, 2000]"
    NAME "std::ratio"
)
metabench_add_dataset(metabench.data.ratio.create.ratio_type_constexpr
    all_ratio_type_constexpr.cpp.erb "[10, 50, 100, 250, 500, 750, 1000, 1500, 2000]"
    NAME "ratio constexpr"
)
metabench_add_dataset(metabench.data.ratio.create.ratio_nttp
    all_ratio_nttp.cpp.erb "[10, 50, 100, 250, 500, 750, 1000, 1500, 2000]"
    NAME "ratio NTTP"
)
metabench_add_chart(metabench.chart.ratio.all
    TITLE "Creation of 2*N ratios"
    SUBTITLE "(smaller is better)"
    DATASETS
        metabench.data.ratio.create.std_ratio
        metabench.data.ratio.create.ratio_type_constexpr
        metabench.data.ratio.create.ratio_nttp
)
```

A short intro to Metabench: ERB file

CREATE_RATIO_TYPE_CONSTEXPR.CPP.ERB

```
#include "ratio_type_constexpr.h"

<% (1..n).each do |i| %>
struct test<%= i %> {
#if defined(METABENCH)
  using r1 = units::ratio<<%= 2 * i - 1 %>, <%= 2 * n %>>;
  using r2 = units::ratio<<%= 2 * i %>, <%= 2 * n %>>;
#else
  using r1 = void;
  using r2 = void;
#endif
};
<% end %>

int main() {}
```

A short intro to Metabench: Generated C++ code

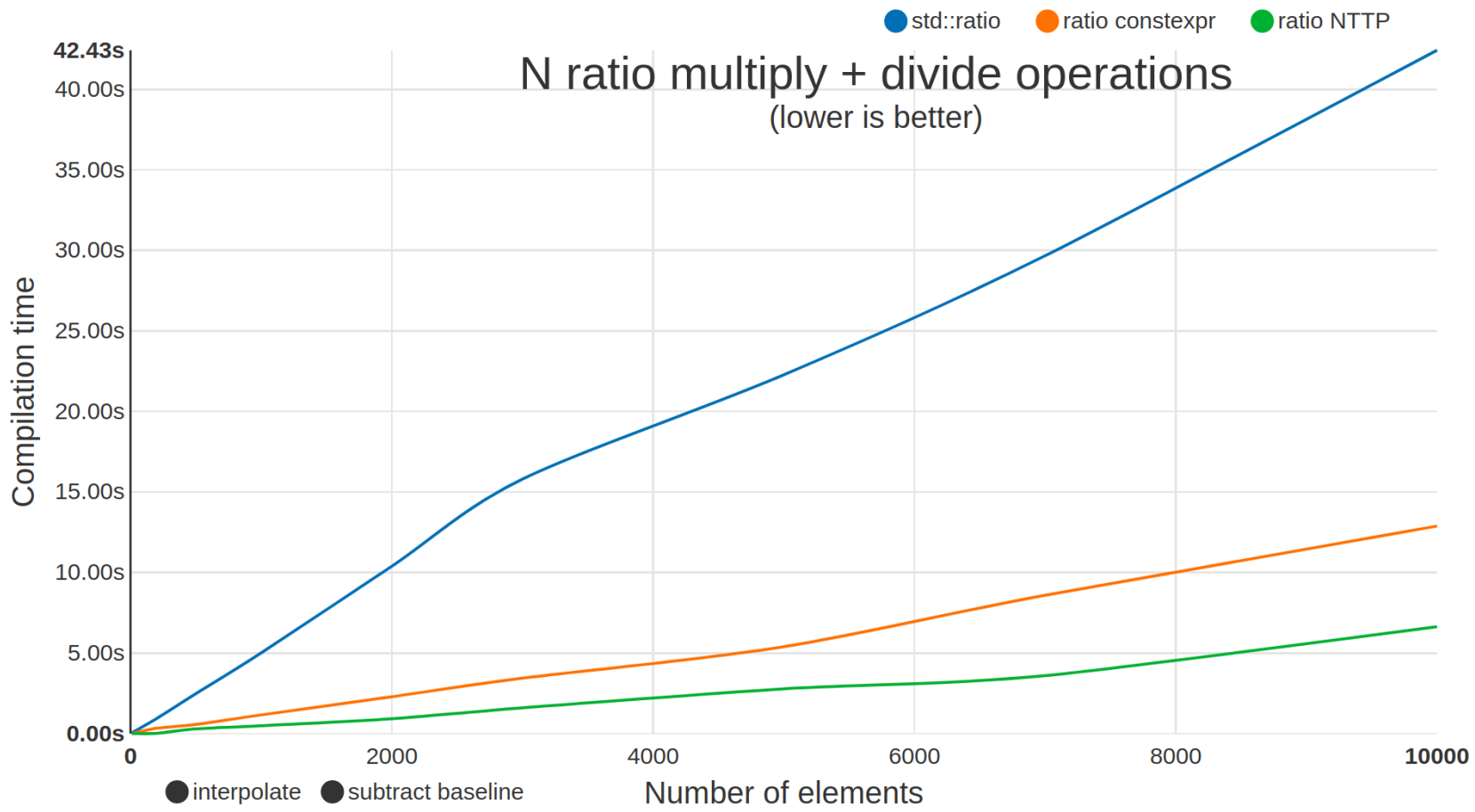
```
#include "ratio_type_constexpr.h"

struct test1 {
#if defined(METABENCH)
  using r1 = std::ratio<1, 20>;
  using r2 = std::ratio<2, 20>;
#else
  using r1 = void;
  using r2 = void;
#endif
};

struct test2 {
#if defined(METABENCH)
  using r1 = std::ratio<3, 20>;
  using r2 = std::ratio<4, 20>;
#else
  using r1 = void;
  using r2 = void;
#endif
};

// ...

int main() {}
```

A short intro to Metabench: Generated C++ code

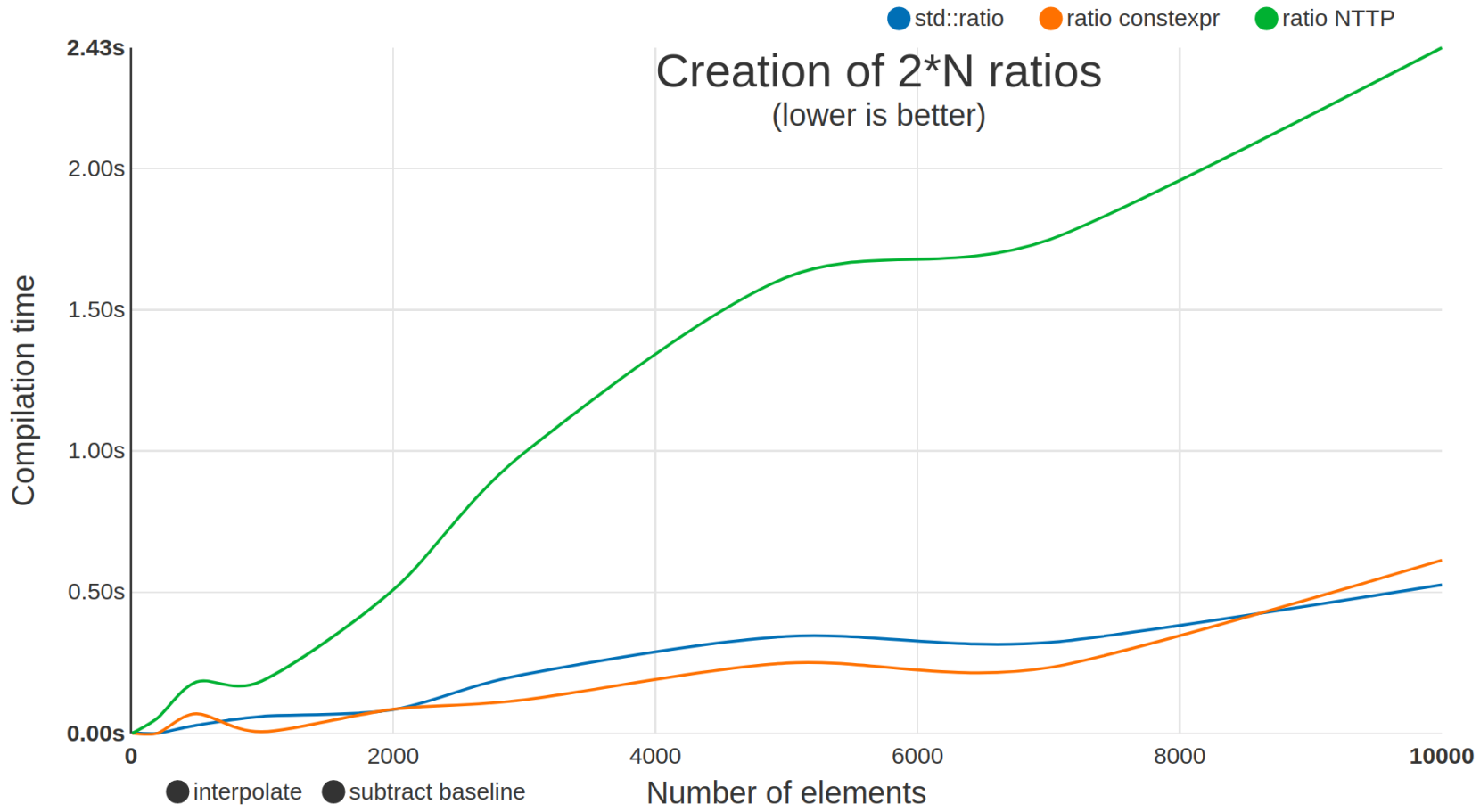
```cpp
#include "ratio_type_constexpr.h"

struct test1 {
#if defined(METABENCH)
  using r1 = std::ratio<1, 20>;
  using r2 = std::ratio<2, 20>;
#else
  using r1 = void;
  using r2 = void;
#endif
};

struct test2 {
#if defined(METABENCH)
  using r1 = std::ratio<3, 20>;
  using r2 = std::ratio<4, 20>;
#else
  using r1 = void;
  using r2 = void;
#endif
};

// ...

int main() {}
```

```cpp
#include "ratio_nttp.h"

struct test1 {
#if defined(METABENCH)
  static constexpr units::ratio r1{1, 20};
  static constexpr units::ratio r2{2, 20};
#else
  static constexpr bool r1 = false;
  static constexpr bool r2 = false;
#endif
};

struct test2 {
#if defined(METABENCH)
  static constexpr units::ratio r1{3, 20};
  static constexpr units::ratio r2{4, 20};
#else
  static constexpr bool r1 = false;
  static constexpr bool r2 = false;
#endif
};

// ...

int main() {}
```

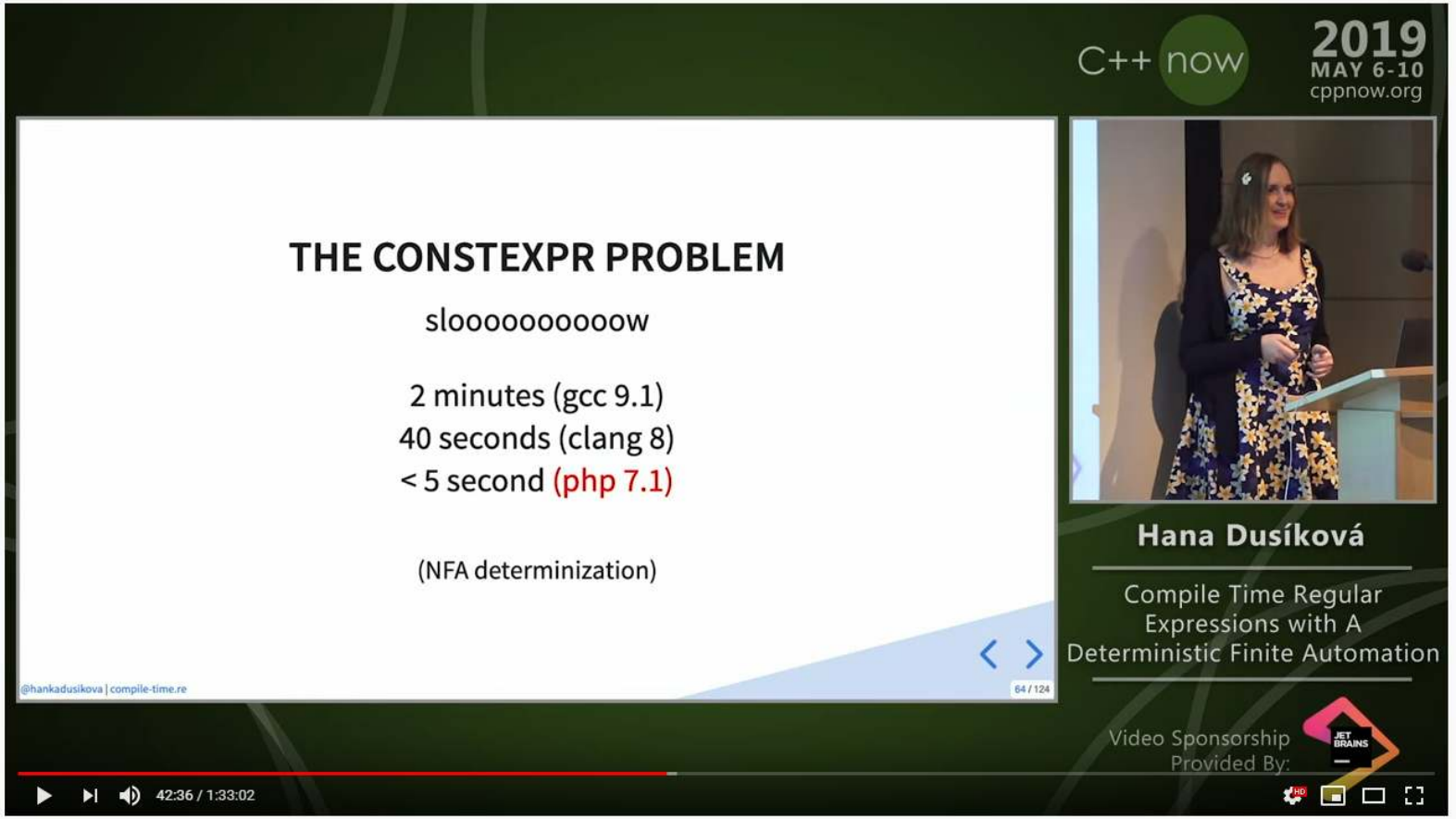
`ratio` performance



`ratio` performance



Current `constexpr` QoI is slow



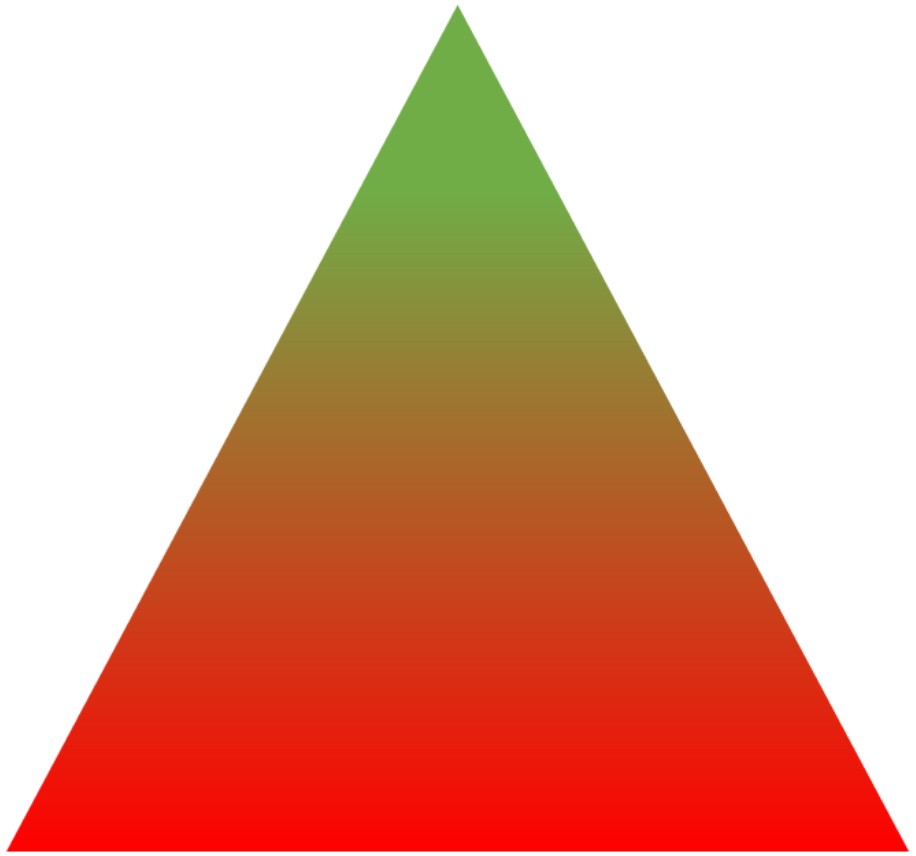
C++Now 2019: Hana Dusíková “Compile Time Regular Expressions with A Deterministic Finite Automaton”

The Rule of Chiel

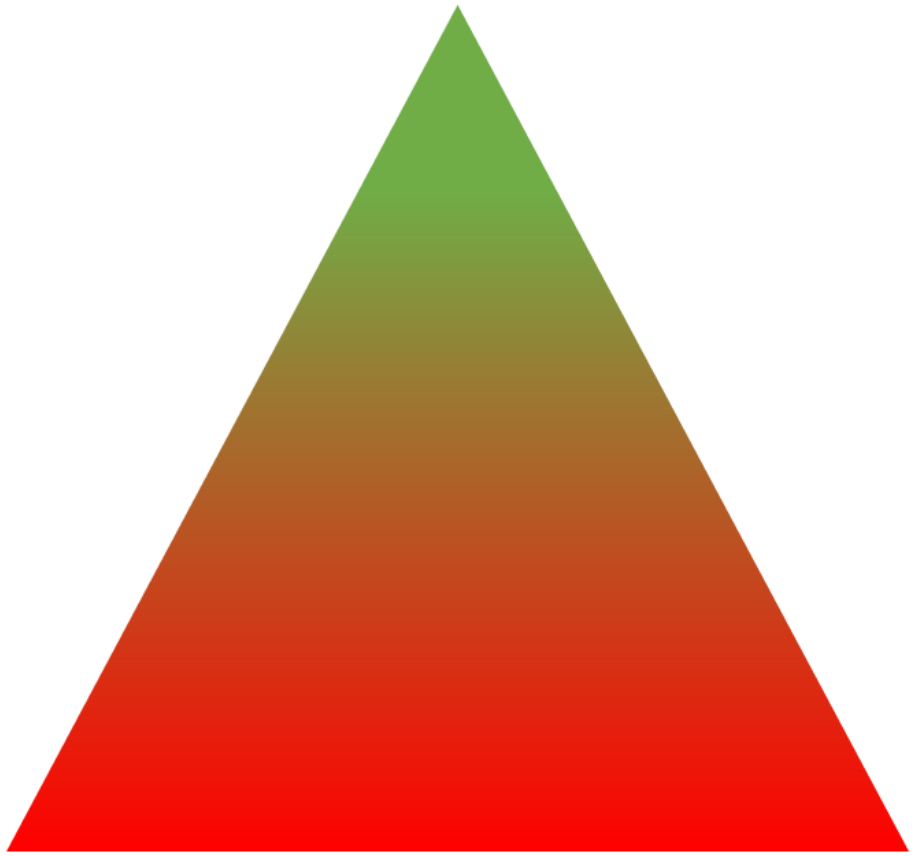
The Rule of Chiel

code::dive 2017 – Odin Holmes – The fastest template metaprogramming in the West

Cost of operations: The Rule of Chiel

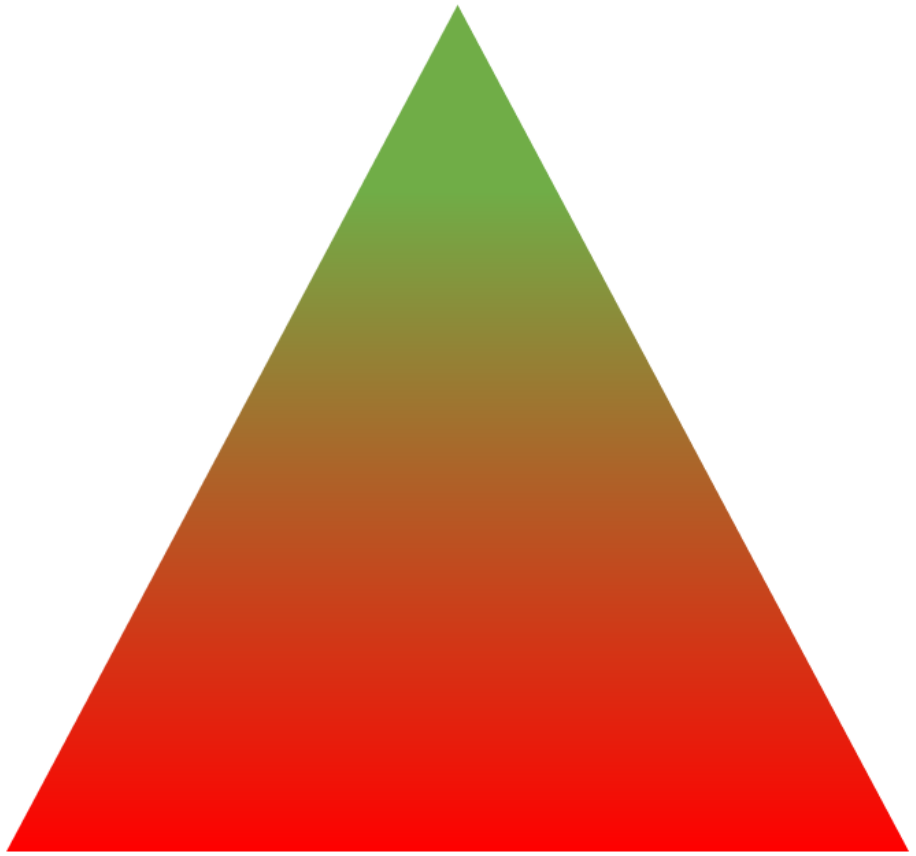


Cost of operations: The Rule of Chiel



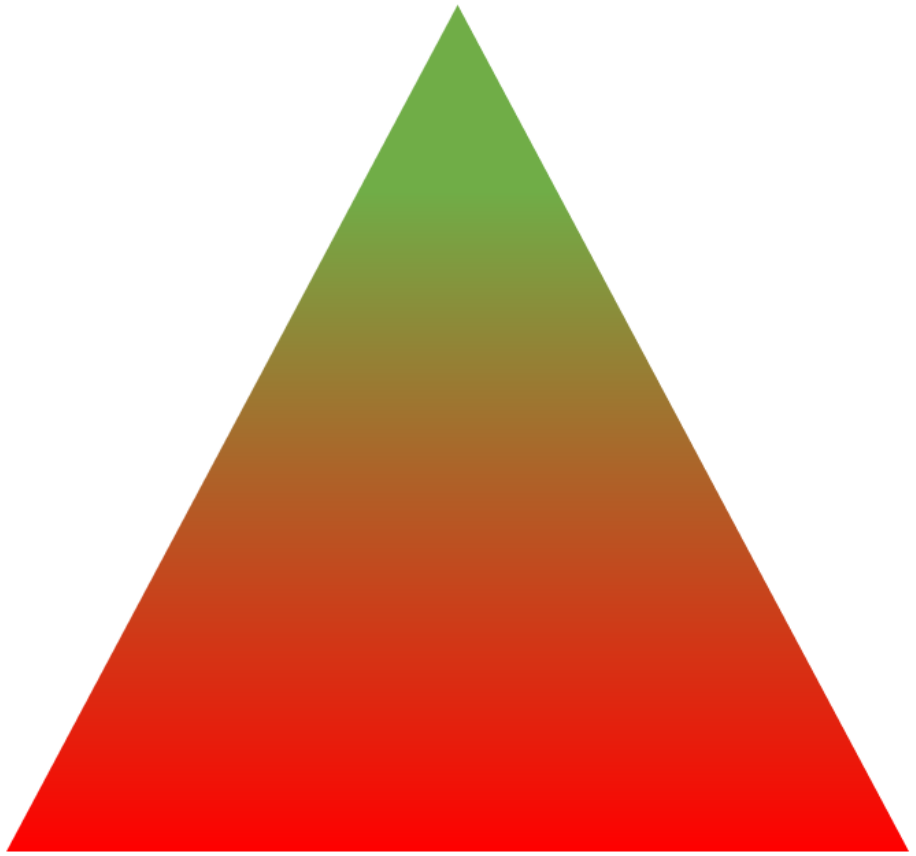
1. Looking up a memoized type

Cost of operations: The Rule of Chiel



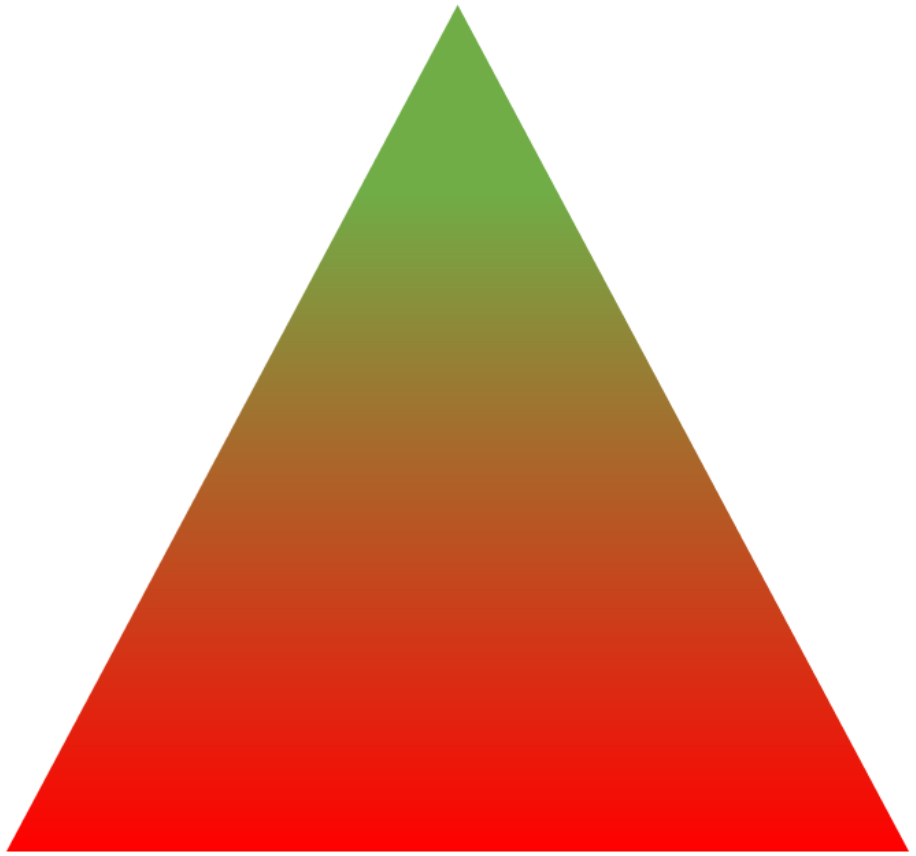
1. Looking up a memoized type
2. Adding a parameter to an alias call

Cost of operations: The Rule of Chiel



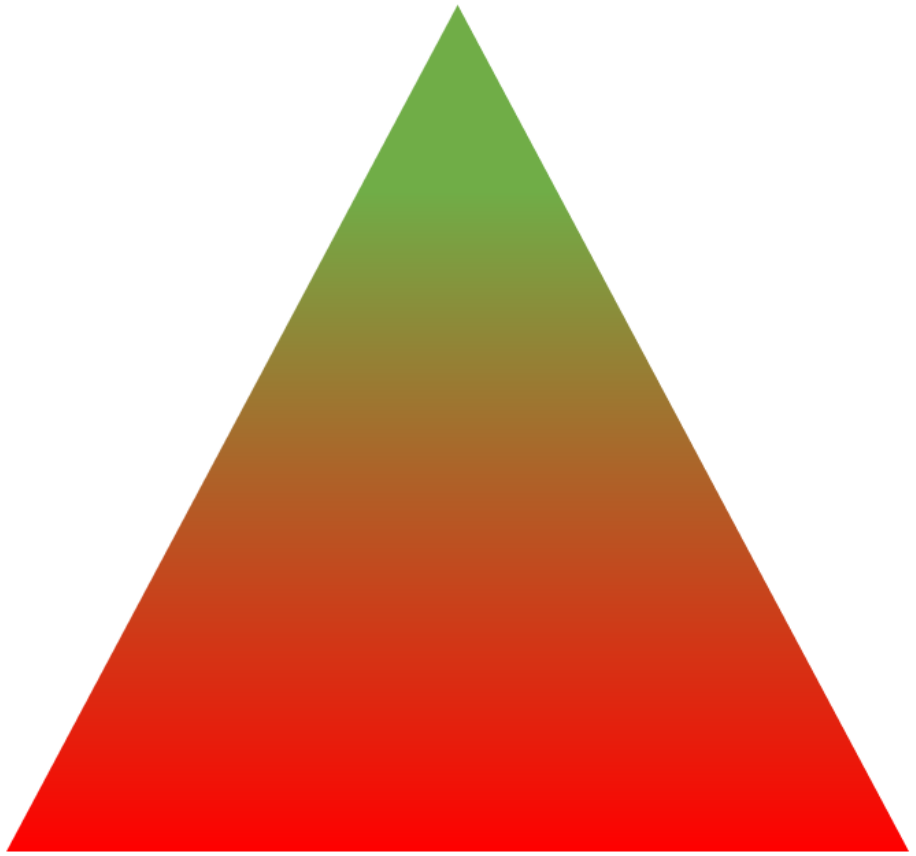
1. Looking up a memoized type
2. Adding a parameter to an alias call
3. Adding a parameter to a type

Cost of operations: The Rule of Chiel



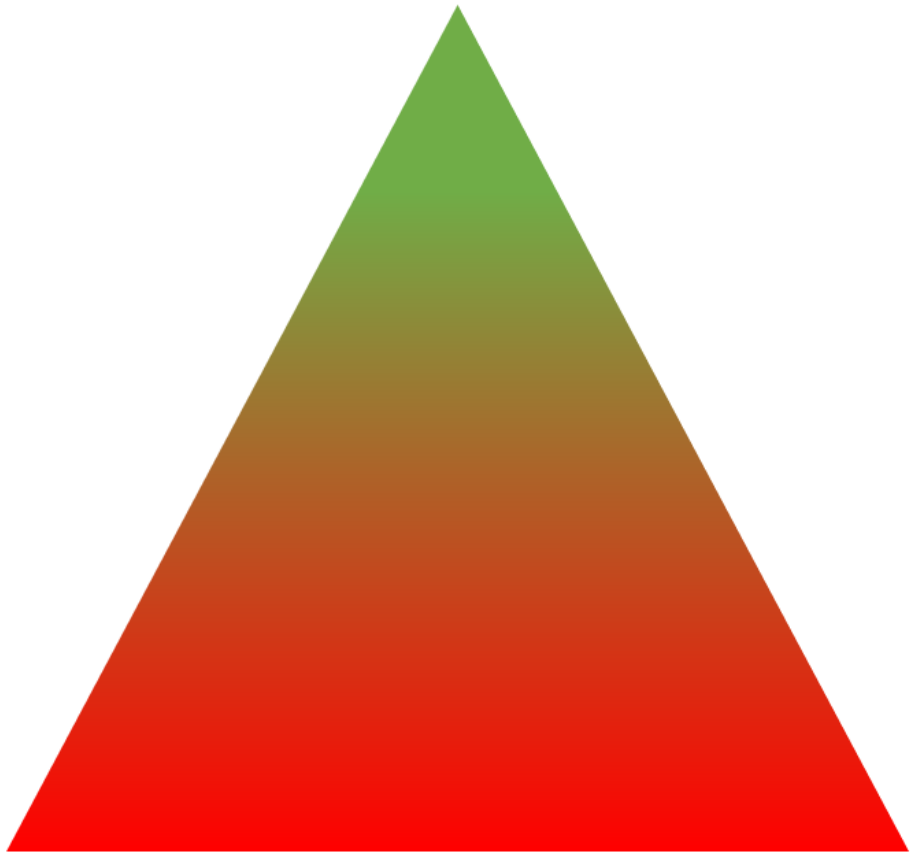
1. Looking up a memoized type
2. Adding a parameter to an alias call
3. Adding a parameter to a type
4. Calling an alias

Cost of operations: The Rule of Chiel



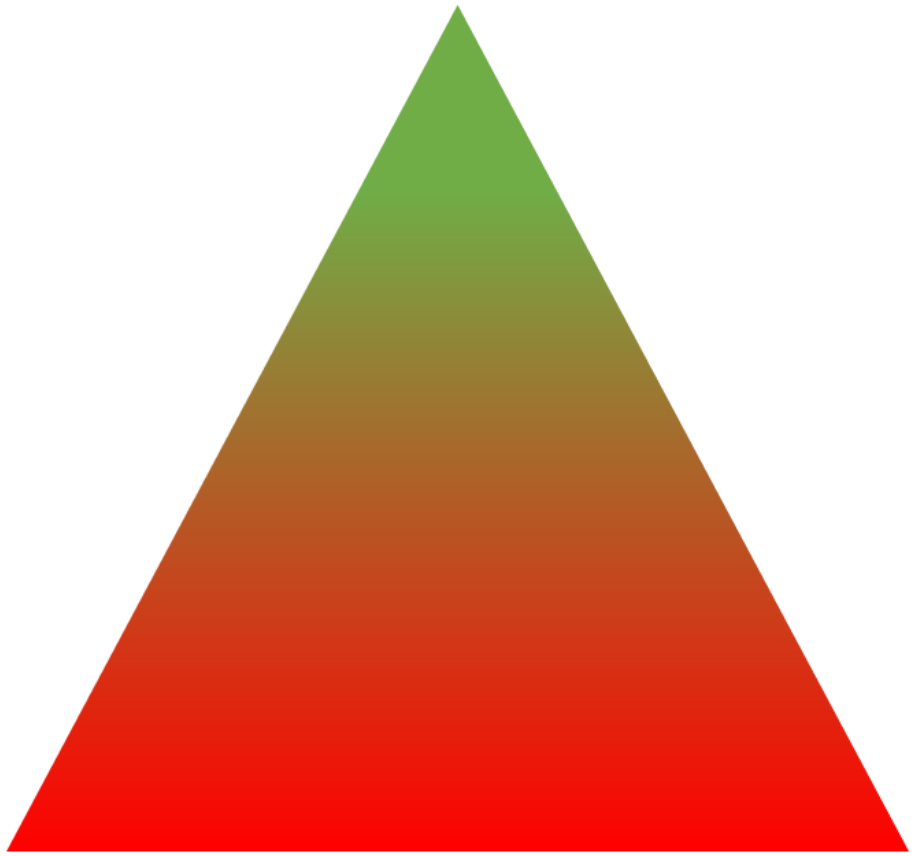
1. Looking up a memoized type
2. Adding a parameter to an alias call
3. Adding a parameter to a type
4. Calling an alias
5. Instantiating a class

Cost of operations: The Rule of Chiel



1. Looking up a memoized type
2. Adding a parameter to an alias call
3. Adding a parameter to a type
4. Calling an alias
5. Instantiating a class
6. Instantiating a function template

Cost of operations: The Rule of Chiel



1. Looking up a memoized type
2. Adding a parameter to an alias call
3. Adding a parameter to a type
4. Calling an alias
5. Instantiating a class
6. Instantiating a function template
7. SFINAE

std::conditional<B, T, F>

TRADITIONAL

```
template<bool B, class T, class F>
struct conditional {
  using type = T;
};

template<class T, class F>
struct conditional<false, T, F> {
  using type = F;
};

template<bool B, class T, class F>
using conditional_t = conditional<B,T,F>::type;
```

std::conditional<B, T, F>

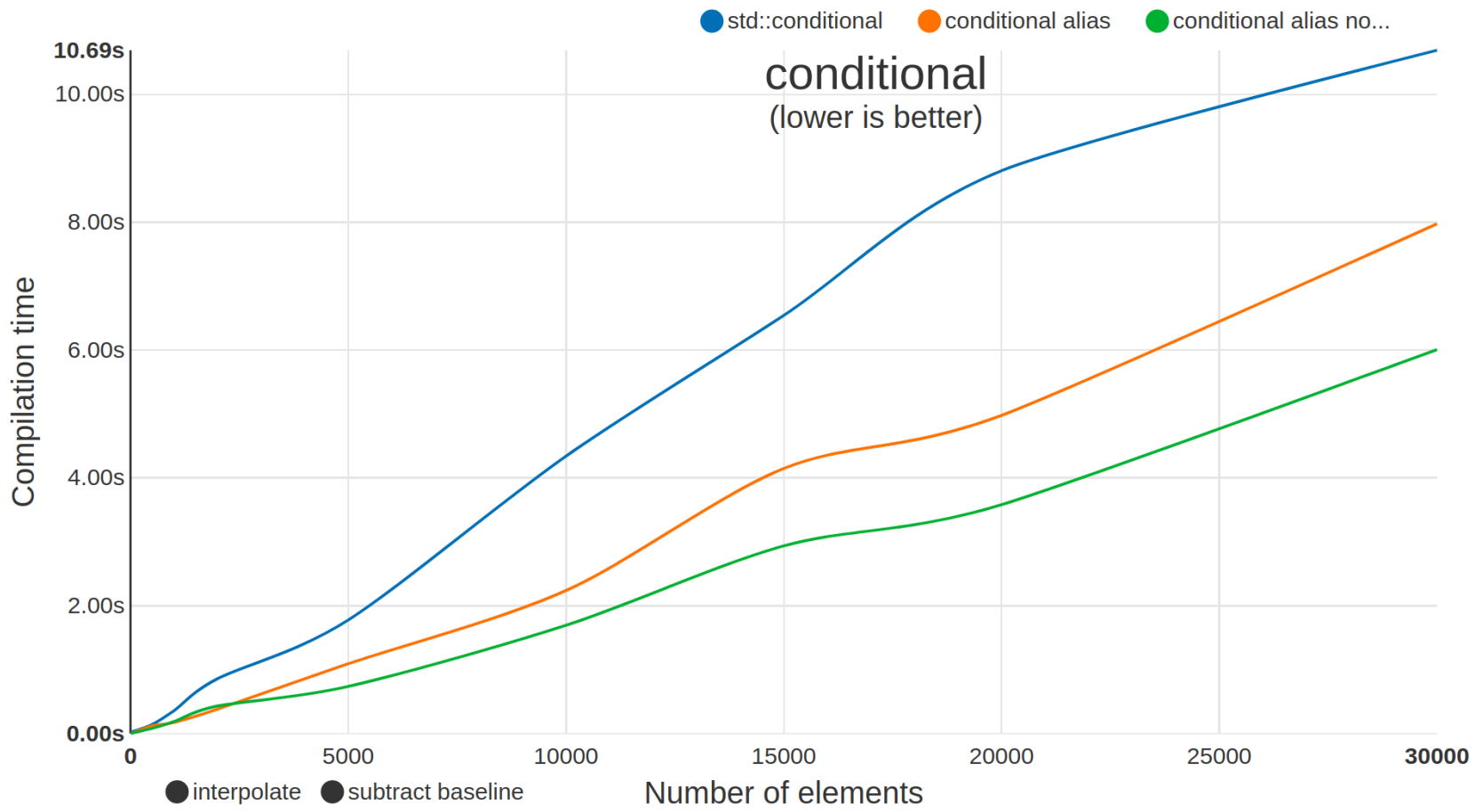
TRADITIONAL

```
template<bool B, class T, class F>
struct conditional {
  using type = T;
};

template<class T, class F>
struct conditional<false, T, F> {
  using type = F;
};

template<bool B, class T, class F>
using conditional_t = conditional<B,T,F>::type;
```

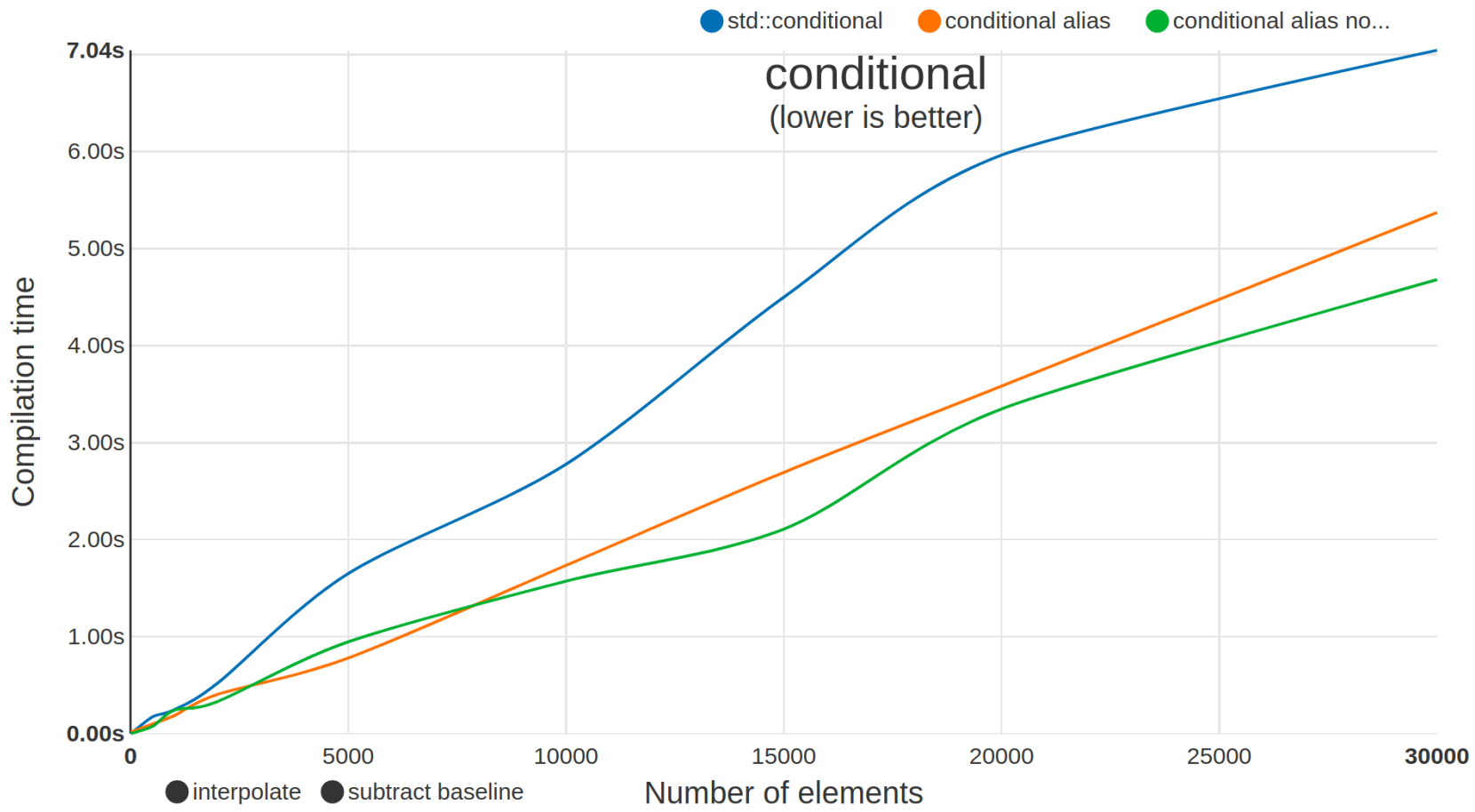
WITH ALIAS TEMPLATE

```
template<bool>
struct conditional {
  template<typename T, typename F>
  using type = F;
};

template<>
struct conditional<true> {
  template<typename T, typename F>
  using type = T;
};

template<bool B, typename T, typename F>
using conditional_t =
  conditional<B>::template type<T, F>;
```

`conditional` performance (gcc-9)



`conditional` performance (clang-8)



std::is_same<T, U>

TRADITIONAL

```
template<class T, class U>
struct is_same : std::false_type {};

template<class T>
struct is_same<T, T> : std::true_type {};
```

```
template<class T, class U>
inline constexpr bool is_same_v =
  is_same<T, U>::value;
```

std::is_same<T, U>

TRADITIONAL

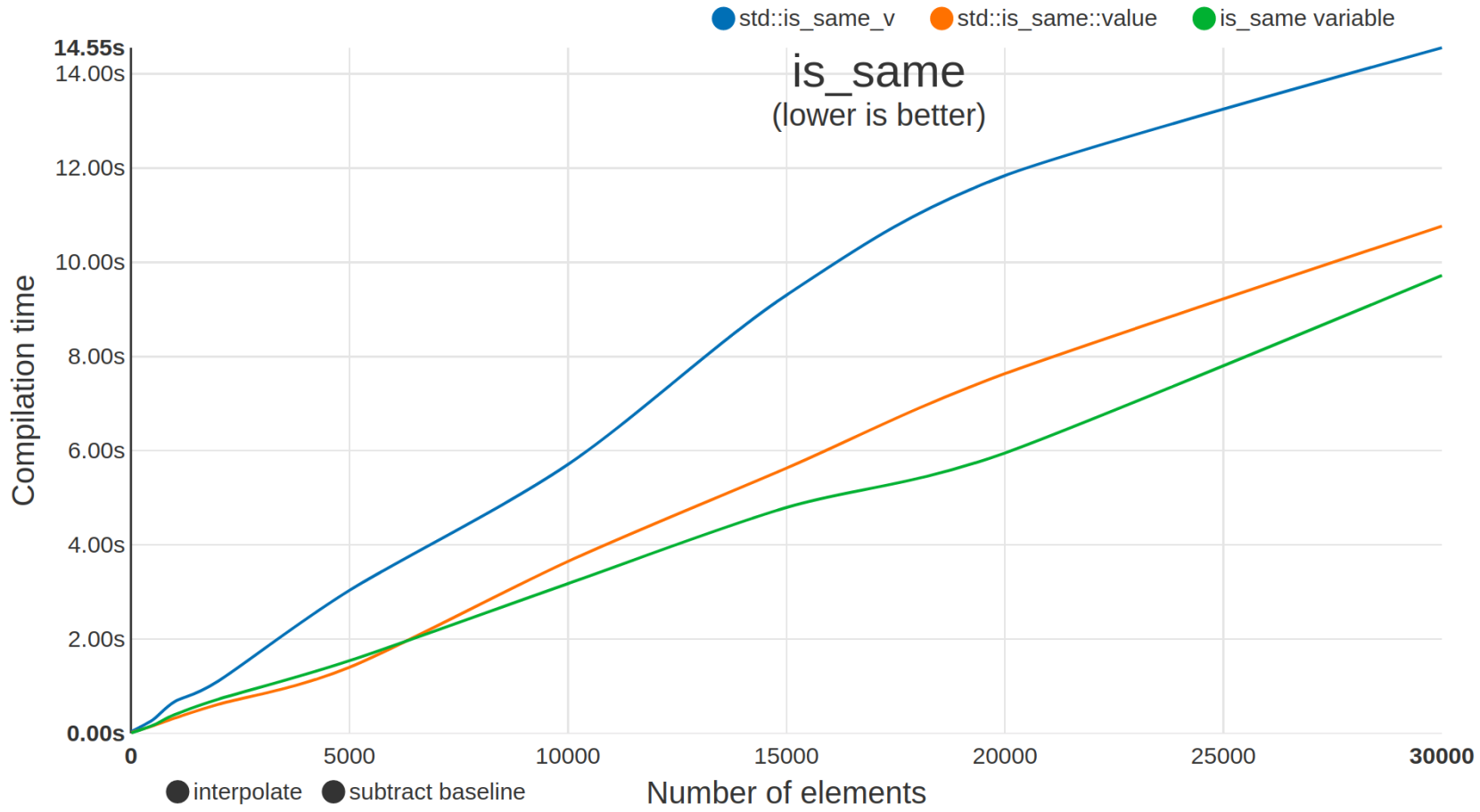
```
template<class T, class U>
struct is_same : std::false_type {};

template<class T>
struct is_same<T, T> : std::true_type {};
```

```
template<class T, class U>
inline constexpr bool is_same_v =
  is_same<T, U>::value;
```

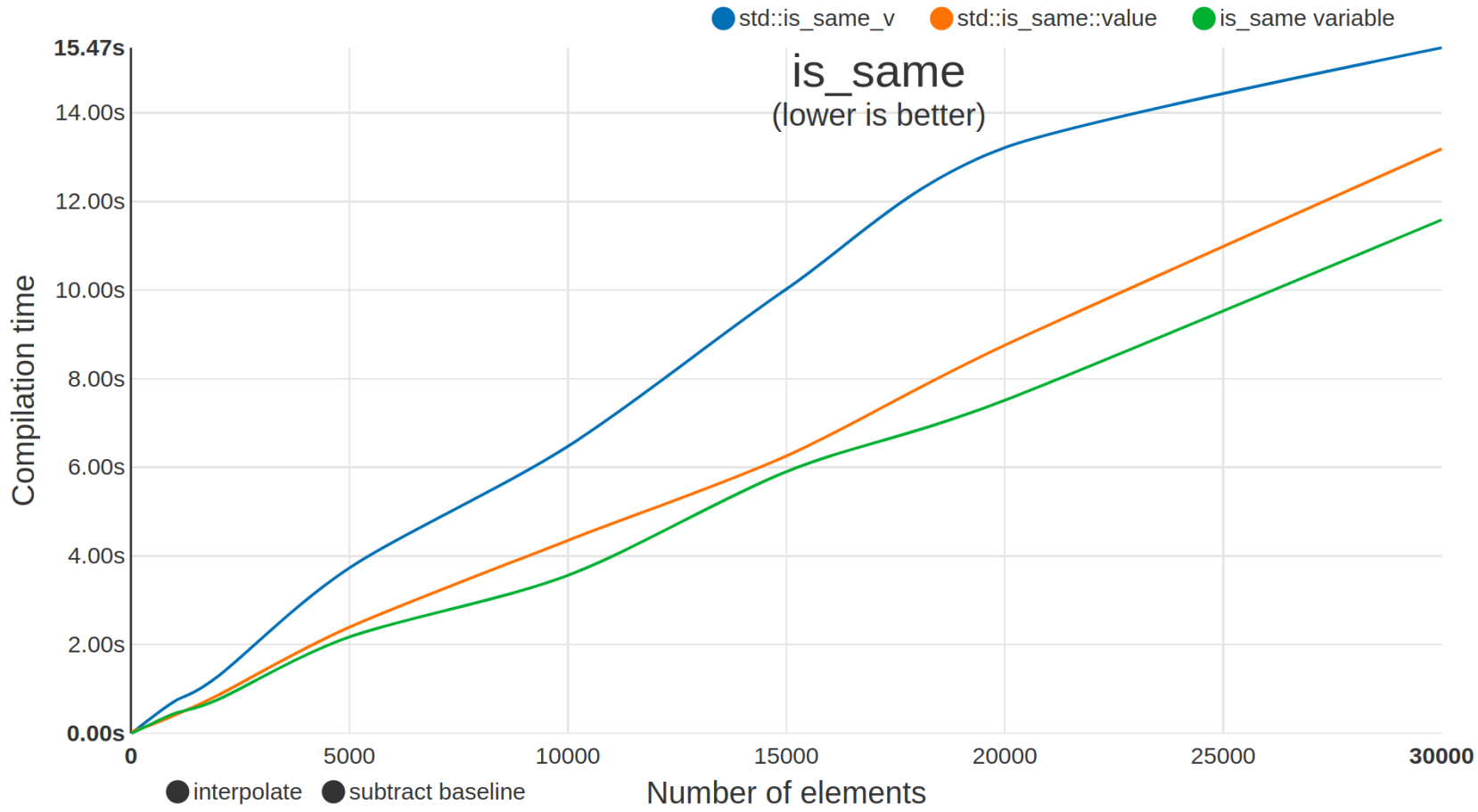
USING VARIABLE TEMPLATES

```
template<class T, class U>
inline constexpr bool is_same = false;

template<class T>
inline constexpr bool is_same<T, T> = true;
```

`is_same` performance (gcc-9)



`is_same` performance (clang-8)



What about C++ Concepts?

What about C++ Concepts?

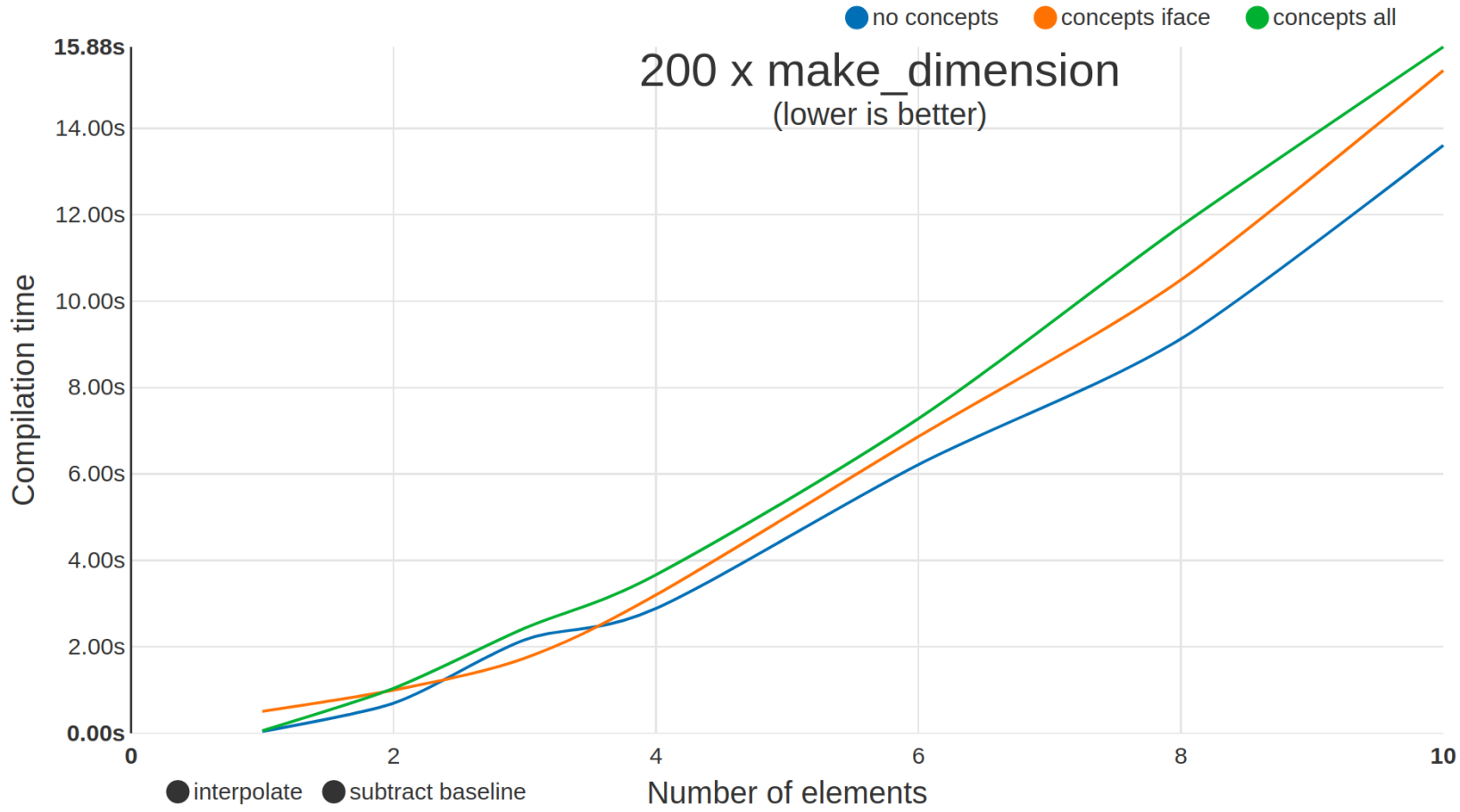
- C++ Concepts are great and *not too expensive at compile-time*

What about C++ Concepts?

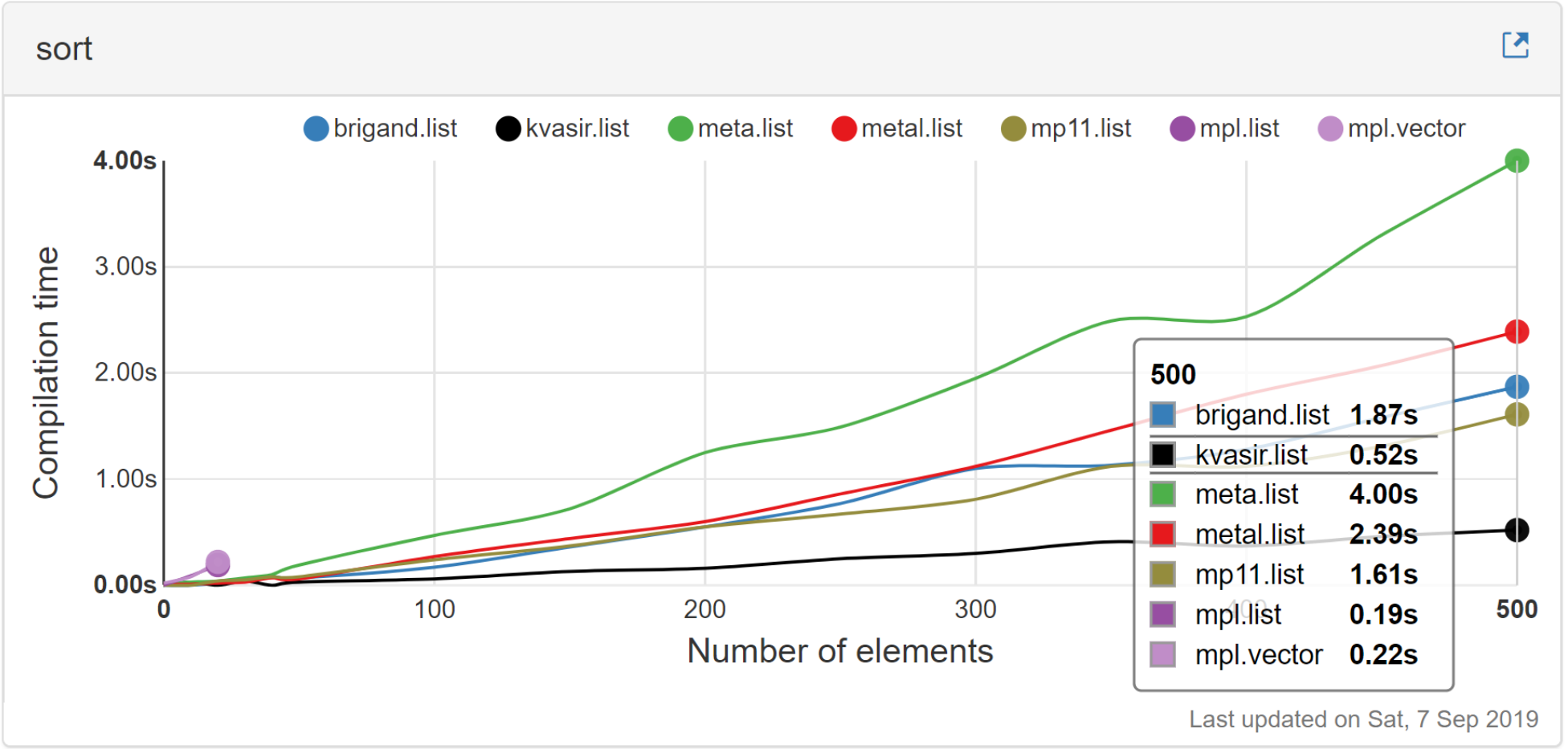
- C++ Concepts are great and *not too expensive at compile-time*
- Compile-time performance impact might be *limited by using concepts only in the user interfaces*

```
template<Exponent... Es>
struct dimension : downcast_base<dimension<Es...>> {};

namespace detail {
  template<typename D1, typename D2>
  struct dimension_divide;

  template<typename... E1, typename... E2>
  struct dimension_divide<dimension<E1...>, dimension<E2...>>
      : dimension_multiply<dimension<E1...>, dimension<exp_invert_t<E2>...>> {
  };
}

template<Dimension D1, Dimension D2>
using dimension_divide_t = detail::dimension_divide<typename D1::base_type,
                                                    typename D2::base_type>::type;
```

C++ Concepts performance



More info on MPL performance on http://metaben.ch



TAKEAWAYS

Rethinking C++ templates

Rethinking C++ templates

1. Think about **end users' experience** and not only about your convenience as a developer

Rethinking C++ templates

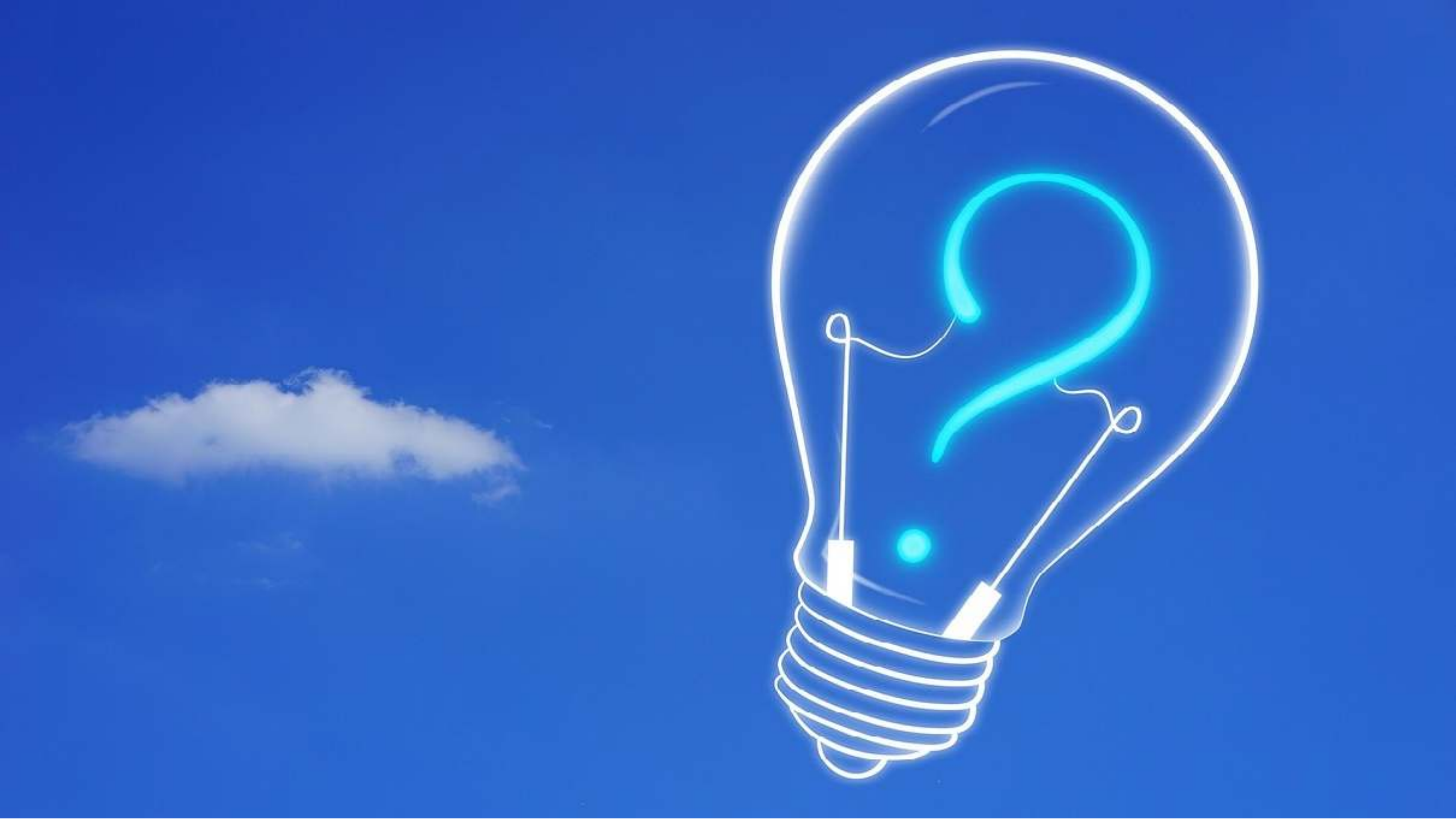
1. Think about **end users' experience** and not only about your convenience as a developer
2. Use **C++ Concepts** to
   - express compile-time contracts
   - improve productivity
   - improve compile-time errors

Rethinking C++ templates

1. Think about **end users' experience** and not only about your convenience as a developer
2. Use **C++ Concepts** to
   - express compile-time contracts
   - improve productivity
   - improve compile-time errors
3. **Use NTTPs** when a template parameter represents a value rather than a type

Rethinking C++ templates

1. Think about **end users' experience** and not only about your convenience as a developer
2. Use **C++ Concepts** to
   - express compile-time contracts
   - improve productivity
   - improve compile-time errors
3. **Use NTTPs** when a template parameter represents a value rather than a type
4. **Optimize compile-time performance**
   - MPL is free at run-time but can be really expensive at compile-time
   - the same MPL algorithm may be implemented using different techniques and take different amount of time to compile




CAUTION
Programming
is addictive
(and too much fun)