<epam>

Rethinking the Way We Do Templates in C++

Mateusz Pusz
November 20, 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

Developer's experience: Boost.Units

```
namespace bu = 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;
}
```

Developer's experience: Boost.Units

```
namespace bu = 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;
}
```

Thanks to template aliases developers have really comfortable environment to develop their code

User's 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's 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's 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's 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;
            ~~^~~
```

This is only the very first line of the error log...

User's 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's 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 ;-)

Developer's experience: NHolthaus Units

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

Again, nice developer's experience thanks to aliases

User's 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's 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's 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's 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.");
                ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

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*

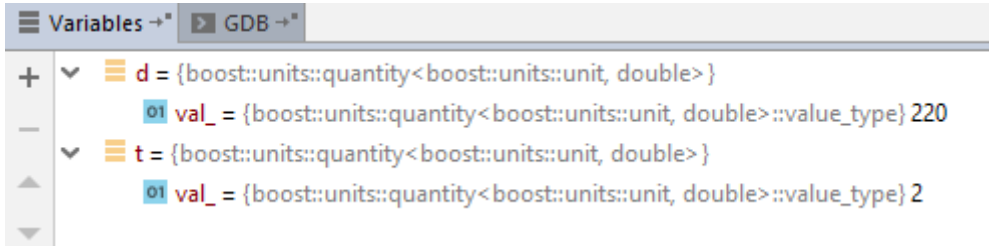
In case of the physical units library we have to rethink the way we do template metaprogramming!

User's 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's 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's 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's 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] {
...
```

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**

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 and non-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 and non-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*

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 and non-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*

A good fit for simple empty types like **`dimension`** and **`unit`**

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 from division of **`length`** by **`time`**?

Downcasting facility (Version 1.0)

```
struct length : make_dimension_t<exp<base_dim_length, 1>> {};
template<>
struct downcast_traits<downcast_base_t<length>> : std::type_identity_t<length> {};

struct metre : unit<length> {};
template<>
struct downcast_traits<downcast_base_t<metre>> : std::type_identity_t<metre> {};
```

Downcasting facility (Version 1.0)

```
struct length : make_dimension_t<exp<base_dim_length, 1>> {};
template<>
struct downcast_traits<downcast_base_t<length>> : std::type_identity_t<length> {};

struct metre : unit<length> {};
template<>
struct downcast_traits<downcast_base_t<metre>> : std::type_identity_t<metre> {};
```

```
struct time : make_dimension_t< exp<base_dim_time, -1>> {};
template<>
struct downcast_traits<downcast_base_t<time>> : std::type_identity_t<time> {};

struct second : unit<time> {};
template<>
struct downcast_traits<downcast_base_t<second>> : std::type_identity_t<second> {};
```

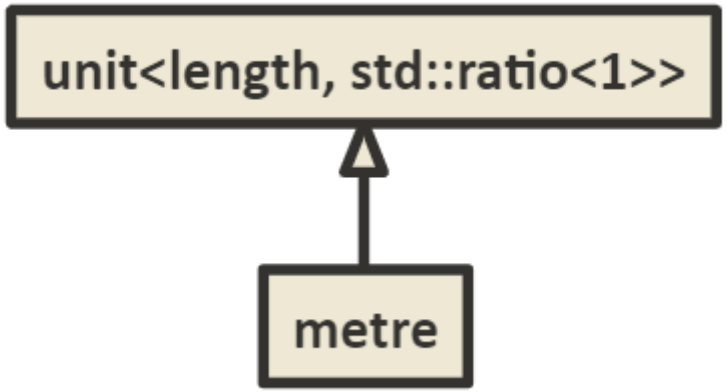
Downcasting facility (Version 1.0)

```
struct length : make_dimension_t<exp<base_dim_length, 1>> {};
template<>
struct downcast_traits<downcast_base_t<length>> : std::type_identity_t<length> {};

struct metre : unit<length> {};
template<>
struct downcast_traits<downcast_base_t<metre>> : std::type_identity_t<metre> {};
```

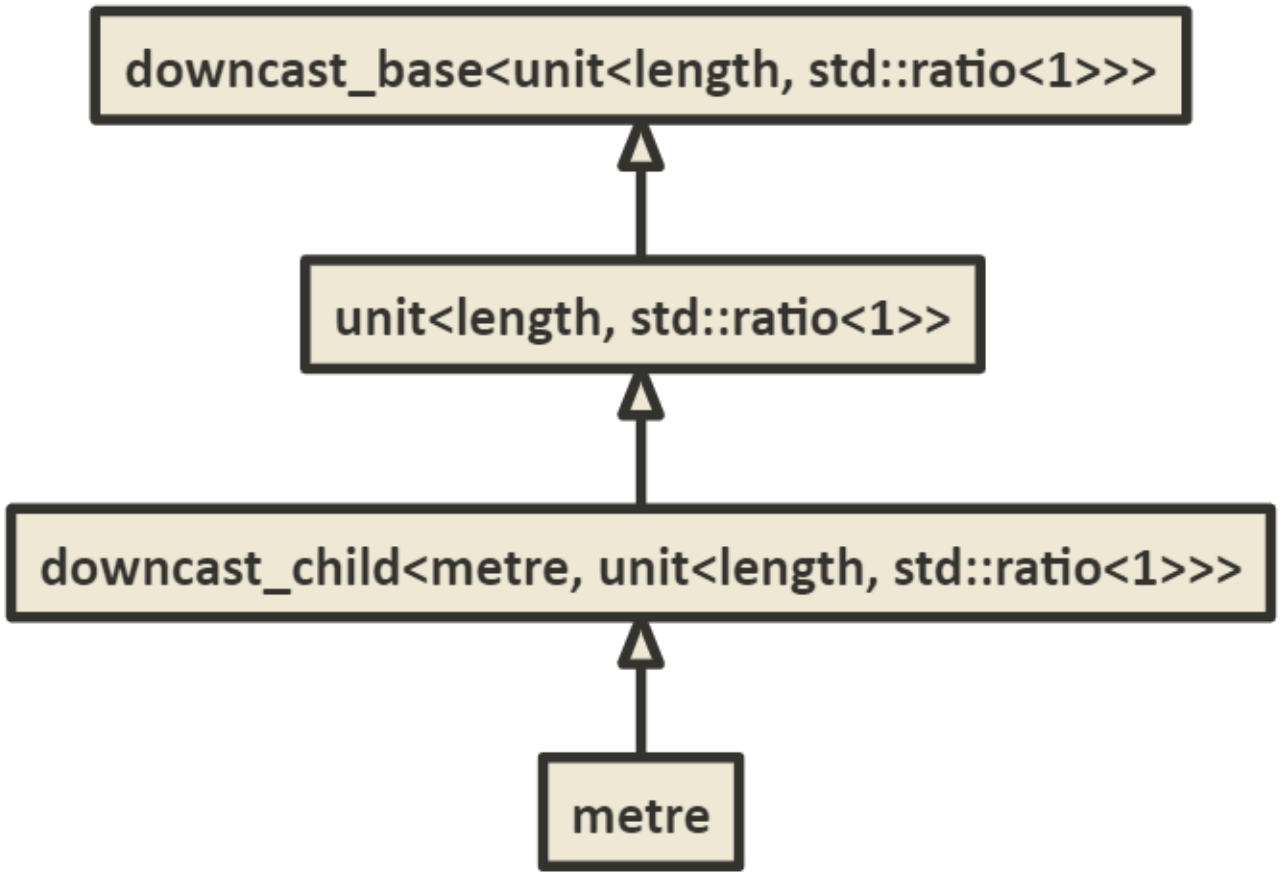
```
struct time : make_dimension_t< exp<base_dim_time, -1>> {};
template<>
struct downcast_traits<downcast_base_t<time>> : std::type_identity_t<time> {};

struct second : unit<time> {};
template<>
struct downcast_traits<downcast_base_t<second>> : std::type_identity_t<second> {};
```

```
struct velocity : make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>> {};
template<>
struct downcast_traits<downcast_base_t<velocity>> : std::type_identity_t<velocity> {};

struct metre_per_second : derived_unit<velocity, metre, second> {};
template<>
struct downcast_traits<downcast_base_t<metre_per_second>> : std::type_identity_t<metre_per_second> {};
```

Downcasting facility (Version 2.0): Overview



Downcasting facility (Version 2.0): Overview



Downcasting facility: Design

```
template<typename BaseType>
struct downcast_base {
  using base_type = BaseType;
  friend auto downcast_guide(downcast_base);  // declaration only (no implementation)
};
```

Downcasting facility: Design

```
template<typename BaseType>
struct downcast_base {
  using base_type = BaseType;
  friend auto downcast_guide(downcast_base);  // declaration only (no implementation)
};
```

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

Downcasting facility: Design

```
template<typename BaseType>
struct downcast_base {
  using base_type = BaseType;
  friend auto downcast_guide(downcast_base);  // declaration only (no implementation)
};
```

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

```
template<typename Target, Downcastable T>
struct downcast_child : T {
  friend auto downcast_guide(typename downcast_child::downcast_base) { return Target(); }
};
```

Downcasting facility: Design

```
template<Downcastable T>
using downcast = decltype(detail::downcast_target_impl<T>());
```

Downcasting facility: Design

```
namespace detail {









  template<typename T>
  constexpr auto downcast_target_impl()
  {
    if constexpr(has_downcast<T>)
      return decltype(downcast_guide(std::declval<downcast_base<T>>()))();
    else
      return T();
  }

}

template<Downcastable T>
using downcast = decltype(detail::downcast_target_impl<T>());
```

Downcasting facility: Design

```
namespace detail {

  template<typename T>
  concept has_downcast = requires {
    downcast_guide(std::declval<downcast_base<T>>());
  };

  template<typename T>
  constexpr auto downcast_target_impl()
  {
    if constexpr(has_downcast<T>)
      return decltype(downcast_guide(std::declval<downcast_base<T>>()))();
    else
      return T();
  }

}

template<Downcastable T>
using downcast = decltype(detail::downcast_target_impl<T>());
```

Downcasting facility: Usage

```
template<Dimension D, Ratio R>
  requires (R::num * R::den > 0)
struct unit : downcast_base<unit<D, R>> {
  using dimension = D;
  using ratio = R;
};
```

Downcasting facility: Usage

```
template<Dimension D, Ratio R>
  requires (R::num * R::den > 0)
struct unit : downcast_base<unit<D, R>> {
  using dimension = D;
  using ratio = R;
};
```

```
template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT>
struct named_coherent_derived_unit : downcast_child<Child, unit<Dim, ratio<1>>> { /* ... */ };
```

Downcasting facility: Usage

```
template<Dimension D, Ratio R>
  requires (R::num * R::den > 0)
struct unit : downcast_base<unit<D, R>> {
  using dimension = D;
  using ratio = R;
};
```

```
template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT>
struct named_coherent_derived_unit : downcast_child<Child, unit<Dim, ratio<1>>> { /* ... */ };
```

```
struct metre : named_coherent_derived_unit<metre, length, "m", si_prefix> {};
```

Downcasting facility: Usage

```
template<Dimension D, Ratio R>
  requires (R::num * R::den > 0)
struct unit : downcast_base<unit<D, R>> {
  using dimension = D;
  using ratio = R;
};
```

```
template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT>
struct named_coherent_derived_unit : downcast_child<Child, unit<Dim, ratio<1>>> { /* ... */ };
```

```
struct metre : named_coherent_derived_unit<metre, length, "m", si_prefix> {};
```

```
// operator*(const quantity<U1, Rep1>& lhs, const quantity<U2, Rep2>& rhs)
using ret = quantity<downcast<unit<dim, ratio_multiply<typename U1::ratio,
                                                       typename U2::ratio>>>,
                     common_rep>;
```

Downcasting facility: User's experience: mp-units

ALIASES

```
Breakpoint 1, avg_speed<units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1> >,
  units::ratio<1000, 1, 0> >, double>, units::quantity<units::unit<units::dimension<units::exp<units::base_dim_time, 1> >,
  units::ratio<3600, 1, 0> >, double> >
(d=..., t=...) at velocity.cpp:31
31      return d / t;
```

Downcasting facility: User's experience: mp-units

ALIASES

```
Breakpoint 1, avg_speed<units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1> >,
  units::ratio<1000, 1, 0> >, double>, units::quantity<units::unit<units::dimension<units::exp<units::base_dim_time, 1> >,
  units::ratio<3600, 1, 0> >, double> >
(d=..., t=...) at velocity.cpp:31
31      return d / t;
```

STRONG TYPES + DOWNCASTING

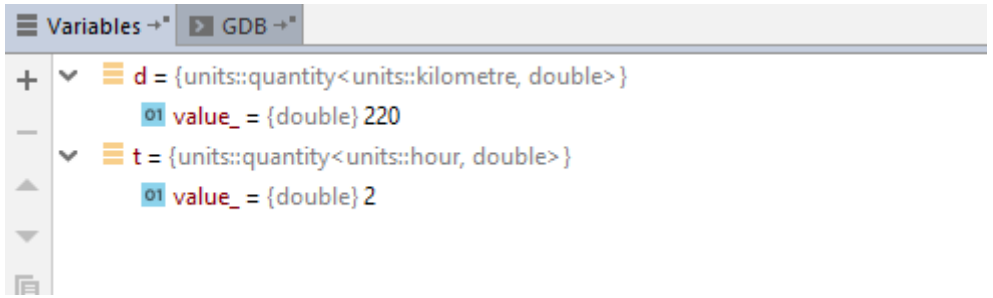
```
Breakpoint 1, avg_speed<units::quantity<units::kilometre, double>, units::quantity<units::hour, double> >
(d=..., t=...) at velocity.cpp:31
31      return d / t;
```

User experience: Debugging

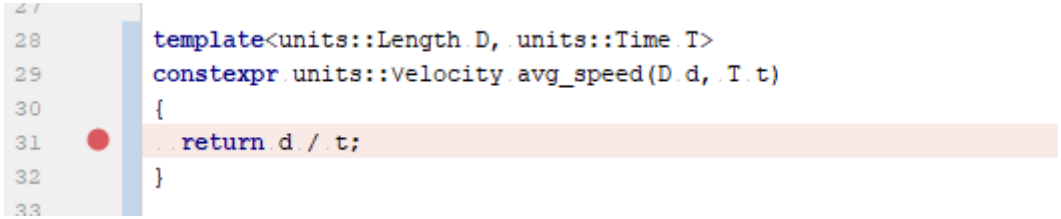
```
28  template<units::Length D, units::Time T>
29  constexpr units::Velocity avg_speed(D d, T t)
30  {
31      return d / t;
32  }
```

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

NTTP

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;
```

Non-type template parameters (NTTP) (C++20)

One of the following (optionally cv-qualified) types

- a **structural type**
- a type that *contains a placeholder* (**auto**) type
- a placeholder for a *deduced class type* (CTAD)

Non-type template parameters (NTTP) (C++20)

One of the following (optionally cv-qualified) types

- a **structural type**
- a type that *contains a placeholder* (**auto**) type
- a placeholder for a *deduced class type* (CTAD)

Structural types

- a *scalar* type with a constant destruction
- an *lvalue reference* type
- a *literal class* where all base classes and non-static data members are public and non-mutable structural types

Non-type template parameters (NTTP) (C++20)

Two template *instantiations refer to the same* class, function, or variable if their corresponding NTTPs are **template argument equivalent**

NTTP: Template argument equivalent values (C++20)

- Integral types with the same values

NTTP: Template argument equivalent values (C++20)

- **Integral** types with the same values
- **Enumeration** type with the same values

NTTP: Template argument equivalent values (C++20)

- Integral types with the same values
- Enumeration type with the same values
- `std::nullptr_t`

NTTP: Template argument equivalent values (C++20)

- **Integral** types with the same values
- **Enumeration** type with the same values
- **`std::nullptr_t`**
- **Floating-point** type with identical values

NTTP: Template argument equivalent values (C++20)

- **Integral** types with the same values
- **Enumeration** type with the same values
- **`std::nullptr_t`**
- **Floating-point** type with identical values
- **Pointer type** with the same pointer value
- **Pointer-to-member** type referring to the same class member or both are **`nullptr`**
- **Reference** type referring to the same object or function

NTTP: Template argument equivalent values (C++20)

- **Integral** types with the same values
- **Enumeration** type with the same values
- **`std::nullptr_t`**
- **Floating-point** type with identical values
- **Pointer type** with the same pointer value
- **Pointer-to-member** type referring to the same class member or both are **`nullptr`**
- **Reference** type referring to the same object or function
- **Array** type with template argument equivalent elements

NTTP: Template argument equivalent values (C++20)

- **Integral** types with the same values
- **Enumeration** type with the same values
- **`std::nullptr_t`**
- **Floating-point** type with identical values
- **Pointer type** with the same pointer value
- **Pointer-to-member** type referring to the same class member or both are **`nullptr`**
- **Reference** type referring to the same object or function
- **Array** type with template argument equivalent elements
- **Class** type with *template argument equivalent* subobjects and reference members

NTTP: Template argument equivalent values (C++20)

- **Integral** types with the same values
- **Enumeration** type with the same values
- **`std::nullptr_t`**
- **Floating-point** type with identical values
- **Pointer type** with the same pointer value
- **Pointer-to-member** type referring to the same class member or both are **`nullptr`**
- **Reference** type referring to the same object or function
- **Array** type with template argument equivalent elements
- **Class** type with *template argument equivalent* subobjects and reference members
- **Union** type where either both have
  - *no active member*
  - the *same active member* and their active members are *template argument equivalent*

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

STD::RATIO

```
struct yard : derived_unit<yard, "yd", length, ratio<9’144, 10’000>> {};
struct foot : derived_unit<foot, "ft", length, ratio_multiply<ratio<1, 3>, yard::ratio>> {};
struct inch : derived_unit<inch, "in", length, ratio_multiply<ratio<1, 12>, foot::ratio>> {};
struct mile : derived_unit<mile, "mi", length, ratio_multiply<ratio<1’760>, yard::ratio>> {};
```

NTTP RATIO

```
struct yard : derived_unit<yard, "yd", length, ratio(9’144, 10’000)> {};
struct foot : derived_unit<foot, "ft", length, yard::ratio / 3> {};
struct inch : derived_unit<inch, "in", length, foot::ratio / 12> {};
struct mile : derived_unit<mile, "mi", length, 1’760 * yard::ratio> {};
```

NTTP in action

STD::RATIO

```
struct yard : derived_unit<yard, "yd", length, ratio<9’144, 10’000>> {};
struct foot : derived_unit<foot, "ft", length, ratio_multiply<ratio<1, 3>, yard::ratio>> {};
struct inch : derived_unit<inch, "in", length, ratio_multiply<ratio<1, 12>, foot::ratio>> {};
struct mile : derived_unit<mile, "mi", length, ratio_multiply<ratio<1’760>, yard::ratio>> {};
```

NTTP RATIO

```
struct yard : derived_unit<yard, "yd", length, ratio(9’144, 10’000)> {};
struct foot : derived_unit<foot, "ft", length, yard::ratio / 3> {};
struct inch : derived_unit<inch, "in", length, foot::ratio / 12> {};
struct mile : derived_unit<mile, "mi", length, 1’760 * yard::ratio> {};
```

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<downcast<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<downcast<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<downcast<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<downcast<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<downcast<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);
```

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

NEW TOYS IN A TOOLBOX

CONCEPTS

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) { /* ... */ }
```

```
auto foo(auto&& t) { /* ... */ }
```

How do you feel about such an interface?

```
void* foo(void* t) { /* ... */ }
```

```
auto foo(auto&& t) { /* ... */ }
```

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

How do you feel about such an interface?

```
void* foo(void* t) { /* ... */ }
```

```
auto foo(auto&& t) { /* ... */ }
```

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

```
template<typename T> class foo { /* ... */ };
```

How do you feel about such an interface?

```
void* foo(void* t) { /* ... */ }
```

```
auto foo(auto&& t) { /* ... */ }
```

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

```
template<typename T> class foo { /* ... */ };
```

Unconstrained template parameters are the **`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*

Concepts perception issue #1

Although Concepts are **constraints on types**, you don't find them by looking at the types in your system. **You find them by studying the algorithms.**

-- Eric Niebler

Concepts perception issue #1

> Although Concepts are **constraints on types**, you don't find them by looking at the types in your system. **You find them by studying the algorithms.**
>
> *-- Eric Niebler*

- Above is often oversimplified as *"All concepts should arise from algorithms"*

Concepts perception issue #1

Although Concepts are **constraints on types**, you don't find them by looking at the types in your system. **You find them by studying the algorithms.**

-- Eric Niebler

- Above is often oversimplified as *"All concepts should arise from algorithms"*

Because this is what we explored so far...

Sicily's Medieval Map of the World

Sicily's Medieval Map of the World (Upside down)

Concept categories

Concept categories

PREDICATES

```
template<class Derived, class Base>
concept derived_from =
  is_base_of_v<Base, Derived> &&
  is_convertible_v<const volatile Derived*, const volatile Base*>;
```

- Names ending with prepositions

Concept categories

CAPABILITIES

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

- Single requirement concepts
- Named with adjectives **`-ible`** or **`-able`**

Concept categories

ABSTRACTIONS

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

- High-level concepts
- Named using very generic nouns

The world is different and bigger than we initially imagined

What about our `std::ratio_multiply`?

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

What about our `std::ratio_multiply`?

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

```
using type = std::ratio_multiply<std::string, std::milli>;
```

What about our `std::ratio_multiply`?

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

```
using type = std::ratio_multiply<std::string, std::milli>;
```

```
include/c++/9.2.0/ratio: In instantiation of 'struct std::__ratio_multiply<std::__cxx11::basic_string<char>,
std::ratio<1, 1000> >':
include/c++/9.2.0/ratio:311:11:   required by substitution of 'template<class _R1, class _R2> using ratio_multiply =
typename std::__ratio_multiply::type [with _R1 = std::__cxx11::basic_string<char>; _R2 = std::ratio<1, 1000>]'
<source>:4:57:   required from here
include/c++/9.2.0/ratio:294:35: error: 'num' is not a member of 'std::__cxx11::basic_string<char>'
  294 |         __safe_multiply<(_R1::num / __gcd1),
      |                          ~~~~~~~~~~^~~~~~~~
include/c++/9.2.0/ratio: In instantiation of 'const intmax_t std::__ratio_multiply<std::__cxx11::basic_string<char>,
std::ratio<1, 1000> >::__gcd1':
include/c++/9.2.0/ratio:294:35:   required from 'struct std::__ratio_multiply<std::__cxx11::basic_string<char>,
std::ratio<1, 1000> >'
include/c++/9.2.0/ratio:311:11:   required by substitution of 'template<class _R1, class _R2> using ratio_multiply =
typename std::__ratio_multiply::type [with _R1 = std::__cxx11::basic_string<char>; _R2 = std::ratio<1, 1000>]'
<source>:4:57:   required from here
include/c++/9.2.0/ratio:287:29: error: 'num' is not a member of
'std::__cxx11::basic_string<char>'
  287 |       static const intmax_t __gcd1 =
      |                              ^~~~~~
```

New concept category?

FAMILY OF INSTANTIATIONS

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

New concept category?

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>;
```

New concept category?

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>;
```

- Named with an upper-case first letter of the class template to match

New concept category?

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>;
```

- Named with an upper-case first letter of the class template to match

Well, that naming is not going to fly in the Committee ;-)

Concepts perception issue #2

> The experience of the authors and implementors of the Ranges TS is that **getting concept definitions and algorithm constraints right is hard**.
>
> *-- P0896 (The One Ranges Proposal)*

Concepts perception issue #2

> The experience of the authors and implementors of the Ranges TS is that **getting concept definitions and algorithm constraints right is hard**.
>
> *-- P0896 (The One Ranges Proposal)*

- Above is often oversimplified as *"Defining concepts is hard, let's keep their number small"*

Concepts perception issue #2

> The experience of the authors and implementors of the Ranges TS is that **getting concept definitions and algorithm constraints right is hard**.
>
> *-- P0896 (The One Ranges Proposal)*

- Above is often oversimplified as *"Defining concepts is hard, let's keep their number small"*

Again, because this is what we explored so far...

Not all concepts have to be hard to define and prove correct

ANY INSTANTIATION OF A CLASS TEMPLATE

```
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;
```

Not all concepts have to be hard to define and prove correct

ANY INSTANTIATION OF A CLASS TEMPLATE

```
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>;
```

Not all concepts have to be hard to define and prove correct

ANY INSTANTIATION OF A CLASS TEMPLATE

```
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>;
```

Is this concept easy to prove correct?

What about our `std::ratio_multiply`?

```
template<Ratio R1, Ratio R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

What about our `std::ratio_multiply`?

```
template<Ratio R1, Ratio R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

```
using type = ratio_multiply<std::string, std::milli>;
```

What about our `std::ratio_multiply`?

```
template<Ratio R1, Ratio R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

```
using type = ratio_multiply<std::string, std::milli>;
```

```
<source>:24:52: error: template constraint failure for 'template<class R1, class R2>  requires (Ratio<R1>) && (Ratio<R2>)
  using ratio_multiply = std::ratio_multiply<R1, R2>'
   24 | using type = ratio_multiply<std::string, std::milli>;
      |                                                    ^
<source>:24:52: note: constraints not satisfied
<source>:11:9:   required for the satisfaction of 'Ratio<std::__cxx11::basic_string<char, std::char_traits<char>,
  std::allocator<char> > >'
<source>:11:17: note: the expression 'is_ratio<T>' evaluated to 'false'
   11 | concept Ratio = is_ratio<T>;
      |                 ^~~~~~~~~~
```

Not all concepts have to be hard to define and prove correct

RATIO-LIKE TYPE

```
template<typename T>
concept Ratio = requires {
    T::num;
    T::den;
};
```

Not all concepts have to be hard to define and prove correct

RATIO-LIKE TYPE

```
template<typename T>
concept Ratio = requires {
    T::num;
    T::den;
};
```

Is this concept easy to prove correct?

What about our `std::ratio_multiply`?

```
template<Ratio R1, Ratio R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

```
using type = ratio_multiply<std::string, std::milli>;
```

What about our `std::ratio_multiply`?

```
template<Ratio R1, Ratio R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

```
using type = ratio_multiply<std::string, std::milli>;
```

```
<source>:24:52: error: template constraint failure for 'template<class R1, class R2>  requires (Ratio<R1>) &&
  (Ratio<R2>) using ratio_multiply = std::ratio_multiply<R1, R2>'
   24 | using type = ratio_multiply<std::string, std::milli>;
      |                                                    ^
<source>:24:52: note: constraints not satisfied
<source>:14:9:   required for the satisfaction of 'Ratio<std::__cxx11::basic_string<char, std::char_traits<char>,
  std::allocator<char> > >'
<source>:14:17:   in requirements
<source>:15:8: note: the required expression 'T::num' is invalid
   15 |     T::num;
      |        ^~~
<source>:16:8: note: the required expression 'T::den' is invalid
   16 |     T::den;
      |        ^~~
```

Sometimes you do not want a named concept

ANY INSTANTIATION OF A CLASS TEMPLATE

```
template<typename T, template<typename> typename Trait>
concept Satisfies = Trait<T>::value;
```

Sometimes you do not want a named concept

ANY INSTANTIATION OF A CLASS TEMPLATE

```
template<typename T, template<typename> typename Trait>
concept Satisfies = Trait<T>::value;
```

Is this concept easy to prove correct?

What about our `std::ratio_multiply`?

```
template<typename T>
struct is_ratio : std::false_type {};

template<intmax_t Num, intmax_t Den>
struct is_ratio<std::ratio<Num, Den>> : std::true_type {};
```

What about our `std::ratio_multiply`?

```
template<typename T>
struct is_ratio : std::false_type {};

template<intmax_t Num, intmax_t Den>
struct is_ratio<std::ratio<Num, Den>> : std::true_type {};
```

```
template<Satisfies<is_ratio> R1, Satisfies<is_ratio> R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

```
using type = ratio_multiply<std::string, std::milli>;
```

What about our `std::ratio_multiply`?

```
template<typename T>
struct is_ratio : std::false_type {};

template<intmax_t Num, intmax_t Den>
struct is_ratio<std::ratio<Num, Den>> : std::true_type {};
```

```
template<Satisfies<is_ratio> R1, Satisfies<is_ratio> R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

```
using type = ratio_multiply<std::string, std::milli>;
```

```
<source>:31:52: error: template constraint failure for 'template<class R1, class R2>  requires (Satisfies<R1, is_ratio>)
  && (Satisfies<R2, is_ratio>) using ratio_multiply = std::ratio_multiply<R1, R2>'
   31 | using type = ratio_multiply<std::string, std::milli>;
      |                                                    ^
<source>:31:52: note: constraints not satisfied
<source>:20:9:   required for the satisfaction of 'Satisfies<std::__cxx11::basic_string<char, std::char_traits<char>,
  std::allocator<char> >, is_ratio>'
<source>:20:9: note: the expression 'Trait<T>::value' evaluated to 'false'
   20 | concept Satisfies = Trait<T>::value;
      |         ^~~~~~~~~
```

Concepts example

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

```
constexpr price calc_fine(units::Velocity auto speed);
```

Concepts 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>;
```

```
constexpr price calc_fine(units::Velocity auto speed);
```

Concepts example

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

```
constexpr price calc_fine(units::Velocity auto speed);
```

Concepts example

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

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

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

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

- Constraining *function template return types*

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

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

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

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

```
const units::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>
  requires (R::num * R::den > 0)
struct unit;
```

And you know what? It is round ;-)

Template Parameter Kinds

Templates are parameterized by *one* or *more* template parameters

Template Parameter Kinds

Templates are parameterized by *one* or *more* template parameters

- **type** template parameters

```
template<typename Ptr> class smart_ptr { /* ... */ };
smart_ptr<int> ptr;
```

Template Parameter Kinds

Templates are parameterized by *one* or *more* template parameters

- **type** template parameters

```
template<typename Ptr> class smart_ptr { /* ... */ };
smart_ptr<int> ptr;
```

- **non-type** template parameters

```
template<typename T, size_t N> class array { /* ... */ };
array<int, 5> a;
```

Template Parameter Kinds

Templates are parameterized by *one* or *more* template parameters

- **type** template parameters

```
template<typename Ptr> class smart_ptr { /* ... */ };
smart_ptr<int> ptr;
```

- **non-type** template parameters

```
template<typename T, size_t N> class array { /* ... */ };
array<int, 5> a;
```

- **template** template parameters

```
template<typename T> class my_deleter {};
template<typename T, template<typename> typename Policy> class handle { /* ... */ };
handle<FILE, my_deleter> h;
```

Imagine a Universal Template Parameter Kind (P1985)

A template parameter **placeholder that can be replaced with any kind of a template parameter** (type, non-type, template)

Imagine a Universal Template Parameter Kind (P1985)

A template parameter **placeholder that can be replaced with any kind of a template parameter** (type, non-type, template)

EXAMPLE

```
template<...>
class Foo;
```

- Declares **Foo** as a class template with **any number and any kind** of template parameters

Concept to check for an instantiation of a class template

```
template<typename T, template<...> typename U>
inline constexpr bool instance_of = false;

template<... Params, template<...> typename U>
inline constexpr bool instance_of<U<Params...>, U> = true;
```

Concept to check for an instantiation of a class template

```
template<typename T, template<...> typename U>
inline constexpr bool instance_of = false;

template<... Params, template<...> typename U>
inline constexpr bool instance_of<U<Params...>, U> = true;
```

```
template<typename T, template<...> typename U>
concept Is = instance_of<T, U>;
```

What about our `std::ratio_multiply`?

```
template<Is<std::ratio> R1, Is<std::ratio> R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

```
using type = ratio_multiply<std::string, std::milli>;
```

What about our `std::ratio_multiply`?

```
template<Is<std::ratio> R1, Is<std::ratio> R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

```
using type = ratio_multiply<std::string, std::milli>;
```

```
<source>:27:52: error: template constraint failure for 'template<class R1, class R2>  requires (Is<R1, std::ratio>) &&
   (Is<R2, std::ratio>) using ratio_multiply =  std::ratio_multiply<R1, R2>'
   27 | using type = ratio_multiply<std::string, std::milli>;
      |                                                    ^
<source>:27:52: note: constraints not satisfied
<source>:21:9:   required for the satisfaction of 'Is<std::__cxx11::basic_string<char, std::char_traits<char>,
   std::allocator<char> >, std::ratio>'
<source>:21:14: note: the expression 'instance_of<T, U>' evaluated to 'false'
   21 | concept Is = instance_of<T, U>;
      |              ^~~~~~~~~~~~~
```

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 = std::ratio<<%= 2 * i - 1 %>, <%= 2 * n %>>;
  using r2 = std::ratio<<%= 2 * i %>, <%= 2 * n %>>;
#else
  using r1 = void;
  using r2 = void;
#endif
};
<% end %>

int main() {}
```

A short intro to Metabench: Generated C++ code

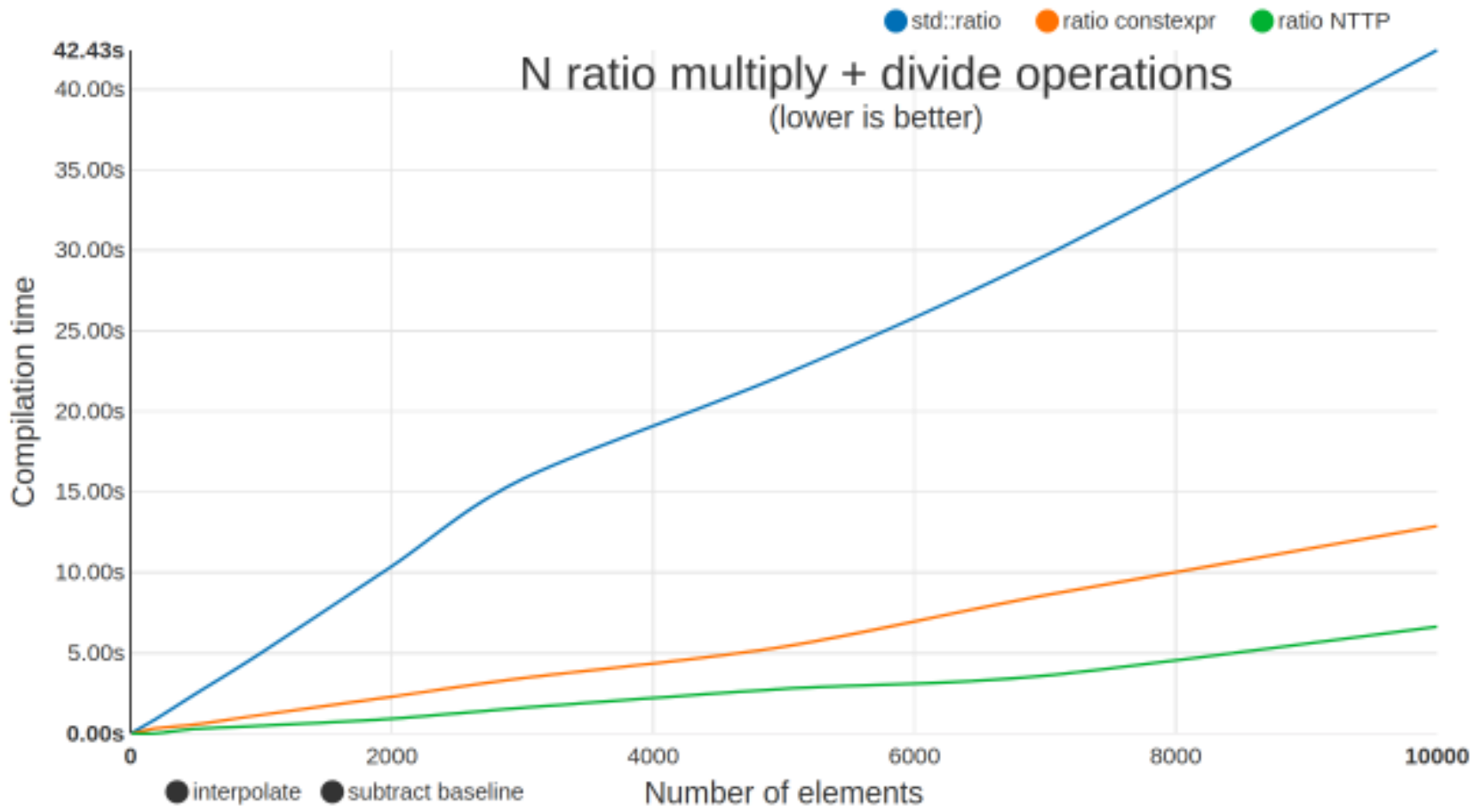
```cpp
#include "ratio_type_constexpr.h"

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() {}
```

A short intro to Metabench: Generated C++ code

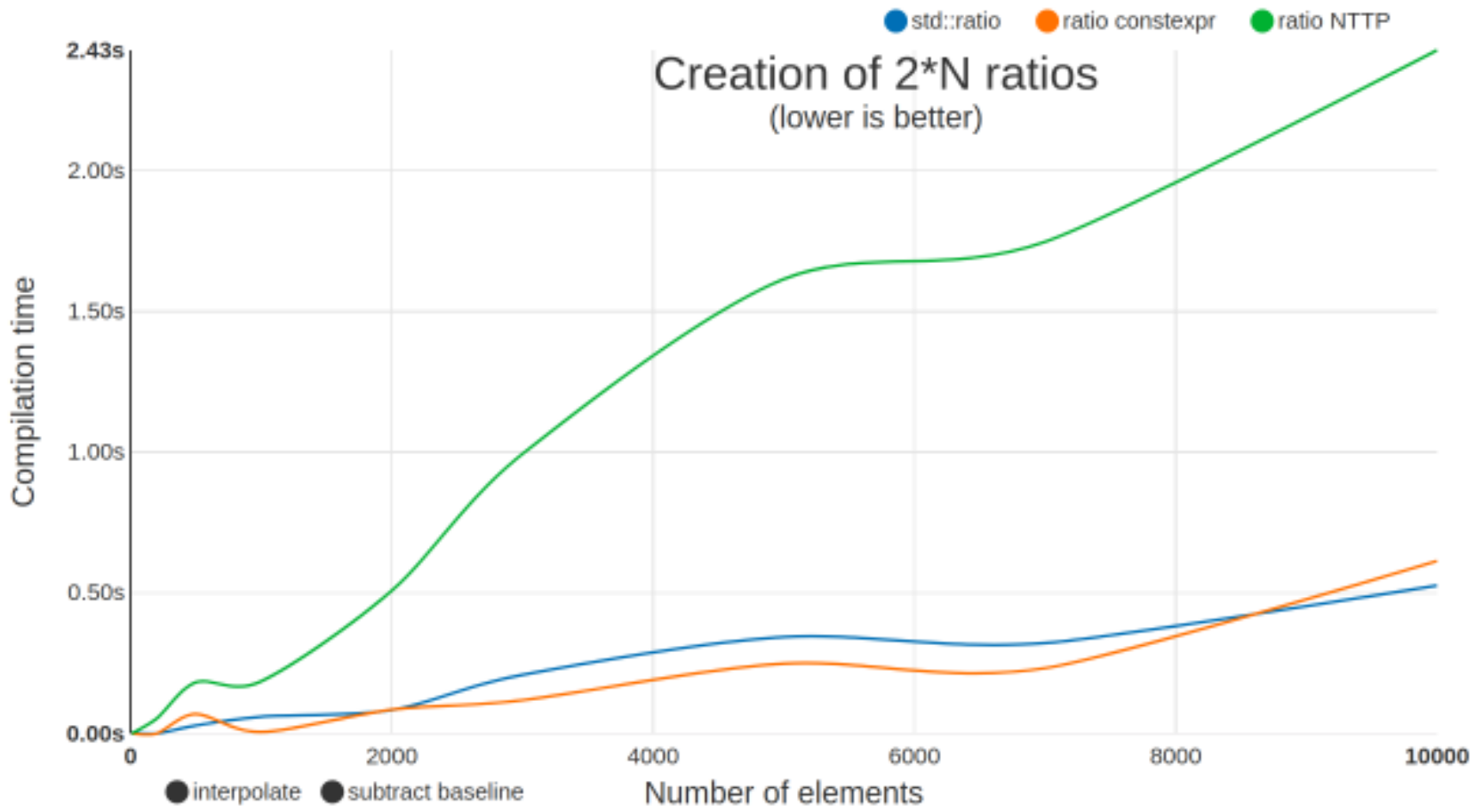
```cpp
#include "ratio_type_constexpr.h"

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() {}
```

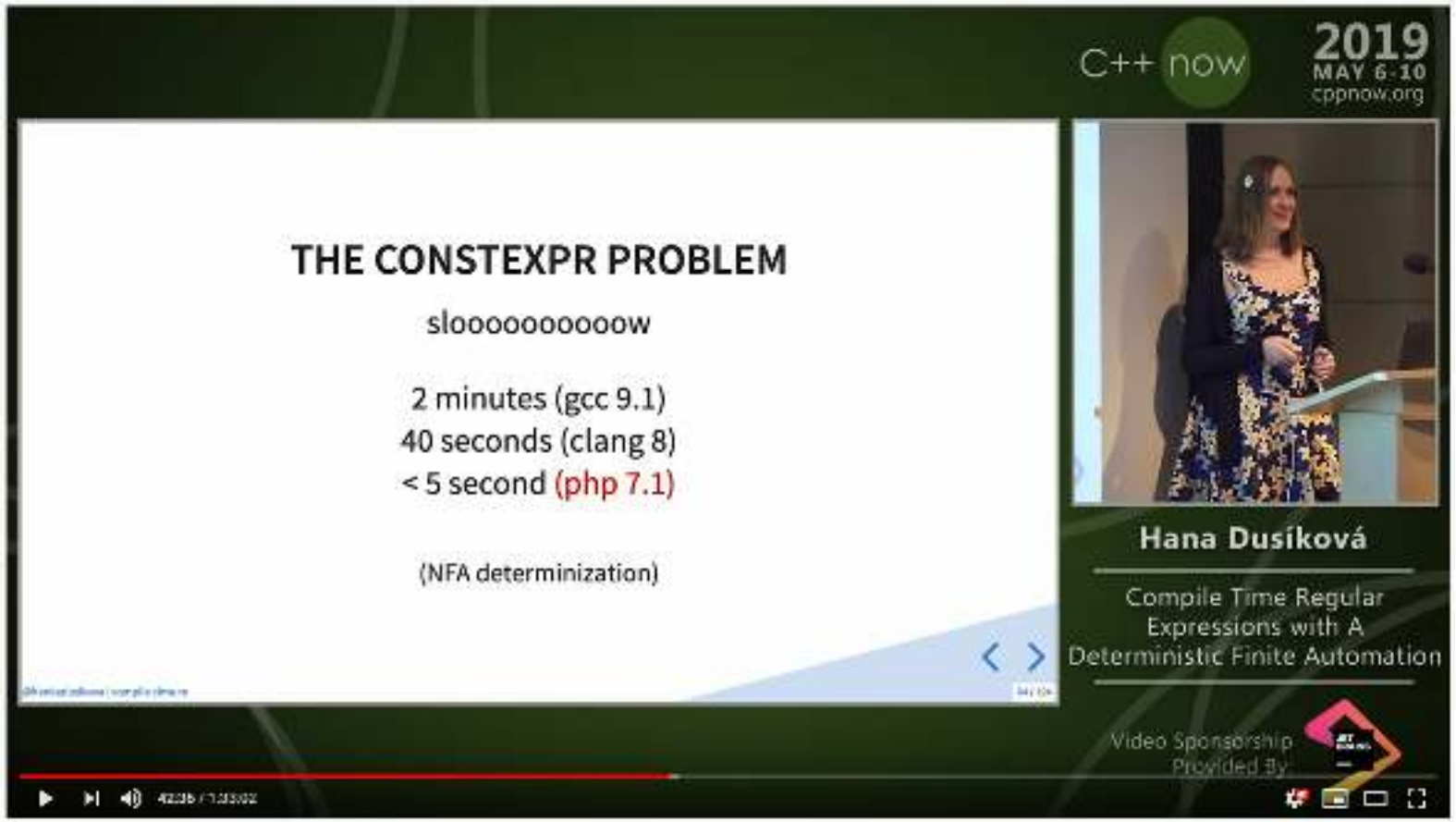
`ratio` performance



`ratio` performance



Current `constexpr` QoI is slow

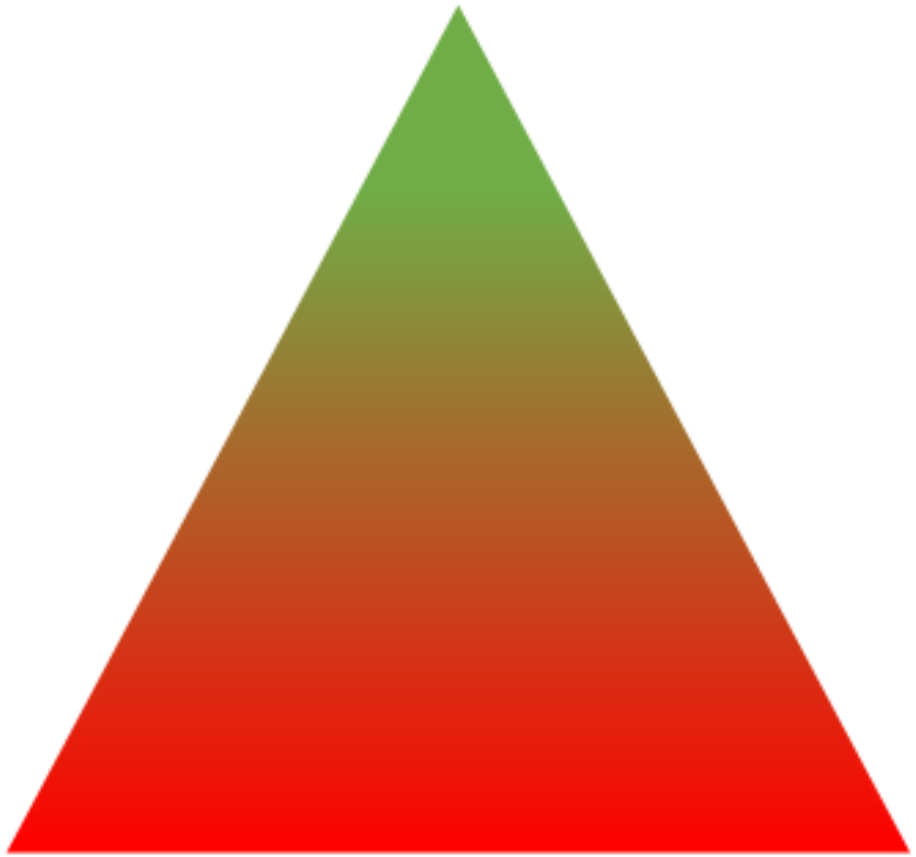


C++Now 2019: Hana Dusíková "Compile Time Regular Expressions with A Deterministic Finite Automator"

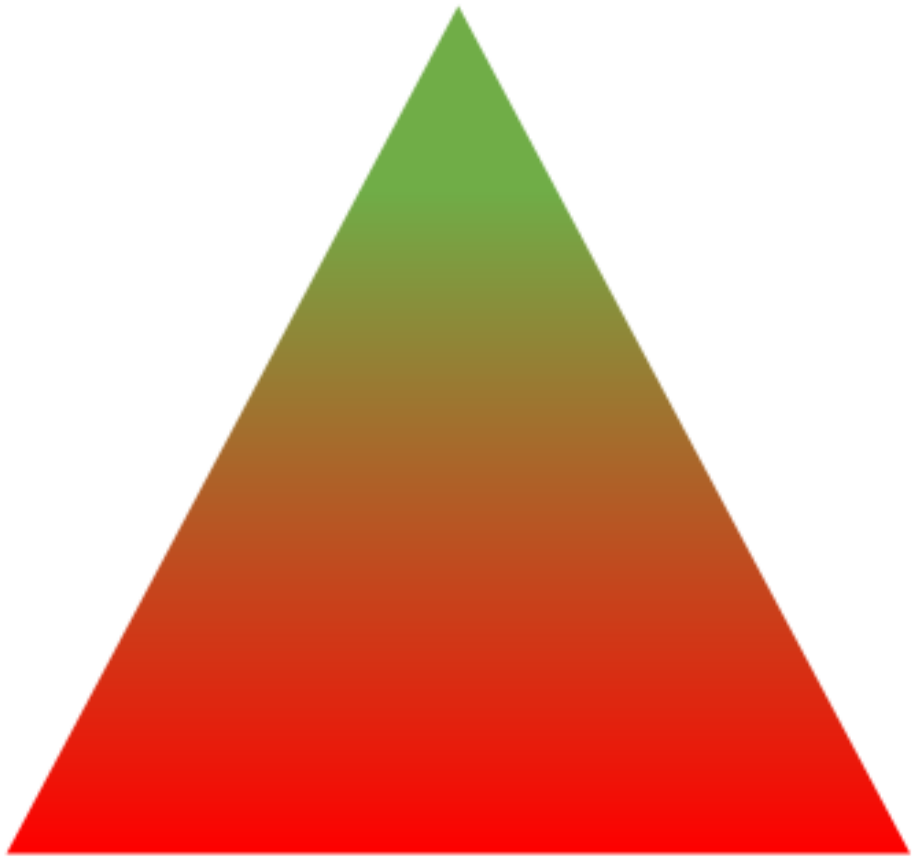
The Rule of Chiel

The Rule of Chiel

Cost of operations: The Rule of Chiel

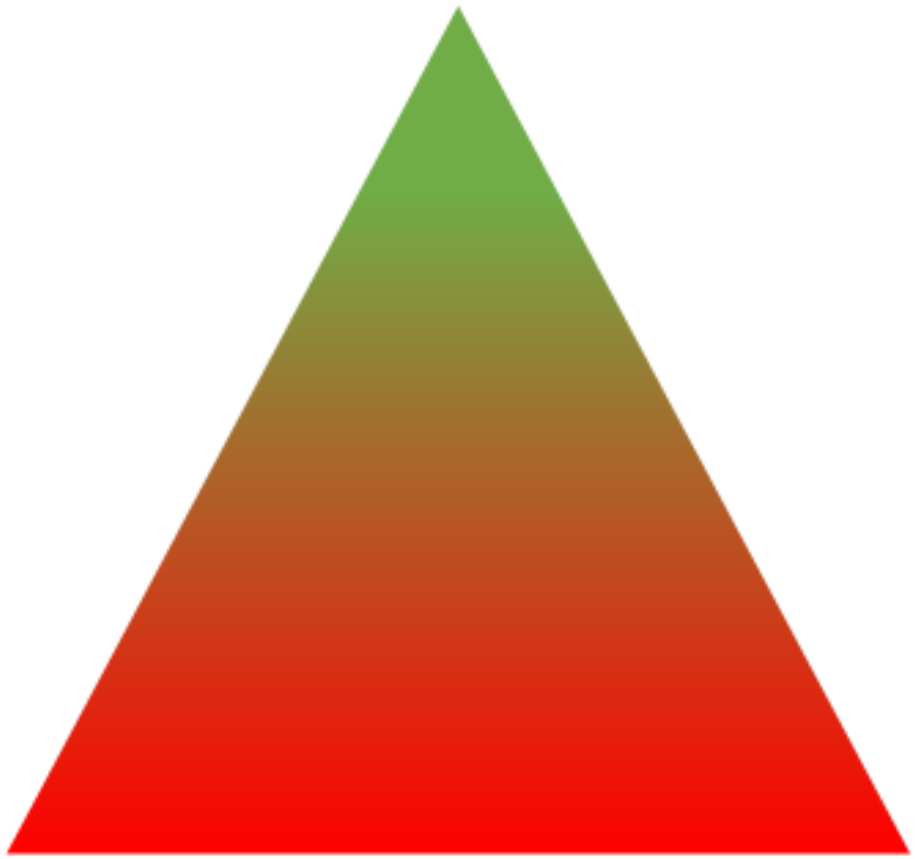


Cost of operations: The Rule of Chiel



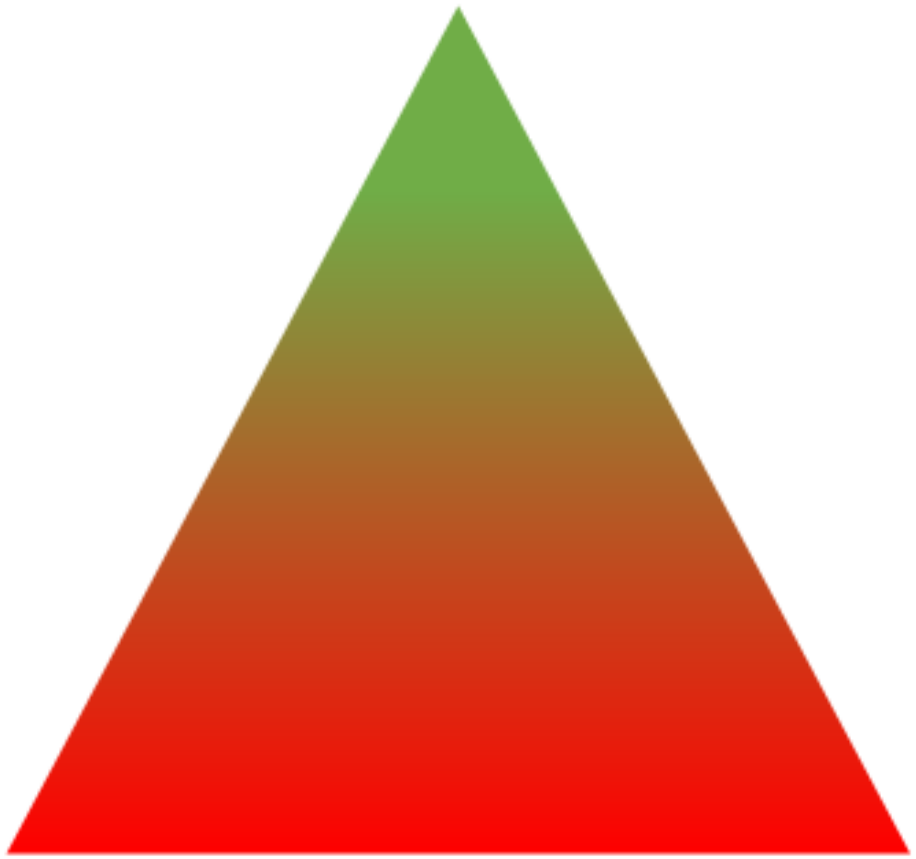
1. Looking up a memoized type

Cost of operations: The Rule of Chiel



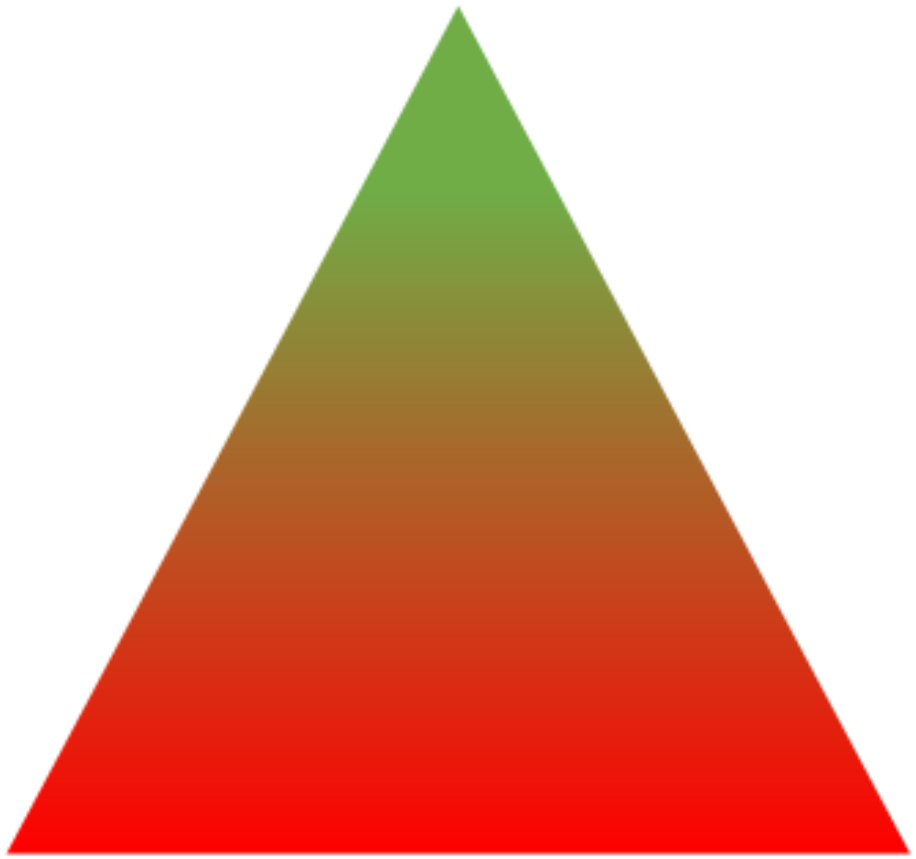
1. Looking up a memoized type
2. Adding a parameter to an alias call

Cost of operations: The Rule of Chiel



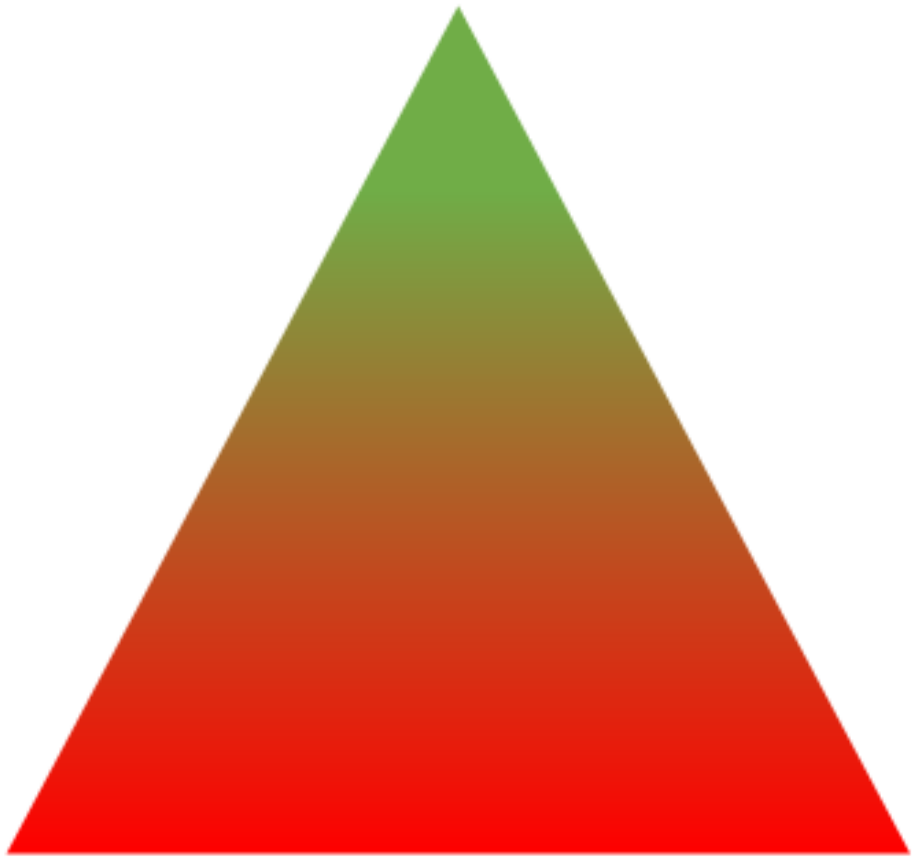
1. Looking up a memoized type
2. Adding a parameter to an alias call
3. Adding a parameter to a type

Cost of operations: The Rule of Chiel



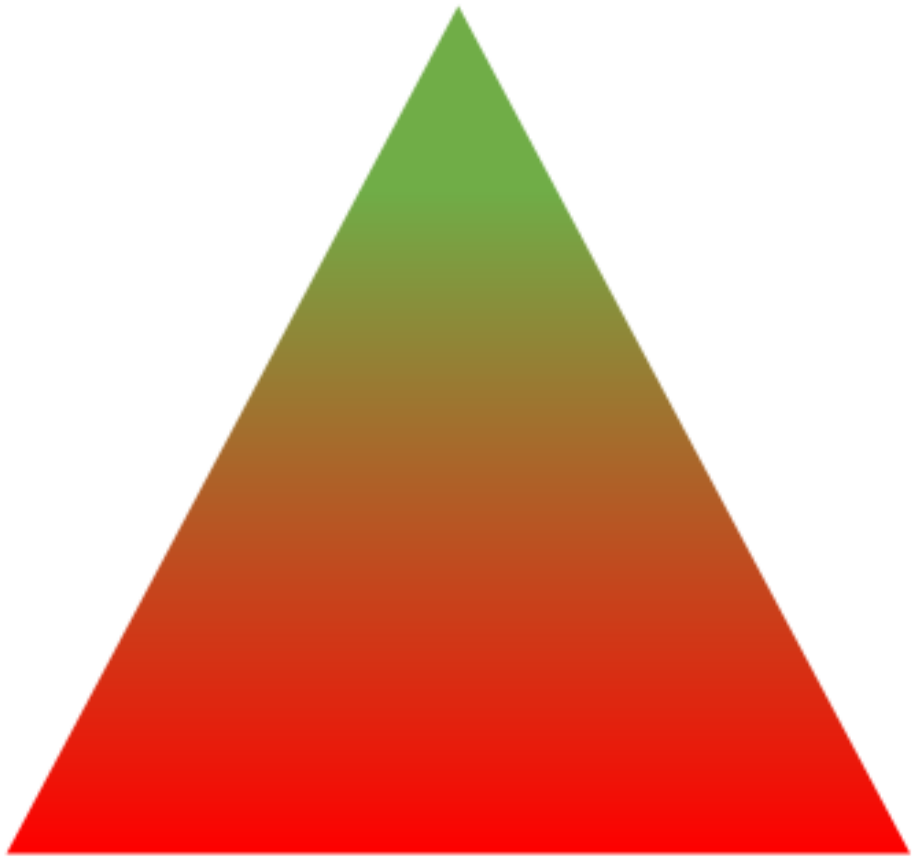
1. Looking up a memoized type
2. Adding a parameter to an alias call
3. Adding a parameter to a type
4. Calling an alias

Cost of operations: The Rule of Chiel



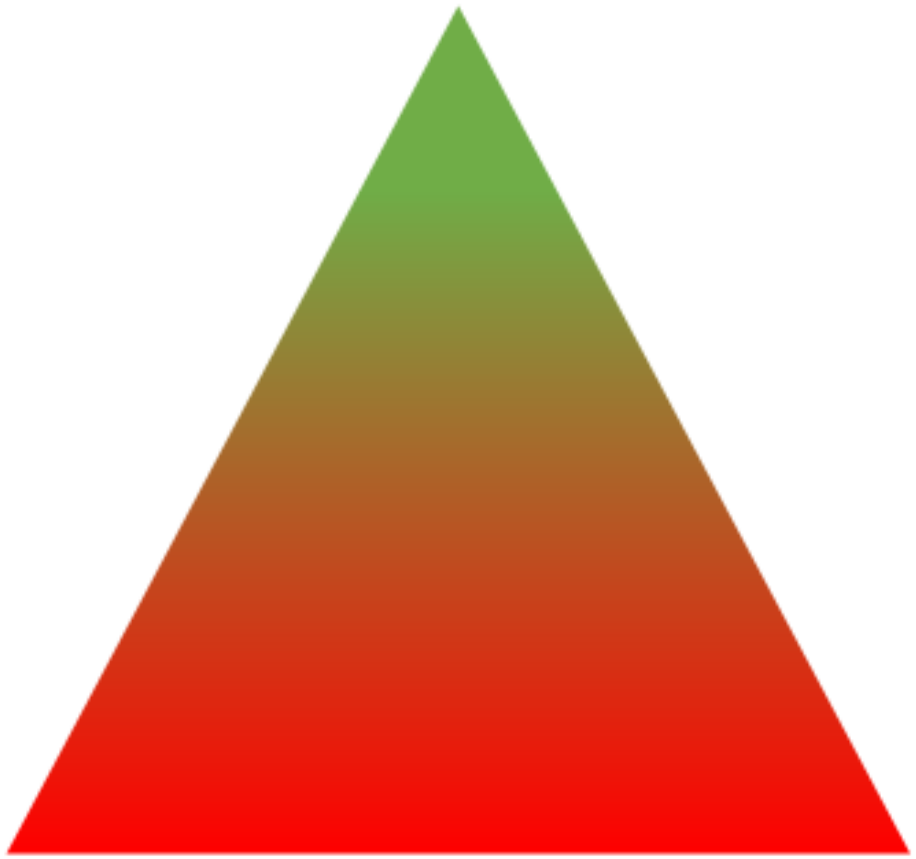
1. Looking up a memoized type
2. Adding a parameter to an alias call
3. Adding a parameter to a type
4. Calling an alias
5. Instantiating a class

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>

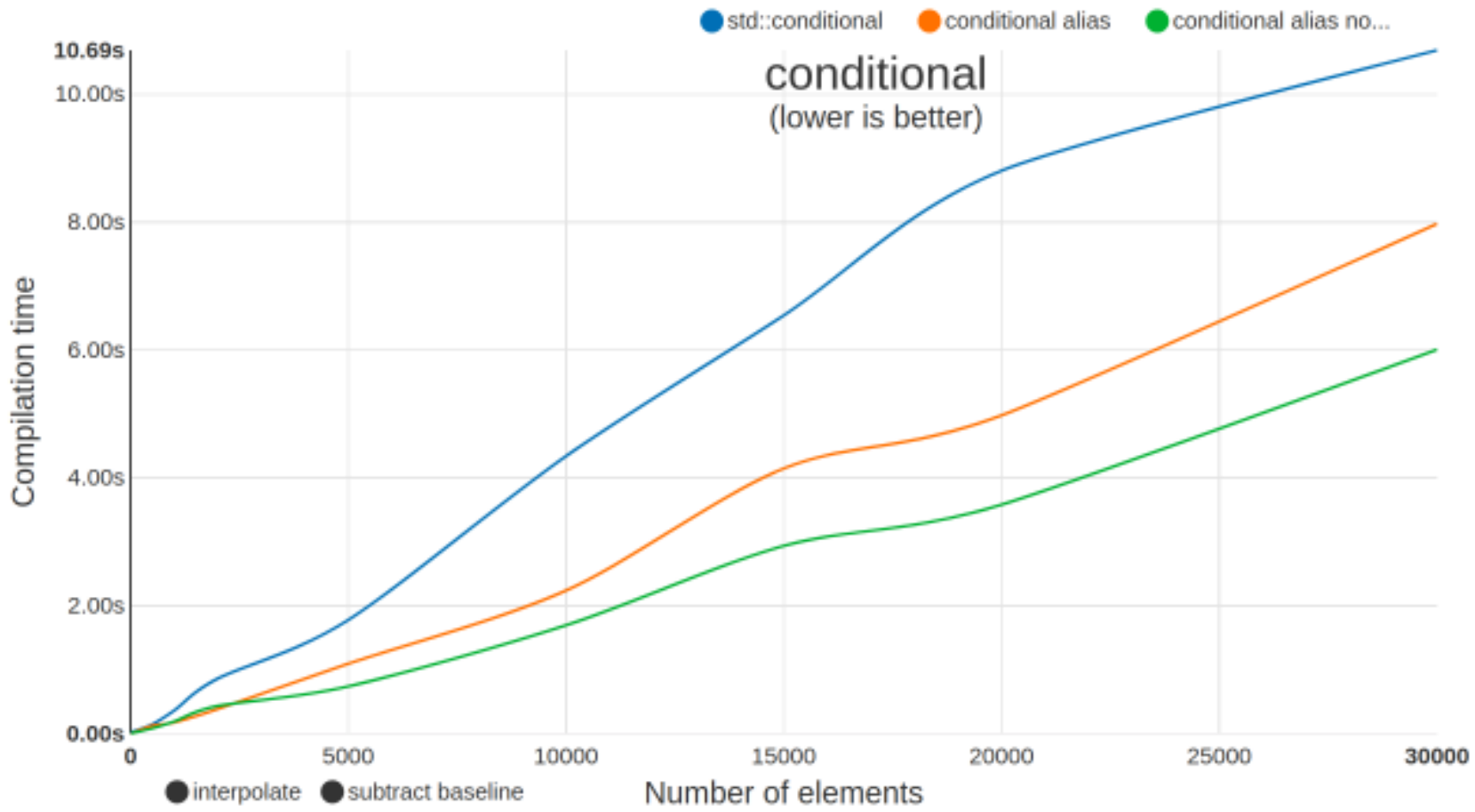
TRADITIONAL

```
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;
```

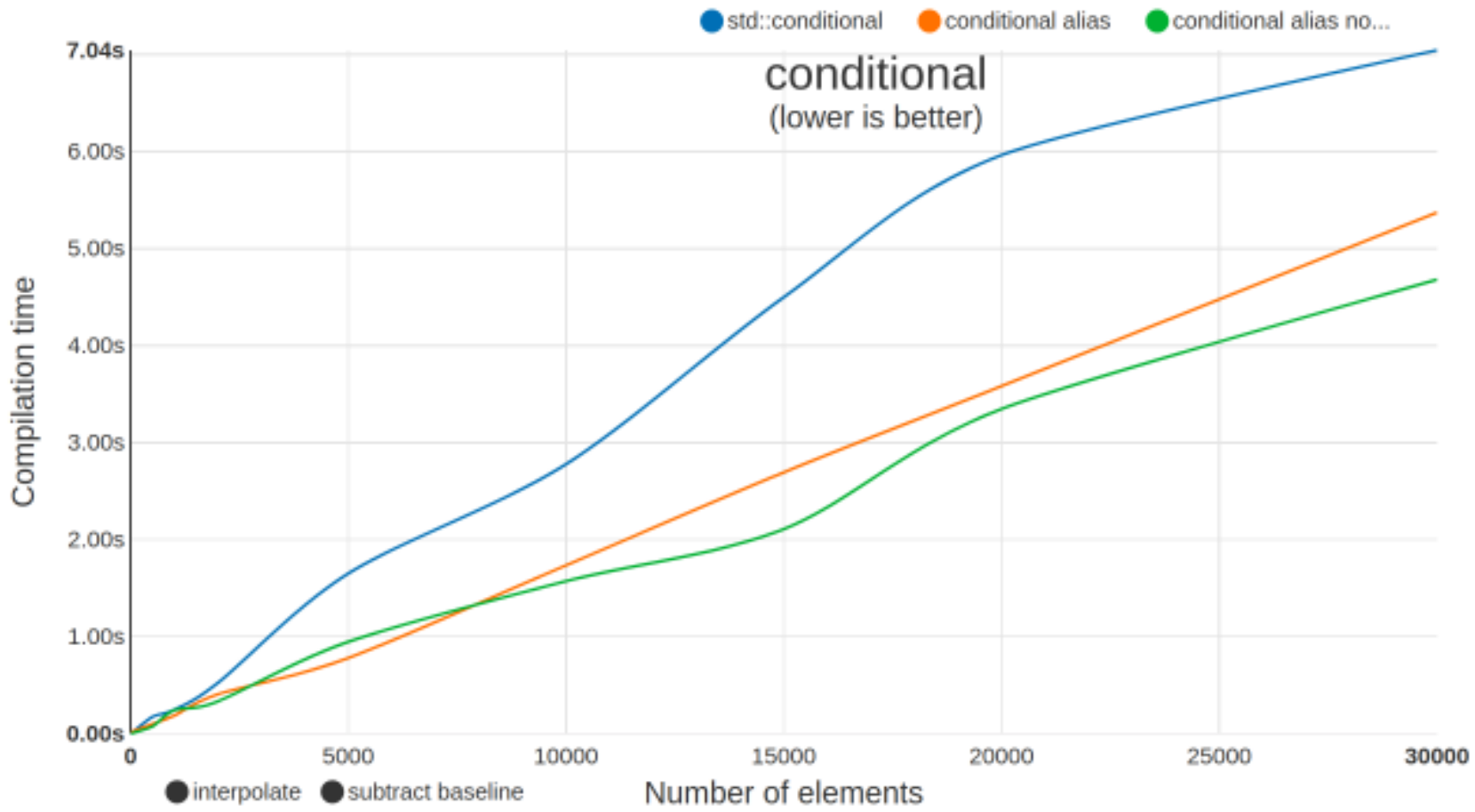
WITH ALIAS TEMPLATE

```
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

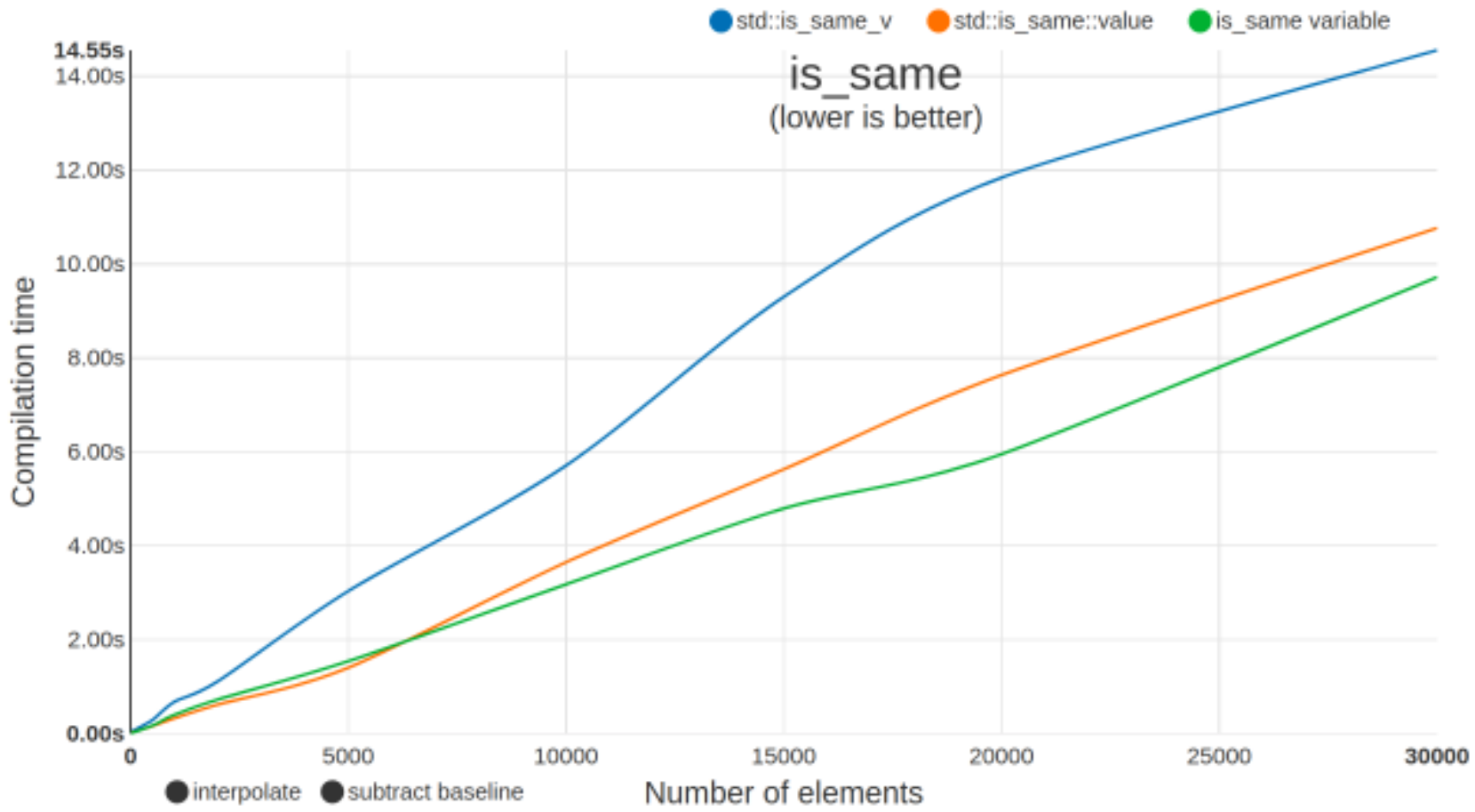
```
template<class T, class U>
struct is_same : std::false_type {};

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;
```

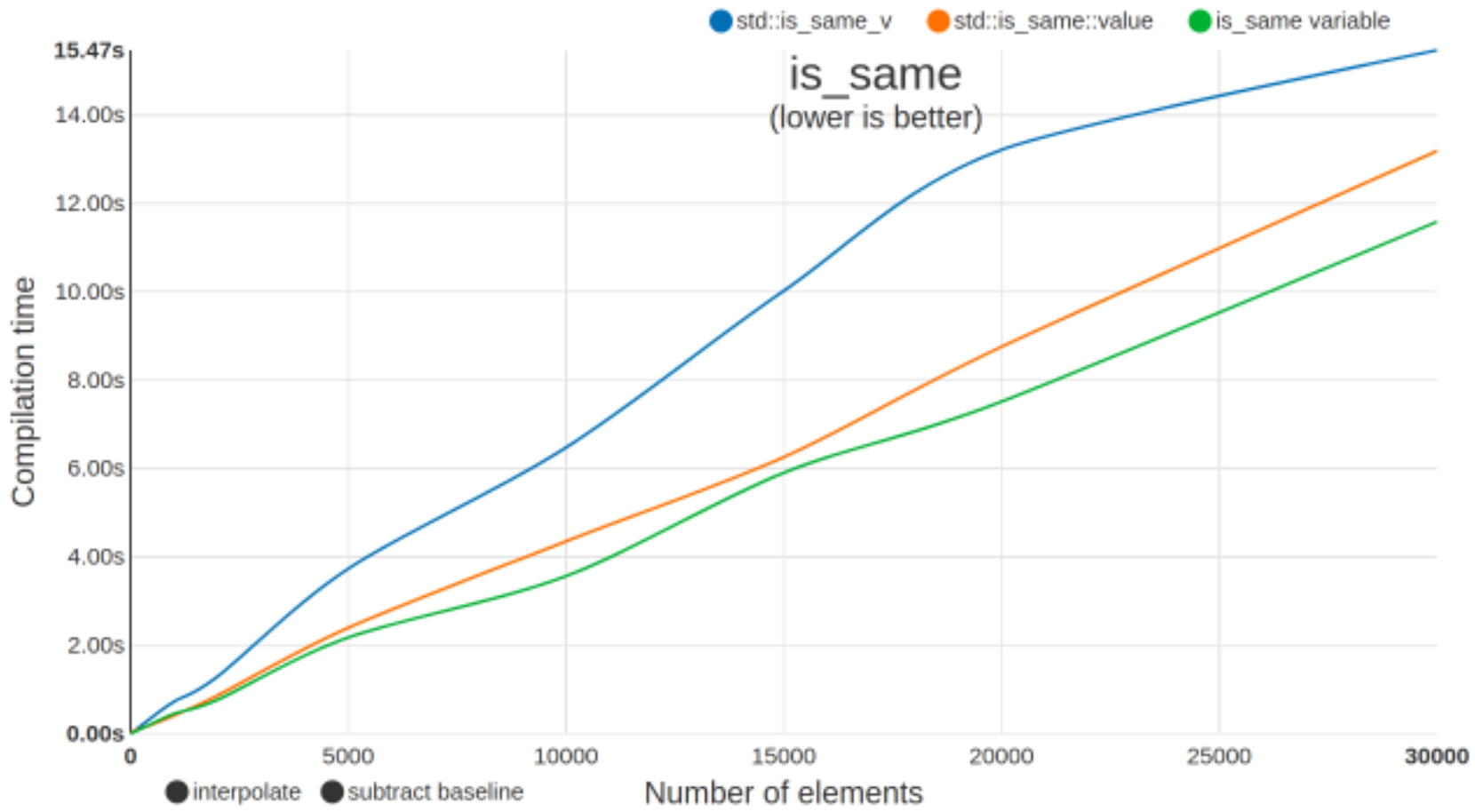
USING VARIABLE TEMPLATES

```
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?

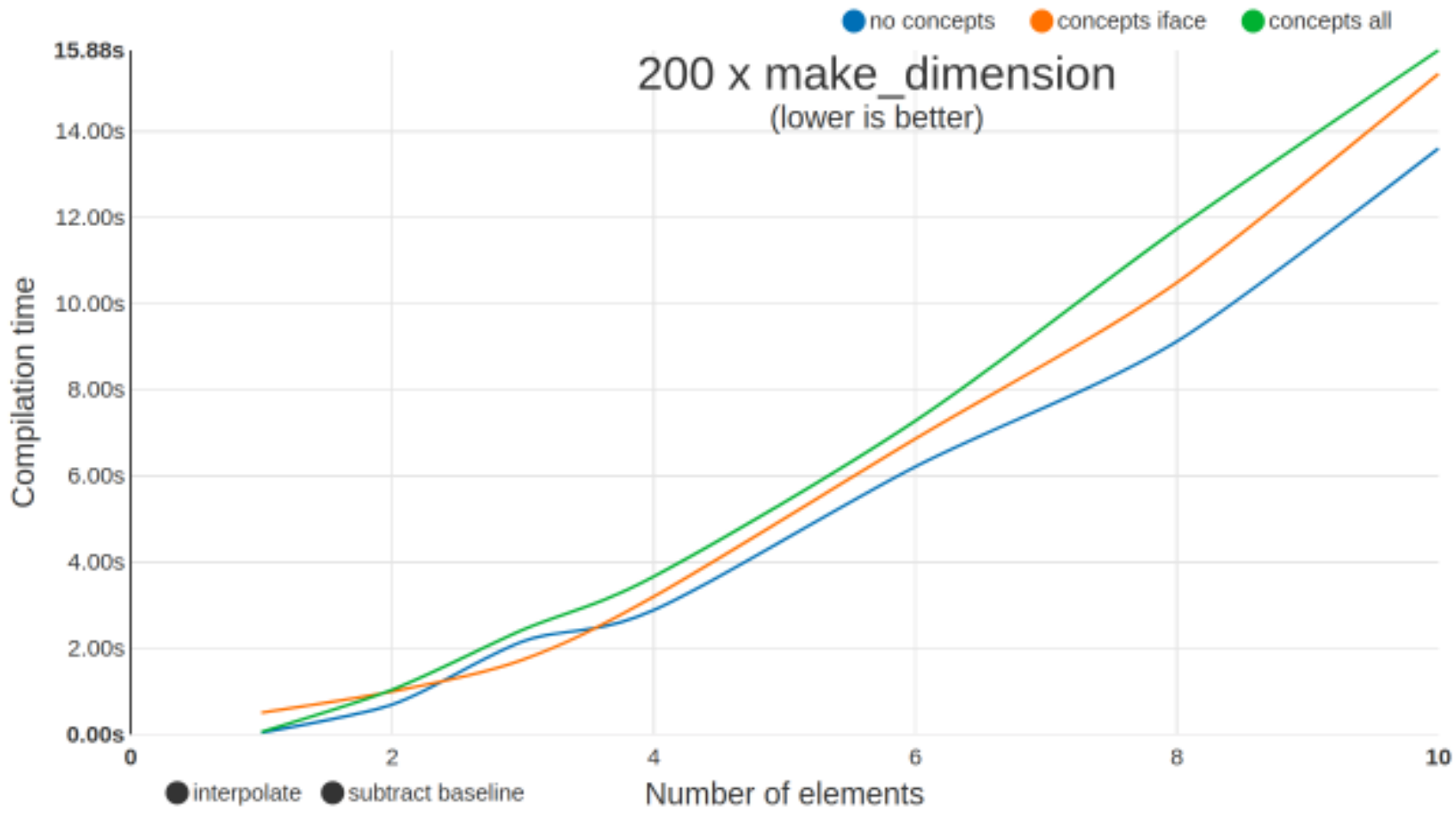
- C++ Concepts are great and *not too expensive at compile-time*

What about C++ Concepts?

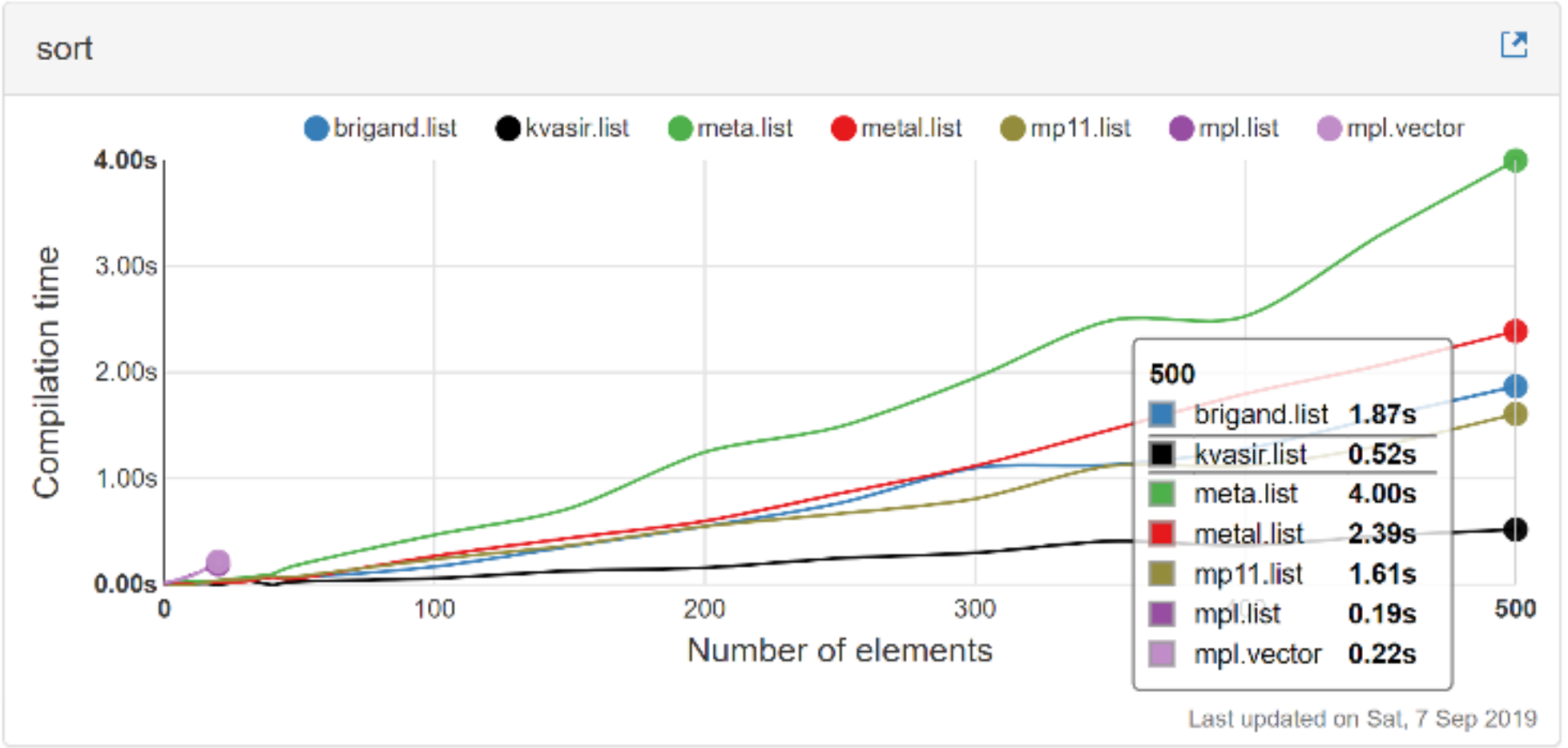
- C++ Concepts are great and *not too expensive at compile-time*
- Compile-time performance impact might be *limited by using concepts only in the user interfaces*

```
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

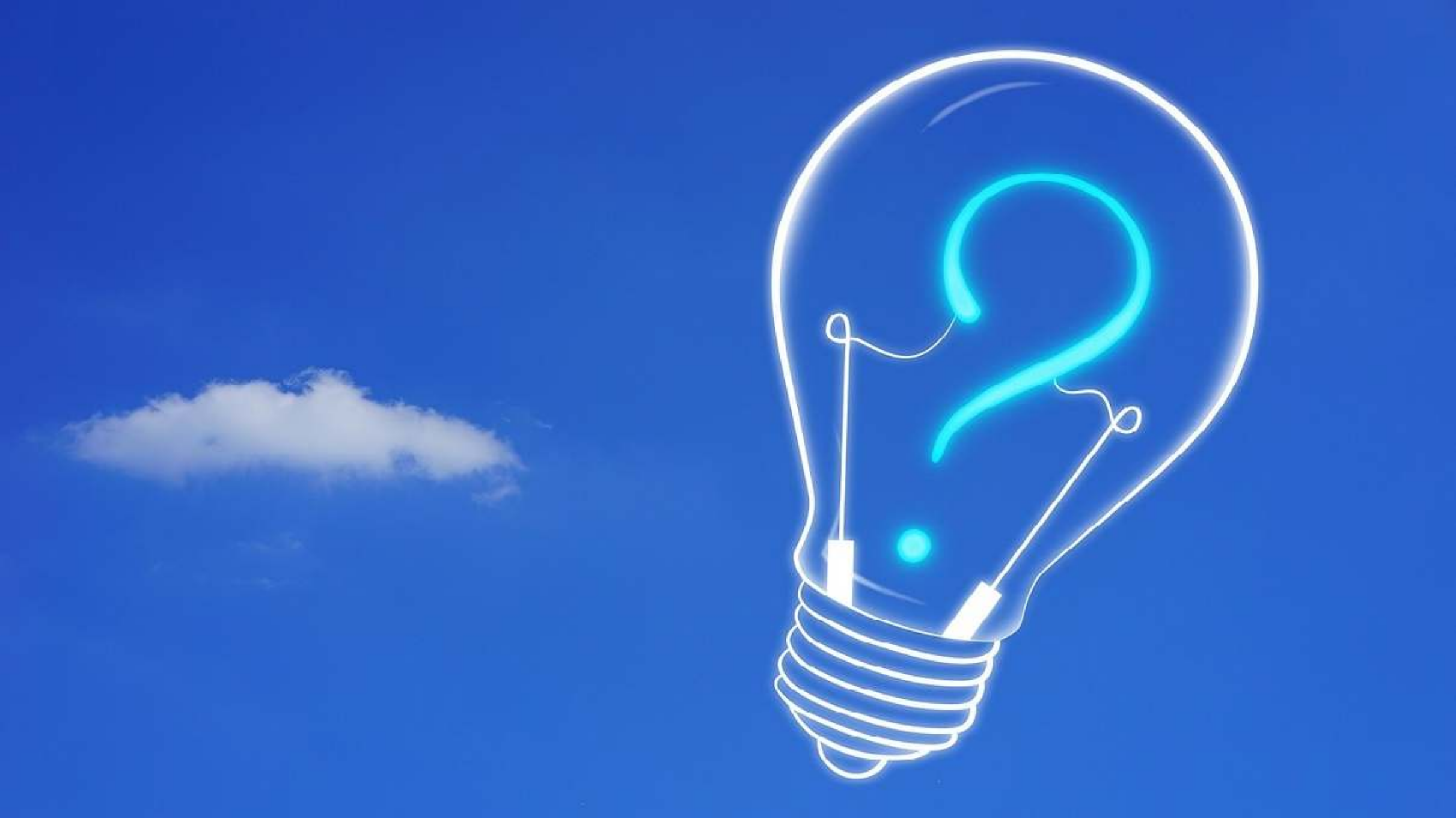
1. Think about **end users' experience** and not only about your convenience as a developer
2. Use **C++ Concepts** to
   - express compile-time contracts
   - improve productivity
   - improve compile-time errors

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)