



# Rethinking the Way We Do Templates in C++

(EVEN MORE)

Mateusz Pusz

June 30, 2020



# Plan for the talk

---

1 User experience

2 New toys in a toolbox

3 Wishful thinking

4 Performance

## **USER EXPERIENCE**

# Physical Units library in a nutshell

---

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

# Physical Units library in a nutshell

---

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

```
// unit conversions
static_assert(1q_h == 3600q_s);
static_assert(1q_km + 1q_m == 1001q_m);
```

# Physical Units library in a nutshell

---

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

```
// unit conversions
static_assert(1q_h == 3600q_s);
static_assert(1q_km + 1q_m == 1001q_m);
```

```
// dimension conversions
static_assert(1q_km / 1q_s == 1000q_m_per_s);
static_assert(2q_km_per_h * 2q_h == 4q_km);
static_assert(2q_km / 2q_km_per_h == 1q_h);

static_assert(2q_m * 3q_m == 6q_m2);

static_assert(10q_km / 5q_km == 2);

static_assert(1000 / 1q_s == 1q_kHz);
```

# Toy example

---

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

# Toy example

---

```
/* speed */ 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

---

```
/* speed */ 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

---

```
/* speed */ 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

---

```
/* speed */ 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 **doubles**

# 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 LOOOONG 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<-1, 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

---

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

# A need to modernize our toolbox

---

- In 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

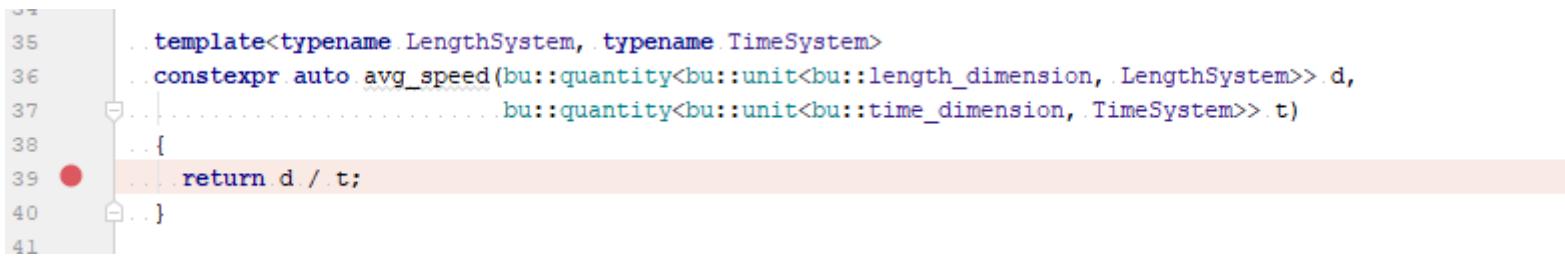
---

- In 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

---



A screenshot of a debugger interface showing a C++ code editor. A red circular breakpoint marker is positioned on the left margin of line 39. The code is a template function for calculating average speed:

```
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

The screenshot shows a debugger interface with a code editor and a variables panel.

**Code Editor:**

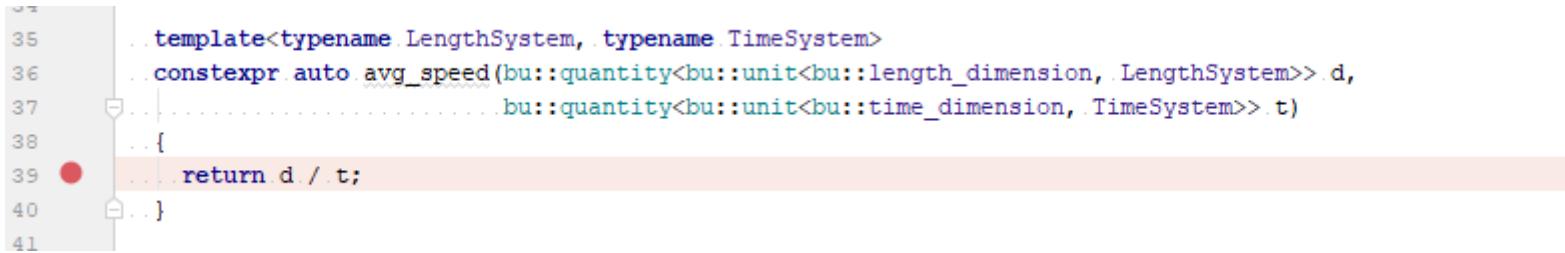
```
35     ...template<typename LengthSystem, typename TimeSystem>
36     ...constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>> d,
37     ...                                bu::quantity<bu::unit<bu::time_dimension, TimeSystem>> t)
38     ...
39     return d / t;
40 }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

A red dot marks the current line of execution at `return d / t;`.

**Variables Panel:**

Variable	Type	Value
<code>d</code>	<code>{boost::units::quantity&lt;boost::units::unit, double&gt;}</code>	<code>01 val_ = {boost::units::quantity&lt;boost::units::unit, double&gt;::value_type} 220</code>
<code>t</code>	<code>{boost::units::quantity&lt;boost::units::unit, double&gt;}</code>	<code>01 val_ = {boost::units::quantity&lt;boost::units::unit, double&gt;::value_type} 2</code>

# User's experience: Debugging: Boost.Units

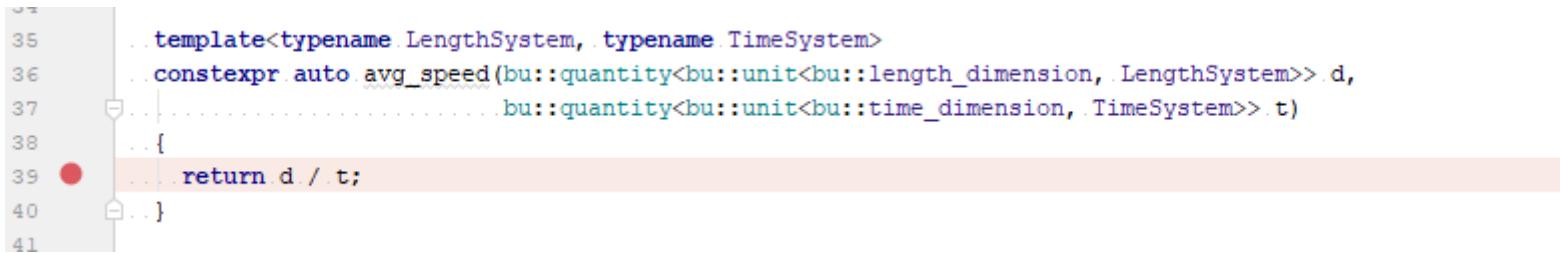


A screenshot of a debugger interface showing a C++ code editor. A red dot marks a breakpoint at line 39. The code is a template function for calculating average speed:

```
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 dim_speed : derived_dimension<exp<dim_length, 1>, exp<dim_time, -1>> {};
```

# Inheritance as a workaround

---

```
struct dim_speed : derived_dimension<exp<dim_length, 1>, exp<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 dim_speed : derived_dimension<exp<dim_length, 1>, exp<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 dim_speed : derived_dimension<exp<dim_length, 1>, exp<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 `derived_dimension` and `scaled_unit`

# Type substitution problem

---

```
Speed auto v = 10q_m / 2q_s;
```

# Type substitution problem

---

```
Speed auto v = 10q_m / 2q_s;
```

```
template<typename D1, typename U1, typename Rep1, typename D2, typename U2, typename Rep2>
[[nodiscard]] constexpr Quantity auto operator/(const quantity<D1, U1, Rep1>& lhs,
                                              const quantity<D2, U2, Rep2>& rhs);
```

# Type substitution problem

```
Speed auto v = 10q_m / 2q_s;
```

```
template<typename D1, typename U1, typename Rep1, typename D2, typename U2, typename Rep2>
[[nodiscard]] constexpr Quantity auto operator/(const quantity<D1, U1, Rep1>& lhs,
                                              const quantity<D2, U2, Rep2>& rhs);
```

How to form a **speed dimension child class** from division of **length** by **time**?

# Downcasting facility (Version 1.0)

---

```
template<typename T>
struct downcast_traits : std::type_identity<T> {};

template<typename T>
using downcast = downcast_traits<T>::type;
```

# Downcasting facility (Version 1.0)

---

```
template<typename T>
struct downcast_traits : std::type_identity<T> {};
```

```
template<typename T>
using downcast = downcast_traits<T>::type;
```

```
struct dim_speed : derived_dimension<exp<dim_length, 1>, exp<dim_time, -1>> {};
```

```
template<>
struct downcast_traits<derived_dimension<exp<dim_length, 1>, exp<dim_time, -1>>> : std::type_identity<dim_speed> {};
```

# Downcasting facility (Version 1.0)

---

```
template<typename T>
struct downcast_traits : std::type_identity<T> {};
```

```
template<typename T>
using downcast = downcast_traits<T>::type;
```

```
struct dim_speed : derived_dimension<exp<dim_length, 1>, exp<dim_time, -1>> {};
```

```
template<>
struct downcast_traits<derived_dimension<exp<dim_length, 1>, exp<dim_time, -1>>> : std::type_identity<dim_speed> {};
```

```
struct dim_area : derived_dimension<exp<dim_length, 2>> {};
```

```
template<>
struct downcast_traits<derived_dimension<exp<dim_length, 2>>> : std::type_identity<dim_area> {};
```

# Downcasting facility (Version 1.0)

```
template<typename T>
struct downcast_traits : std::type_identity<T> {};
```

```
template<typename T>
using downcast = downcast_traits<T>::type;
```

```
struct dim_speed : derived_dimension<exp<dim_length, 1>, exp<dim_time, -1>> {};
```

```
template<>
struct downcast_traits<derived_dimension<exp<dim_length, 1>, exp<dim_time, -1>>> : std::type_identity<dim_speed> {};
```

```
struct dim_area : derived_dimension<exp<dim_length, 2>> {};
```

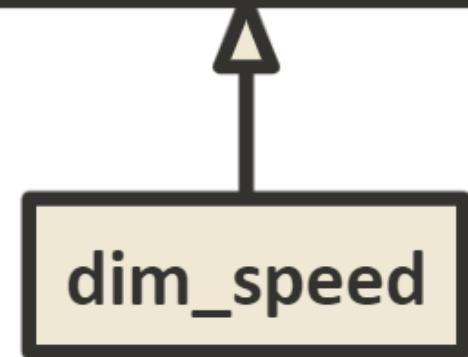
```
template<>
struct downcast_traits<derived_dimension<exp<dim_length, 2>>> : std::type_identity<dim_area> {};
```

**downcast\_traits** specialization had to be provided for every derived dimension in the library

# Downcasting facility (Version 2.0): Overview

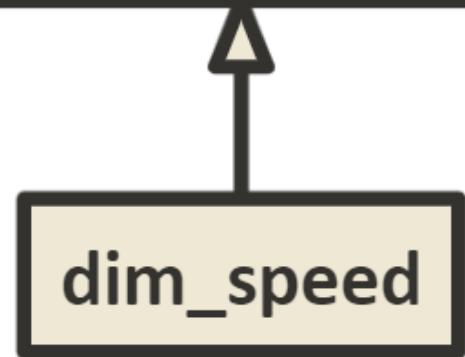
---

```
derived_dimension<exp<dim_length, 1>, exp<dim_time, -1>>
```



# Downcasting facility (Version 2.0): Overview

```
derived_dimension<exp<dim_length, 1>, exp<dim_time, -1>>
```

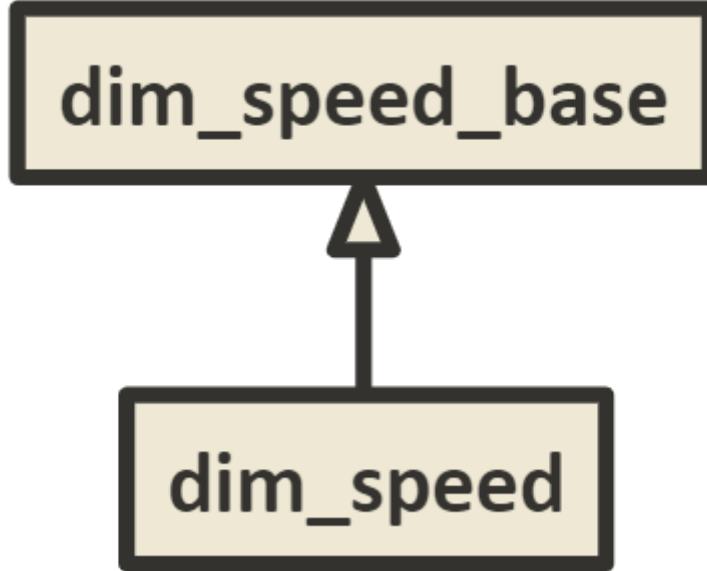


- For slideware let's assume

```
using dim_speed_base = derived_dimension<exp<dim_length, 1>, exp<dim_time, -1>>;
```

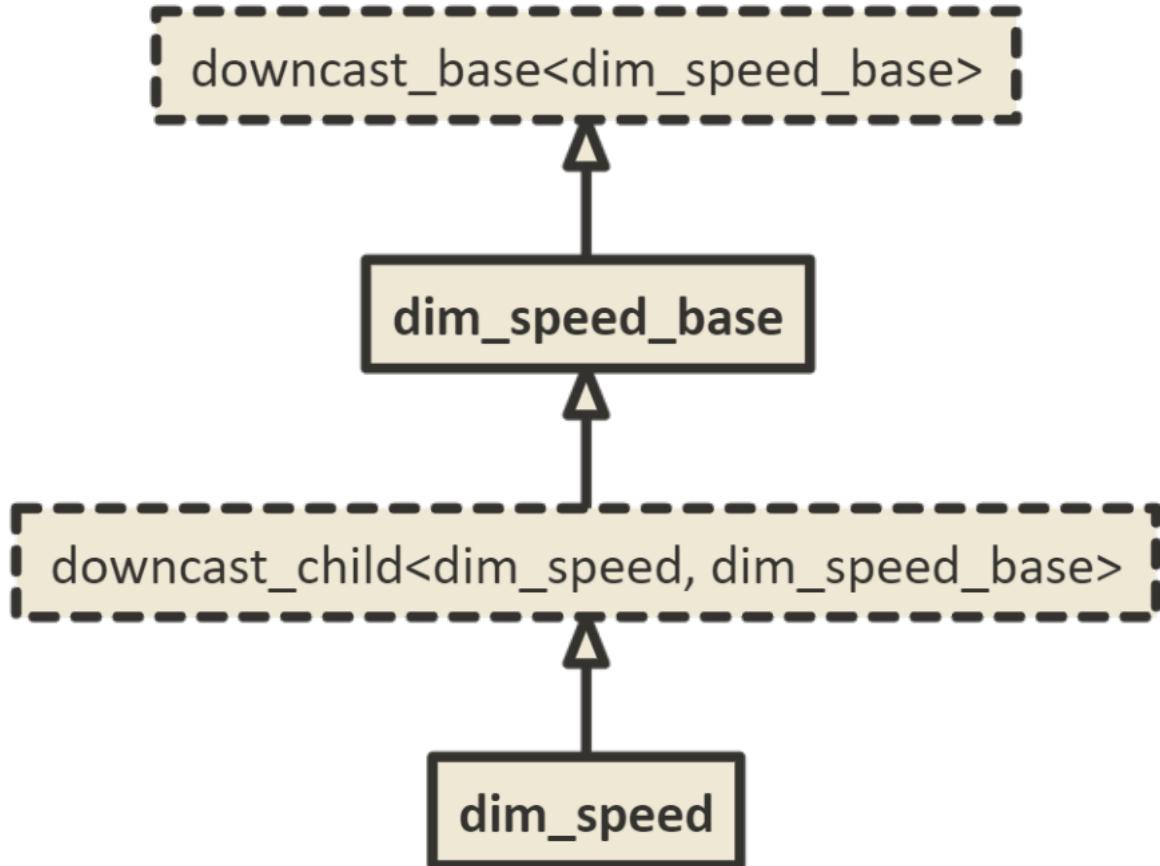
# Downcasting facility (Version 2.0): Overview

---

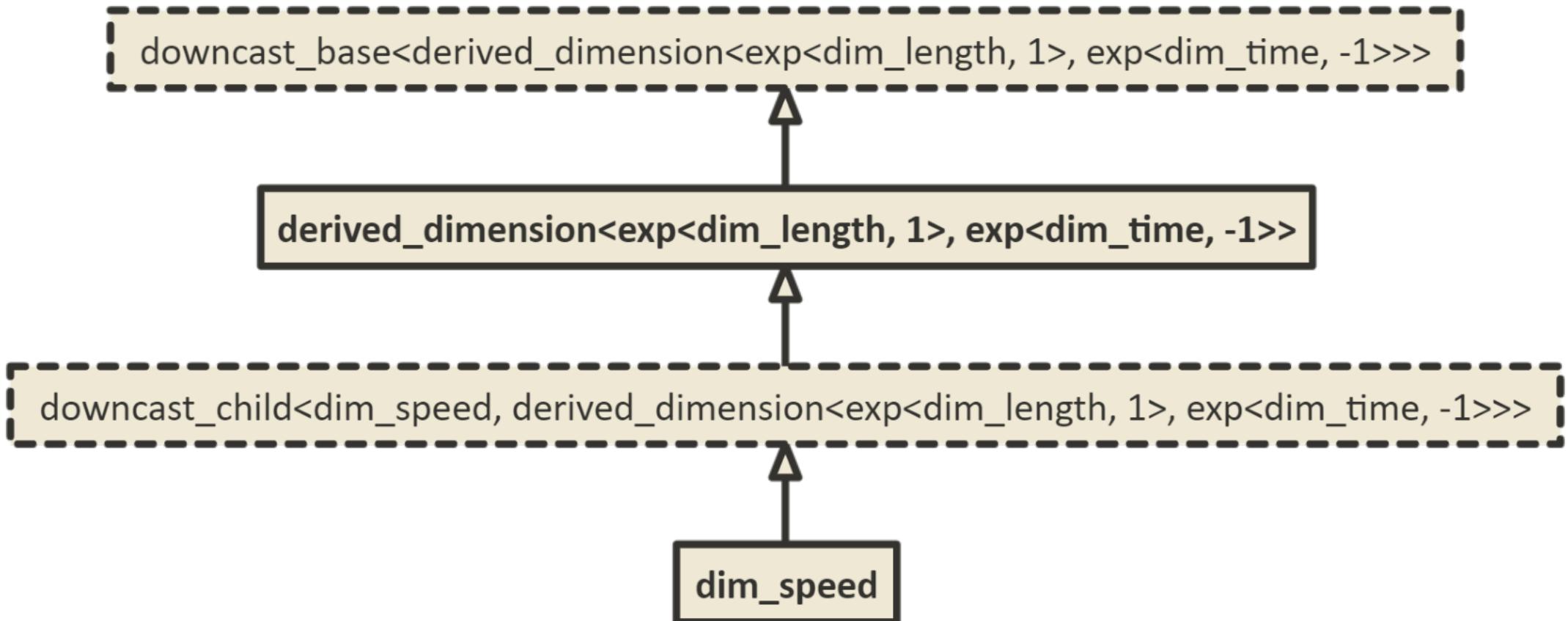


# Downcasting facility (Version 2.0): Overview

---



# Downcasting facility (Version 2.0): Overview



# Downcasting facility: Design

---

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

# Downcasting facility: Design

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

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

# Downcasting facility: Design

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

```
template<typename T>
concept Downcastable =
    requires {
        typename T::downcast_base_type;
    } &&
    std::derived_from<T, downcast_base<typename T::downcast_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_impl<T>());
```

# Downcasting facility: Design

```
namespace detail {

    template<typename T>
    constexpr auto downcast_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_impl<T>());
}
```

# Downcasting facility: Design

```
namespace detail {

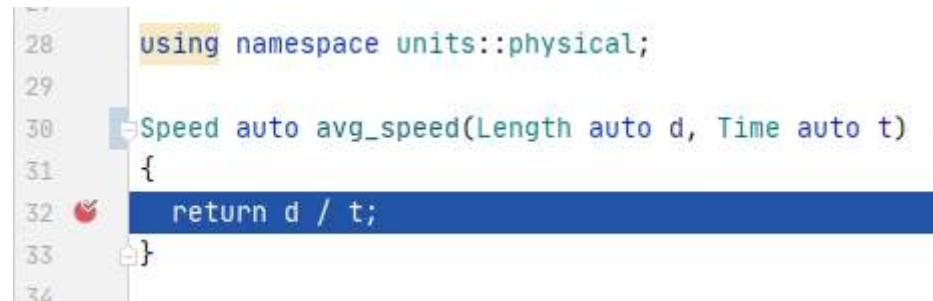
    template<typename T>
    concept has_downcast = requires {
        downcast_guide(std::declval<downcast_base<T>>());
    };

    template<typename T>
    constexpr auto downcast_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_impl<T>());
}
```

# User experience: Debugging

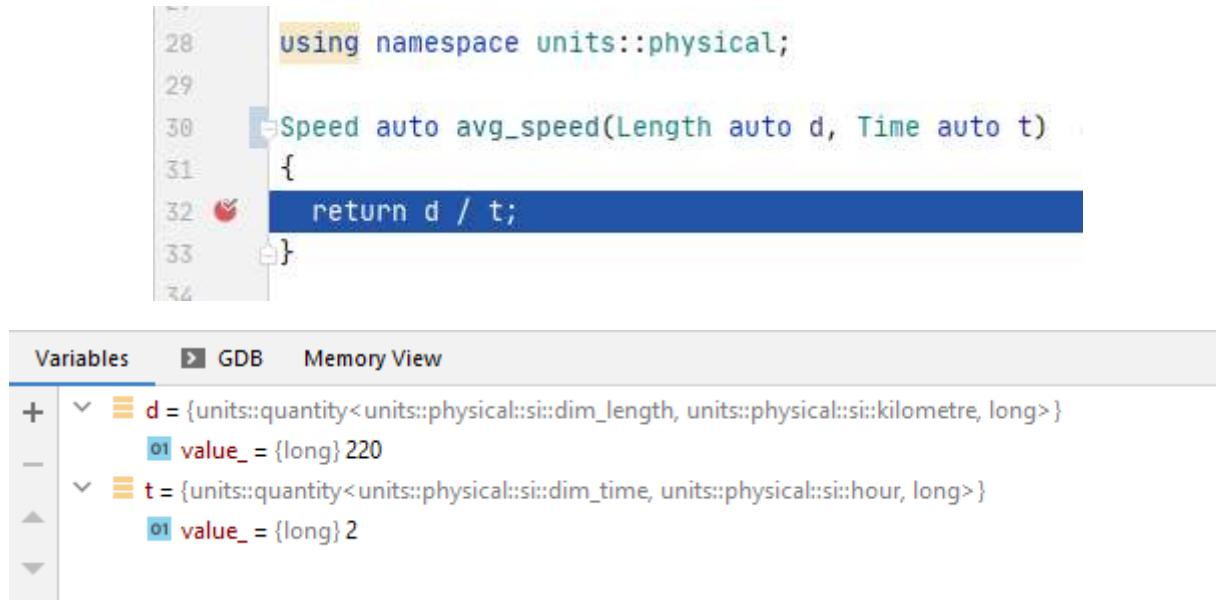
---



```
28     using namespace units::physical;
29
30     Speed auto avg_speed(Length auto d, Time auto t)
31     {
32         return d / t;
33     }
34 
```

A screenshot of a code editor showing a debugger interface. A callout box points from the top left to the division operator in the line `return d / t;`. The callout box contains a red breakpoint icon and the word 'Breakpoint'. The code editor shows several lines of C++ code, including `using namespace units::physical;`, a function definition for `avg\_speed` taking `Length` and `Time` parameters and returning a `Speed`, and a single-line return statement.

# User experience: Debugging



The screenshot shows a debugger interface with two main panes. The top pane displays a portion of C++ code:

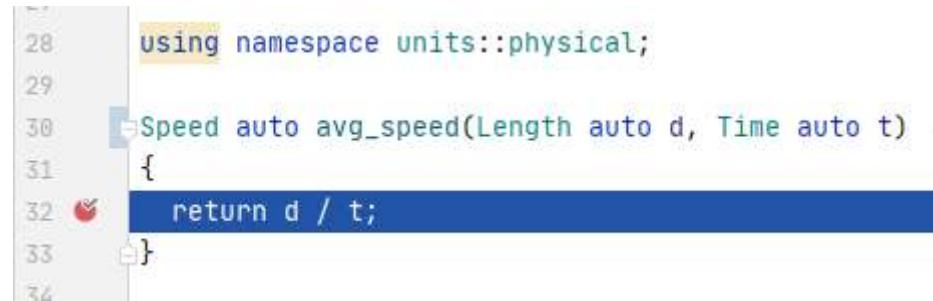
```
28     using namespace units::physical;
29
30     Speed auto avg_speed(Length auto d, Time auto t)
31     {
32         return d / t;
33     }
34 
```

The line `return d / t;` is highlighted with a blue bar, and a red breakpoint icon is visible on line 32. The bottom pane is the Variables view, showing the current values of variables `d` and `t`:

Variables	GDB	Memory View
+ d = {units::quantity<units::physical::si::dim_length, units::physical::si::kilometre, long>} 01 value_ = {long} 220		
- t = {units::quantity<units::physical::si::dim_time, units::physical::si::hour, long>} 01 value_ = {long} 2		

# User experience: Debugging

---



A screenshot of a debugger interface showing a C++ code editor. The code is as follows:

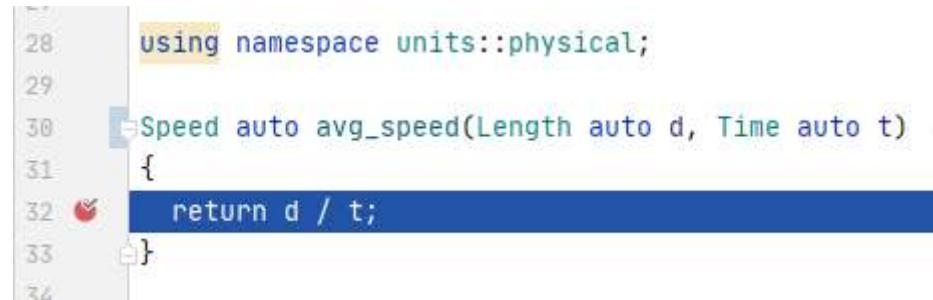
```
28     using namespace units::physical;
29
30     Speed auto avg_speed(Length auto d, Time auto t)
31     {
32         return d / t;
33     }
34 
```

The line `return d / t;` is highlighted with a blue bar, and a red breakpoint icon is visible on the left margin next to line 32.

Breakpoint 1, avg\_speed<units::quantity<units::physical::si::dim\_length, units::physical::si::kilometre, long>,  
                          units::quantity<units::physical::si::dim\_time, units::physical::si::hour, long> >  
(d=..., t=...) at hello\_units.cpp:32  
32      return d / t;

# User experience: Debugging

---



A screenshot of a code editor showing a C++ file. The code defines a function `avg_speed` that takes `Length` and `Time` as parameters and returns a `Speed`. A red breakpoint icon is visible on line 32, where the expression `d / t;` is highlighted in blue.

```
28     using namespace units::physical;
29
30     Speed auto avg_speed(Length auto d, Time auto t)
31     {
32         return d / t;
33     }
34 }
```

```
(gdb) ptype d
type = class units::quantity<units::physical::si::dim_length, units::physical::si::kilometre, long>
[with D = units::physical::si::dim_length, U = units::physical::si::kilometre, Rep = long] {
...
...
```

# 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;
    static constexpr intmax_t 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>;
};
```

# 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))
    {
    }

    // ...
};
```

# 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

BEFORE

```
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 in action

## BEFORE

```
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>> {};
```

## AFTER

```
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

```
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>> {};
```

## AFTER

```
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

```
static_assert(std::is_same_v<ratio_multiply<ratio<2>, ratio<3, 8>>::type, ratio<3, 4>::type>);
```

# NTTP in action

---

BEFORE

```
static_assert(std::is_same_v<ratio_multiply<ratio<2>, ratio<3, 8>>::type, ratio<3, 4>::type>);
```

AFTER

```
static_assert(ratio(2) * ratio(3, 8) == ratio(3, 4));
```

# NTTP in action

---

## BEFORE

```
template<typename ExpList>
struct base_units_ratio;

template<typename E>
struct base_units_ratio<exp_list<E>> {
    using type = exp_ratio<E>::type;
};

template<typename E, typename... Es>
struct base_units_ratio<exp_list<E, Es...>> {
    using type = ratio_multiply<typename exp_ratio<E>::type, typename base_units_ratio<exp_list<Es...>>::type>;
};
```

# NTTP in action

## BEFORE

```
template<typename ExpList>
struct base_units_ratio;

template<typename E>
struct base_units_ratio<exp_list<E>> {
    using type = exp_ratio<E>::type;
};

template<typename E, typename... Es>
struct base_units_ratio<exp_list<E, Es...>> {
    using type = ratio_multiply<typename exp_ratio<E>::type, typename base_units_ratio<exp_list<Es...>>::type>;
};
```

## AFTER

```
template<typename... Es>
constexpr ratio base_units_ratio(exp_list<Es...>)
{
    return (exp_ratio<Es>() * ...);
}
```

# 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

# 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

If a template parameter behaves like a value it probably should be an NTTP.

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

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

# How do you feel about such an interface?

---

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

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

```
template<typename T> auto foo(T&& 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*

# 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

# Sicily's Medieval Map of the World

---



# Sicily's Medieval Map of the World



Most of the experience we have with concepts comes from the range-v3 library (algorithms). Let's try to view it from another angle too...

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

---



# 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

---

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

---

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

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...

# The world is different and bigger than we initially imagined

---



# 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;
      ^~~
```

# Maybe we do not need named concepts here?

---

```
template<typename R1, typename R2>
    requires is_ratio<R1> && is_ratio<R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

# Maybe we do not need named concepts here?

---

```
template<typename R1, typename R2>
    requires is_ratio<R1> && is_ratio<R2>
using ratio_multiply = std::ratio_multiply<R1, R2>;
```

But named concepts are really handy in this and many other cases...

# Concepts example

---

```
template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

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

# Concepts example

---

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

```
template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

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

# Concepts example

---

```
template<typename T>
concept Quantity = is_instantiation_of<quantity>;  
  
template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && equivalent_dim<typename T::dimension, D>;  
  
template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

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

# Concepts example

```
template<typename T>
concept Dimension = BaseDimension<T> || DerivedDimension<T>;  
  
template<typename T>
concept Quantity = is_instantiation_of<quantity>;  
  
template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && equivalent_dim<typename T::dimension, D>;  
  
template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

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

# Concepts example

```
template<typename T>
concept DerivedDimension = is_derived_from_instantiation_of<T, derived_dimension_base>;  
  
template<typename T>
concept Dimension = BaseDimension<T> || DerivedDimension<T>;  
  
template<typename T>
concept Quantity = is_instantiation_of<quantity>;  
  
template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && equivalent_dim<typename T::dimension, D>;  
  
template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

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

# Concepts example

```
template<typename T>
concept BaseDimension = is_derived_from_base_dimension<T>;  
  
template<typename T>
concept DerivedDimension = is_derived_from_instantiation_of<T, derived_dimension_base>;  
  
template<typename T>
concept Dimension = BaseDimension<T> || DerivedDimension<T>;  
  
template<typename T>
concept Quantity = is_instantiation_of<quantity>;  
  
template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && equivalent_dim<typename T::dimension, D>;  
  
template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

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

# Not just a syntactic sugar

---

# Not just a syntactic sugar

---

- Constraining *function template return types*

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

# Not just a syntactic sugar

---

- Constraining *function template return types*

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

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

```
const units::Speed auto speed = avg_speed(220.km, 2.h);
```

# Not just a syntactic sugar

---

- Constraining *function template return types*

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

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

```
const units::Speed auto speed = avg_speed(220.km, 2.h);
```

- Constraining *class template parameters* without introducing more parameters

```
template<Dimension D, ratio R>
    requires UnitRatio<R>
struct unit;
```

```
template<typename Q, direction D>
    requires Quantity<Q> || QuantityPoint<Q>
class vector;
```

# What is the difference?

---

```
template<typename T>
class wrapper {
    T data_;
public:
    template<typename U,
              typename = std::enable_if_t<std::is_integral_v<T>>>
    void foo(U u);

    // ...
};
```

```
template<typename T>
class wrapper {
    T data_;
public:
    template<typename U>
    requires std::is_integral_v<T>
    void foo(U u);

    // ...
};
```

# What is the difference?

```
template<typename T>
class wrapper {
    T data_;
public:
    template<typename U, typename Z = T,
              requires std::enable_if_t<std::is_integral_v<Z>>>
        void foo(U u);

    // ...
};
```

```
template<typename T>
class wrapper {
    T data_;
public:
    template<typename U>
        requires std::is_integral_v<T>
        void foo(U u);

    // ...
};
```

SFINAE works only on current template parameters.  
Concepts just work as expected.

# What is the difference?

```
template<typename T>
class wrapper {
    T data_;
public:
    template<typename U, typename Z = T,
              typename = std::enable_if_t<std::is_integral_v<Z>>>
        void foo(U u);

    template<typename Z = T,
              typename = std::enable_if_t<std::is_integral_v<Z>>>
        void boo(Z t);

    // ...
};
```

```
template<typename T>
class wrapper {
    T data_;
public:
    template<typename U>
        requires std::is_integral_v<T>
        void foo(U u);

    void boo(T t)
        requires std::is_integral_v<T>;
    // ...
};
```

SFINAE works only on current template parameters.  
Concepts just work as expected.

# 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*

# And you know what? It is round ;-)

---



## WISHFUL THINKING

# 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;
```

# Sometimes you do not want a named concept

---

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

# Sometimes you do not want a named concept

---

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

```
template<typename T>
struct is_ratio : std::false_type {};  
  
template<intmax_t N, intmax_t D>
struct is_ratio<std::ratio<N, D>> : std::true_type {};
```

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

# Sometimes you do not want a named concept

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

```
template<typename T>
struct is_ratio : std::false_type {};  
  
template<intmax_t N, intmax_t D>
struct is_ratio<std::ratio<N, D>> : std::true_type {};
```

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

```
template<typename T>
inline constexpr bool is_ratio = false;  
  
template<intmax_t N, intmax_t D>
inline constexpr bool is_ratio<std::ratio<N, D>> = true;
```

- does not compile :-)

# Sometimes you do not want a named concept

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

```
template<typename T>
struct is_ratio : std::false_type {};  
  
template<intmax_t N, intmax_t D>
struct is_ratio<std::ratio<N, D>> : std::true_type {};
```

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

```
template<typename T>
inline constexpr bool is_ratio = false;  
  
template<intmax_t N, intmax_t D>
inline constexpr bool is_ratio<std::ratio<N, D>> = true;
```

- does not compile :-)

Template template parameter kind works only for class templates

# Making variable templates a first class citizen (P2008)

---

## CLASS TEMPLATES

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

# Making variable templates a first class citizen (P2008)

---

## CLASS TEMPLATES

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

## VARIABLE TEMPLATES

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

# Making variable templates a first class citizen (P2008)

---

## CLASS TEMPLATES

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

## VARIABLE TEMPLATES

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

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

# Making variable templates a first class citizen (P2008)

## CLASS TEMPLATES

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

## VARIABLE TEMPLATES

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

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

If you have ideas for other cool use cases of this feature PLEASE let me know.

# 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<template auto>
class X;
```

```
template<template auto...>
class Y;
```

- takes **any kind** of template parameter
- takes **any number and any kind** of template parameters

# Concept to check for an instantiation of a class template

---

```
template<typename T, template<template auto...> typename Type>
inline constexpr bool is_instantiation_of = false;

template<template auto... Params, template<template auto...> typename Type>
inline constexpr bool is_instantiation_of<Type<Params...>, Type> = true;
```

# Concept to check for an instantiation of a class template

---

```
template<typename T, template<template auto...> typename Type>
inline constexpr bool is_instantiation_of = false;

template<template auto... Params, template<template auto...> typename Type>
inline constexpr bool is_instantiation_of<Type<Params...>, Type> = true;
```

```
template<typename T, template<template auto...> typename Type>
concept instantiation_of = is_instantiation_of<T, Type>;
```

# What about our std::ratio\_multiply?

---

```
template<instantiation_of<std::ratio> R1, instantiation_of<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<instantiation_of<std::ratio> R1, instantiation_of<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
(instantiation_of<R1, std::ratio>) && (instantiation_of<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 'instantiation_of<std::__cxx11::basic_string<char, std::char_traits<char>,
  std::allocator<char> >, std::ratio>'
<source>:21:14: note: the expression 'is_instantiation_of<T, Type>' evaluated to 'false'
21 | concept instantiation_of = is_instantiation_of<T, Type>;
      ^~~~~~
```

# Which style do you prefer?

---

```
// C-like
unit named_coherent_derived_unit(void* dimension, void* symbol, void* prefix_type);
unit coherent_derived_unit(void* dimension);
unit named_scaled_derived_unit(void* dimension, void* symbol, void* ratio, void* prefix_type);
unit named_deduced_derived_unit(void* dimension, void* symbol, void* prefix_type, void* unit, ...);
unit deduced_derived_unit(void* dimension, void* unit, ...);
unit prefixed_derived_unit(void* prefix, void* unit);
```

# Which style do you prefer?

```
// C-like
unit named_coherent_derived_unit(void* dimension, void* symbol, void* prefix_type);
unit coherent_derived_unit(void* dimension);
unit named_scaled_derived_unit(void* dimension, void* symbol, void* ratio, void* prefix_type);
unit named_deduced_derived_unit(void* dimension, void* symbol, void* prefix_type, void* unit, ...);
unit deduced_derived_unit(void* dimension, void* unit, ...);
unit prefixed_derived_unit(void* prefix, void* unit);
```

```
// strong_types + C-like functions
unit named_coherent_derived_unit(dimension dim, string symbol, prefix_type pt);
unit coherent_derived_unit(dimension dim);
unit named_scaled_derived_unit(dimension dim, string symbol, ratio r, prefix_type pt);
unit named_deduced_derived_unit(dimension dim, string symbol, prefix_type pt, unit u, ...);
unit deduced_derived_unit(dimension dim, unit u, ...);
unit prefixed_derived_unit(prefix p, unit u);
```

# Which style do you prefer?

```
// C-like
unit named_coherent_derived_unit(void* dimension, void* symbol, void* prefix_type);
unit coherent_derived_unit(void* dimension);
unit named_scaled_derived_unit(void* dimension, void* symbol, void* ratio, void* prefix_type);
unit named_deduced_derived_unit(void* dimension, void* symbol, void* prefix_type, void* unit, ...);
unit deduced_derived_unit(void* dimension, void* unit, ...);
unit prefixed_derived_unit(void* prefix, void* unit);
```

```
// strong types + C-like functions
unit named_coherent_derived_unit(dimension dim, string symbol, prefix_type pt);
unit coherent_derived_unit(dimension dim);
unit named_scaled_derived_unit(dimension dim, string symbol, ratio r, prefix_type pt);
unit named_deduced_derived_unit(dimension dim, string symbol, prefix_type pt, unit u, ...);
unit deduced_derived_unit(dimension dim, unit u, ...);
unit prefixed_derived_unit(prefix p, unit u);
```

```
// strong types + function overloading
unit derived_unit(dimension dim, string symbol, prefix_type pt);
unit derived_unit(dimension dim);
unit derived_unit(dimension dim, string symbol, ratio r, prefix_type pt);
unit derived_unit(dimension dim, string symbol, prefix_type pt, unit u, ...);
unit derived_unit(dimension dim, unit u, ...);
unit derived_unit(prefix p, unit u);
```

# What about class templates?

## PRE-CONCEPTS

```
template<typename Child, typename Dim, basic_fixed_string Symbol, typename PT>
struct named_coherent_derived_unit;

template<typename Child, typename Dim>
struct coherent_derived_unit;

template<typename Child, typename Dim, basic_fixed_string Symbol, typename R, typename PT = no_prefix>
struct named_scaled_derived_unit;

template<typename Child, typename Dim, basic_fixed_string Symbol, typename PT, typename U, typename... Us>
struct named_deduced_derived_unit;

template<typename Child, typename Dim, typename U, typename... Us>
struct deduced_derived_unit;

template<typename Child, typename P, typename U>
struct prefixed_derived_unit;
```

# What about class templates?

## CONCEPTS

```
template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT>
struct named_coherent_derived_unit;

template<typename Child, Dimension Dim>
struct coherent_derived_unit;

template<typename Child, Dimension Dim, basic_fixed_string Symbol, Ratio R, PrefixType PT = no_prefix>
struct named_scaled_derived_unit;

template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT, Unit U, Unit... Us>
struct named_deduced_derived_unit;

template<typename Child, Dimension Dim, Unit U, Unit... Us>
    requires U::is_named && (Us::is_named && ... && true)
struct deduced_derived_unit;

template<typename Child, Prefix P, Unit U>
    requires (!std::same_as<typename U::prefix_type, no_prefix>)
struct prefixed_derived_unit;
```

# What about class templates?

---

## CLASS TEMPLATE PARTIAL SPECIALIZATION OVERLOADING

```
template<template auto...>
struct derived_unit;
```

# What about class templates?

## CLASS TEMPLATE PARTIAL SPECIALIZATION OVERLOADING

```
template<template auto...>
struct derived_unit;
```

```
template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT>
struct derived_unit<Child, Dim, Symbol, PT>;
```

```
template<typename Child, Dimension Dim>
struct derived_unit<Child, Dim>;
```

```
template<typename Child, Dimension Dim, basic_fixed_string Symbol, Ratio R, PrefixType PT = no_prefix>
struct derived_unit<Child, Dim, Symbol, R, PT>;
```

```
template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT, Unit U, Unit... Us>
struct derived_unit<Child, Dim, Symbol, PT, U, Us...>;
```

```
template<typename Child, Dimension Dim, Unit U, Unit... Us>
  requires U::is_named && (Us::is_named && ... && true)
struct derived_unit<Child, Dim, U, Us...>;
```

```
template<typename Child, Prefix P, Unit U>
  requires (!std::same_as<typename U::prefix_type, no_prefix>)
struct derived_unit<Child, P, U>;
```

# Speed definition example (Today)

---

## LENGTH

```
struct metre : named coherent derived unit<metre, length, "m", si_prefix> {};
struct kilometre : prefixed derived unit<kilometre, kilo, metre> {};
struct yard : named scaled derived unit<yard, length, "yd", ratio(9'144, 10'000)> {};
struct mile : named scaled derived unit<mile, length, "mi", 1'760 * yard::ratio> {};
```

# Speed definition example (Today)

---

## LENGTH

```
struct metre : named_coherent_derived_unit<metre, length, "m", si_prefix> {};
struct kilometre : prefixed_derived_unit<kilometre, kilo, metre> {};
struct yard : named_scaled_derived_unit<yard, length, "yd", ratio(9'144, 10'000)> {};
struct mile : named_scaled_derived_unit<mile, length, "mi", 1'760 * yard::ratio> {};
```

## TIME

```
struct second : named_coherent_derived_unit<second, time, "s", si_prefix> {};
struct hour : named_scaled_derived_unit<hour, time, "h", 3600 * second::ratio> {};
```

# Speed definition example (Today)

## LENGTH

```
struct metre : named_coherent_derived_unit<metre, length, "m", si_prefix> {};
struct kilometre : prefixed_derived_unit<kilometre, kilo, metre> {};
struct yard : named_scaled_derived_unit<yard, length, "yd", ratio(9'144, 10'000)> {};
struct mile : named_scaled_derived_unit<mile, length, "mi", 1'760 * yard::ratio> {};
```

## TIME

```
struct second : named_coherent_derived_unit<second, time, "s", si_prefix> {};
struct hour : named_scaled_derived_unit<hour, time, "h", 3600 * second::ratio> {};
```

## SPEED

```
struct metre_per_second : coherent_derived_unit<metre_per_second, speed> {};
struct kilometre_per_hour : deduced_derived_unit<kilometre_per_hour, speed, kilometre, hour> {};
struct mile_per_hour : deduced_derived_unit<mile_per_hour, speed, mile, hour> {};
```

# Speed definition example (Future)

## LENGTH

```
struct metre : derived_unit<metre, length, "m", si_prefix> {};
struct kilometre : derived_unit<kilometre, kilo, metre> {};
struct yard : derived_unit<yard, length, "yd", ratio(9'144, 10'000)> {};
struct mile : derived_unit<mile, length, "mi", 1'760 * yard::ratio> {};
```

## TIME

```
struct second : derived_unit<second, time, "s", si_prefix> {};
struct hour : derived_unit<hour, time, "h", 3600 * second::ratio> {};
```

## SPEED

```
struct metre_per_second : derived_unit<metre_per_second, speed> {};
struct kilometre_per_hour : derived_unit<kilometre_per_hour, speed, kilometre, hour> {};
struct mile_per_hour : derived_unit<mile_per_hour, speed, mile, hour> {};
```

# 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

---

```
#include "ratio_type_constexpr.h"

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

struct test2 {
#ifndef 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

```
#include "ratio_type_constexpr.h"

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

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

// ...

int main() {}
```

```
#include "ratio_nttp.h"

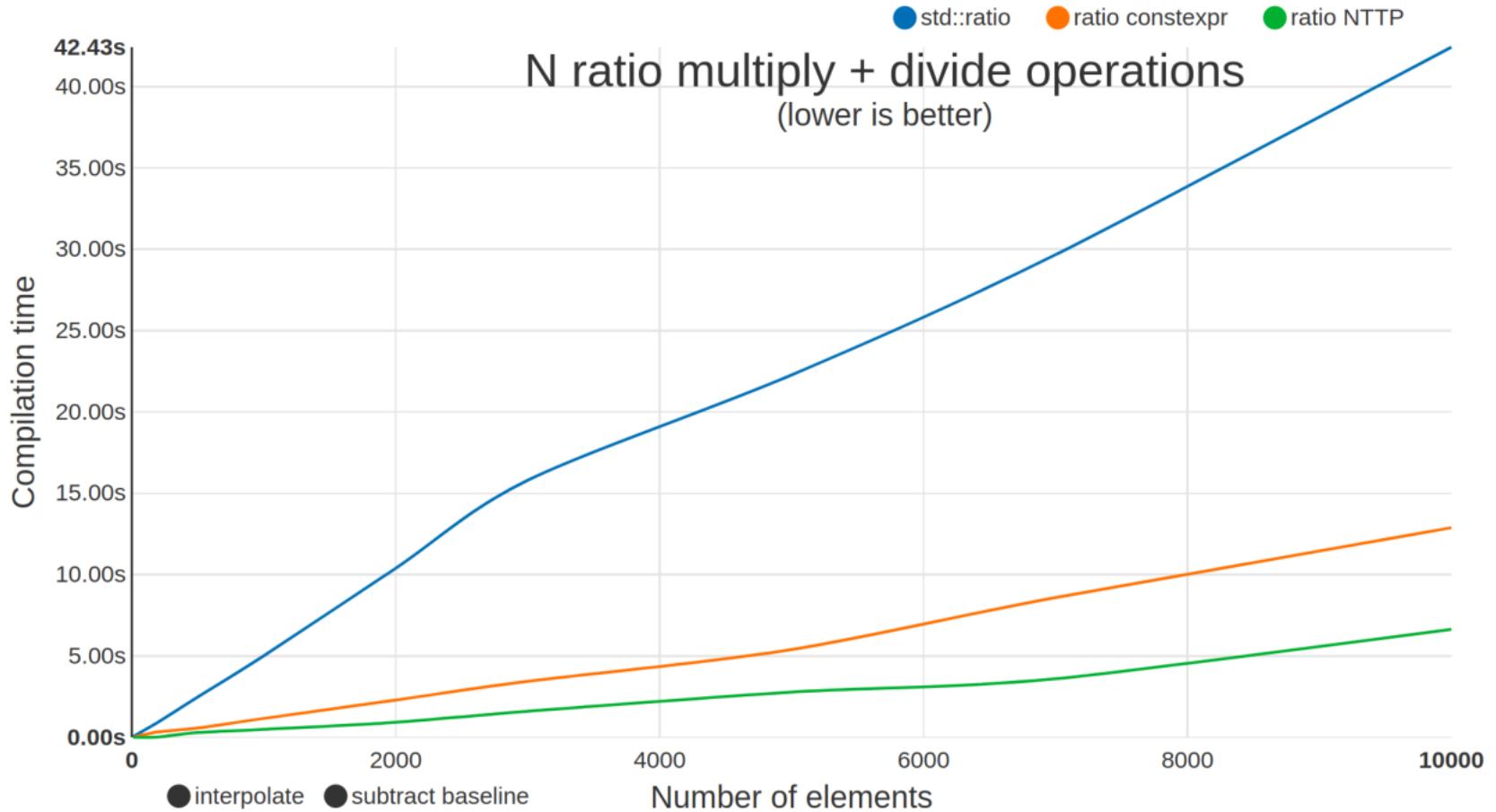
struct test1 {
#ifndef METABENCH
    static constexpr units::ratio r1{1, 20};
    static constexpr units::ratio r2{2, 20};
#else
    static constexpr bool r1 = false;
    static constexpr bool r2 = false;
#endif
};

struct test2 {
#ifndef 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
};

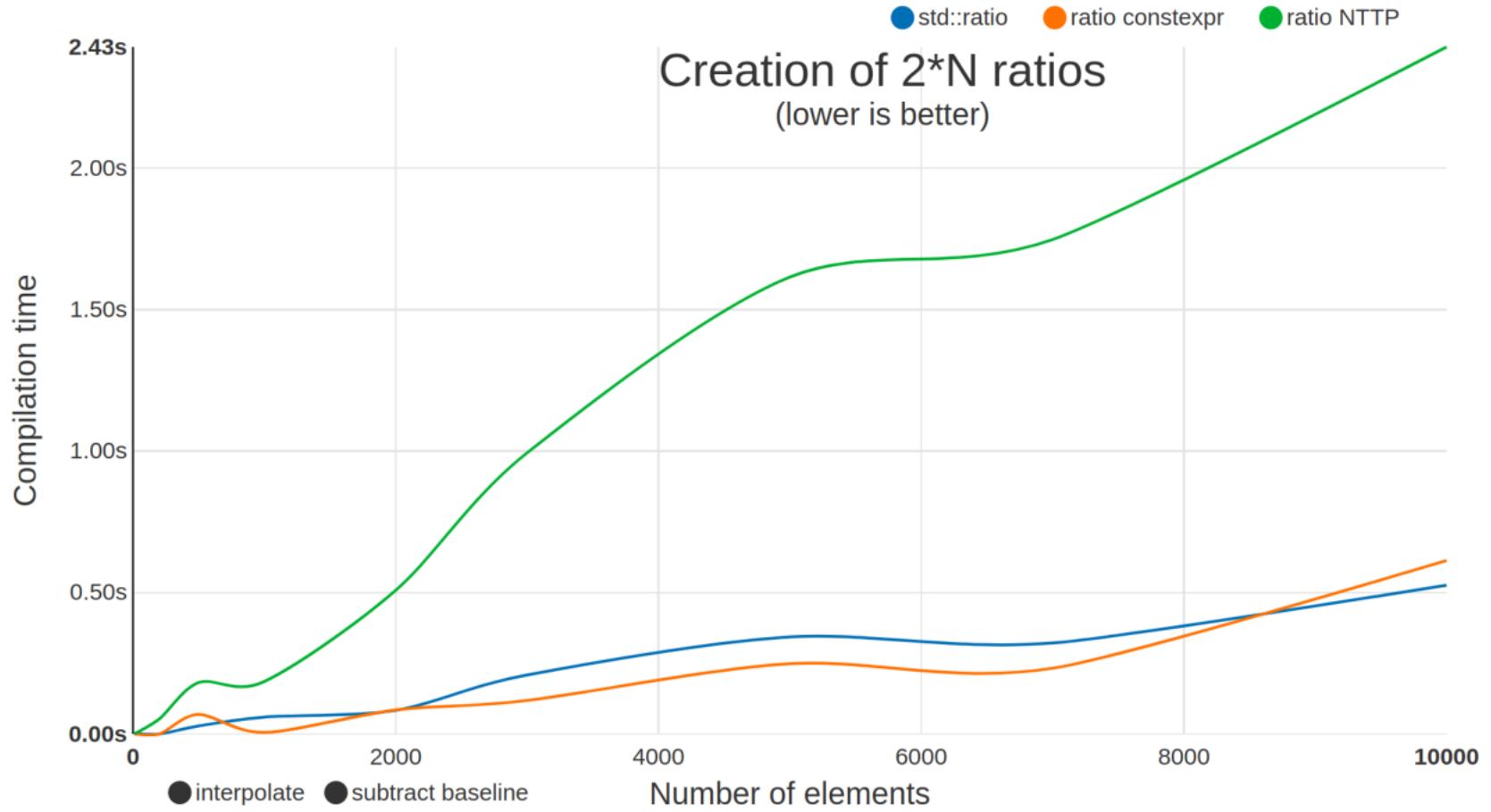
// ...

int main() {}
```

# ratio performance



# ratio performance



# Current constexpr QoI is slow

The screenshot shows a presentation slide titled "THE CONSTEXPR PROBLEM". The slide contains the following text:  
sloooooooooow  
2 minutes (gcc 9.1)  
40 seconds (clang 8)  
< 5 second (**php 7.1**)  
(NFA determinization)

On the right side of the slide, there is a photo of the speaker, Hana Dusíková, standing at a podium. Below the photo, her name is displayed: "Hana Dusíková". The title of the presentation is "Compile Time Regular Expressions with A Deterministic Finite Automaton". At the bottom of the slide, it says "Video Sponsorship Provided By: JET BRAINS".

At the bottom left of the slide, there is a timestamp: 42:36 / 1:33:02. The overall background of the slide is dark green.

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

# The Rule of Chiel

---

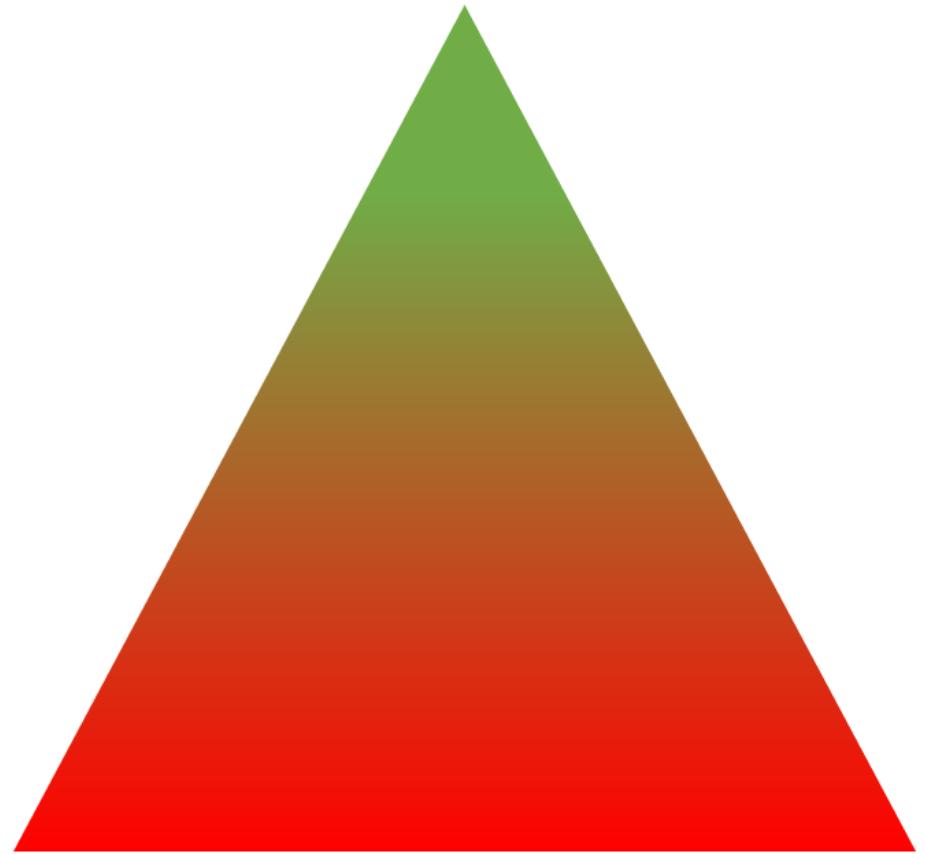
# The Rule of Chiel

---



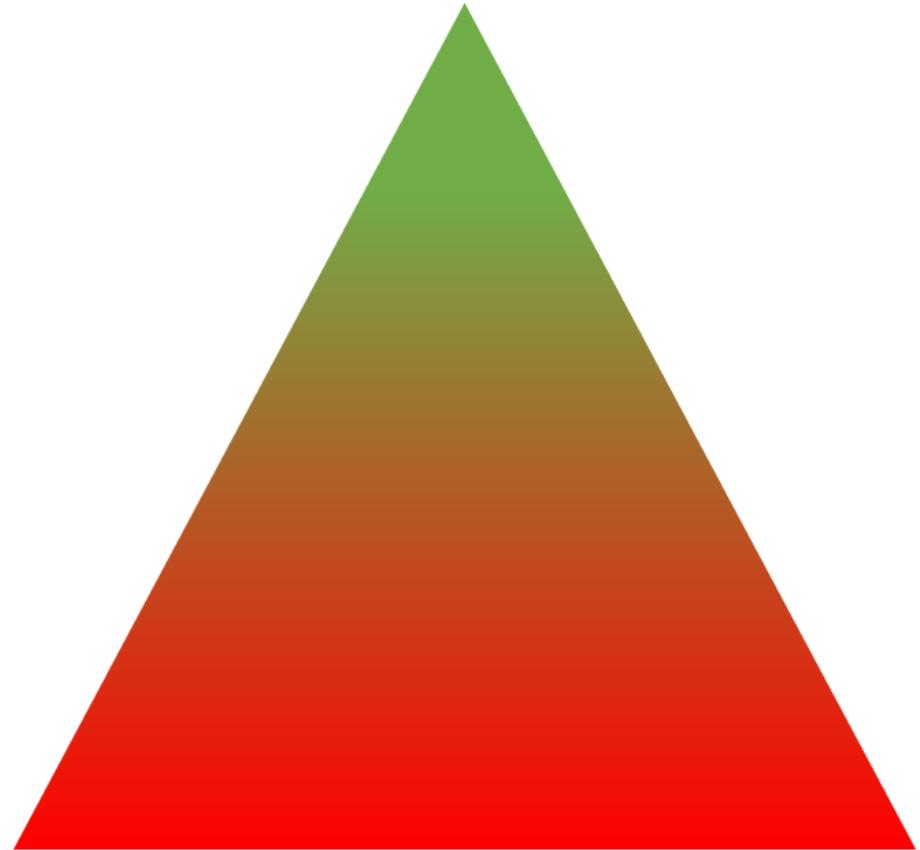
# Cost of operations: The Rule of Chiel

---



# Cost of operations: The Rule of Chiel

---

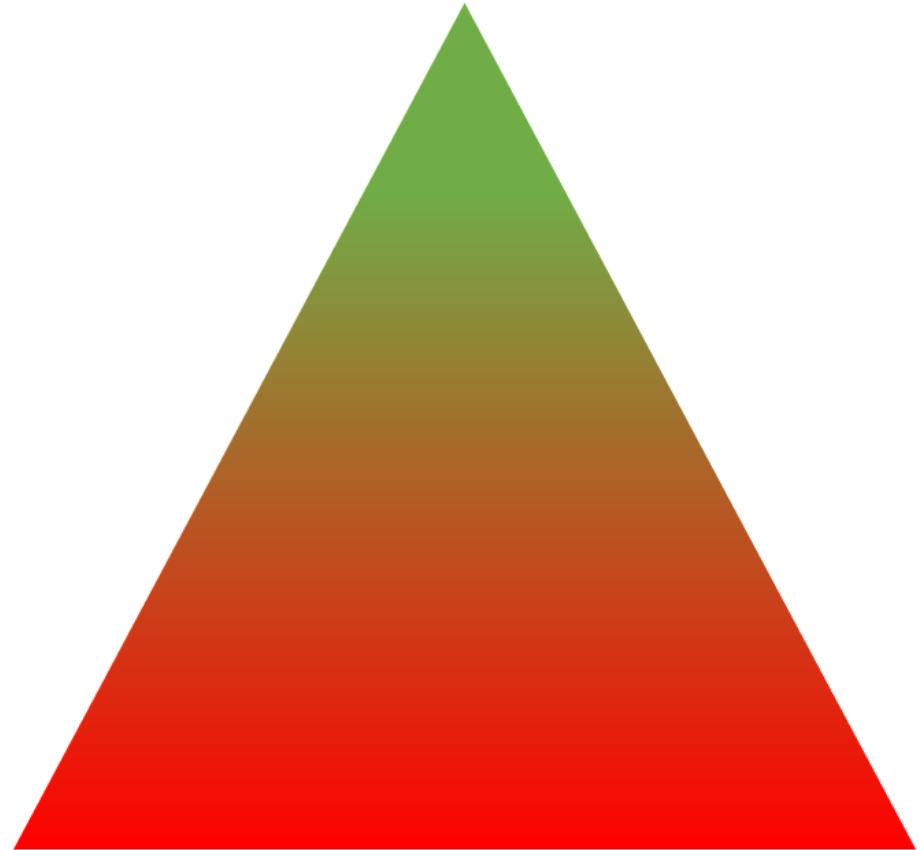


1

Looking up a memoized type

# Cost of operations: The Rule of Chiel

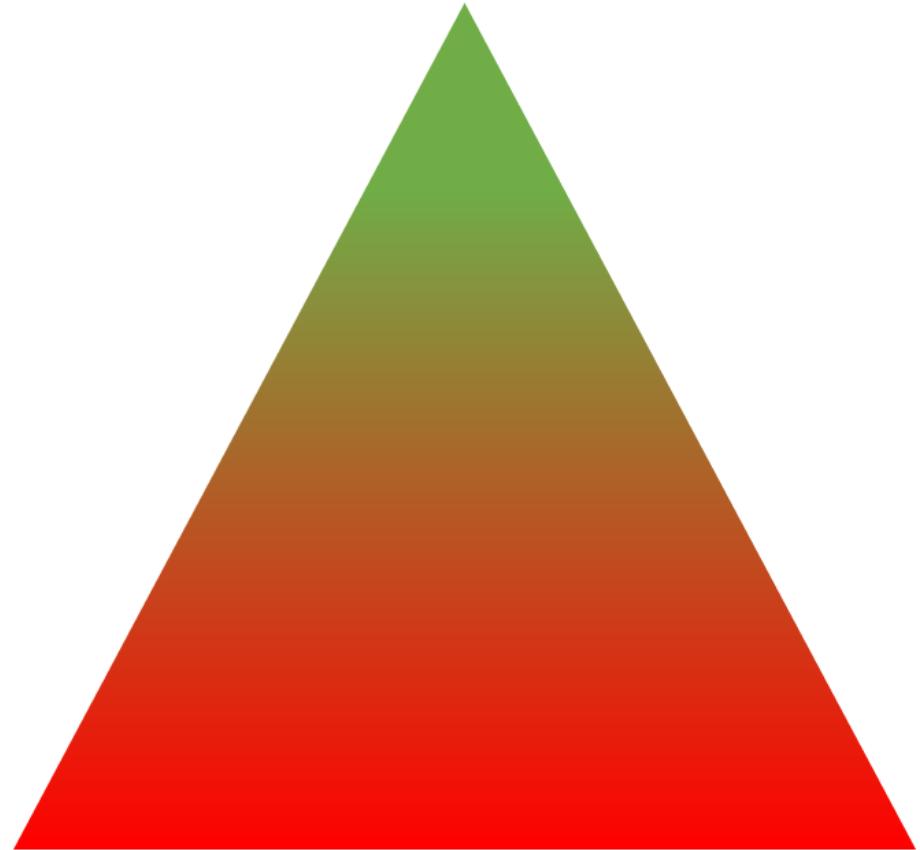
---



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

# Cost of operations: The Rule of Chiel

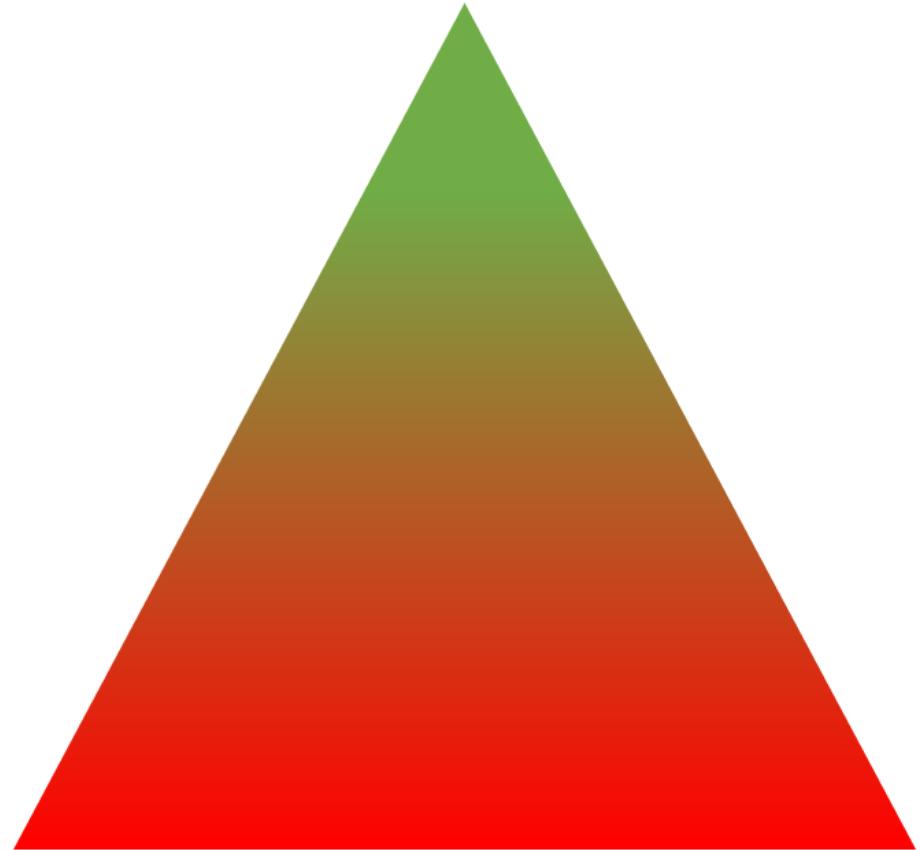
---



- 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

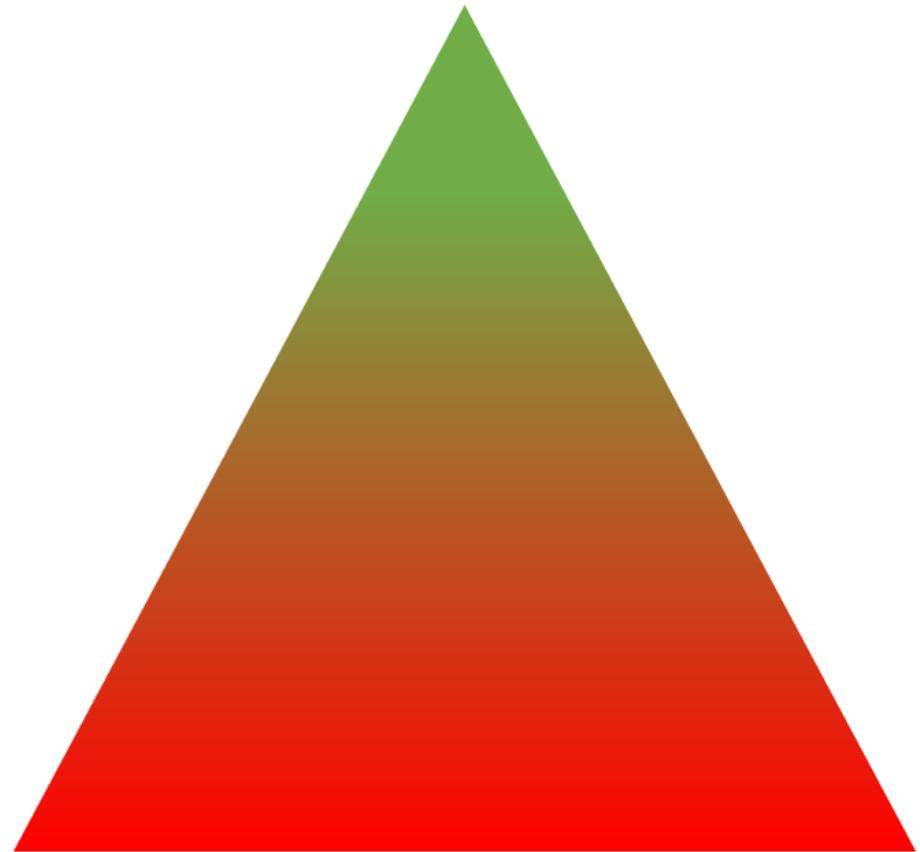
---



- 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

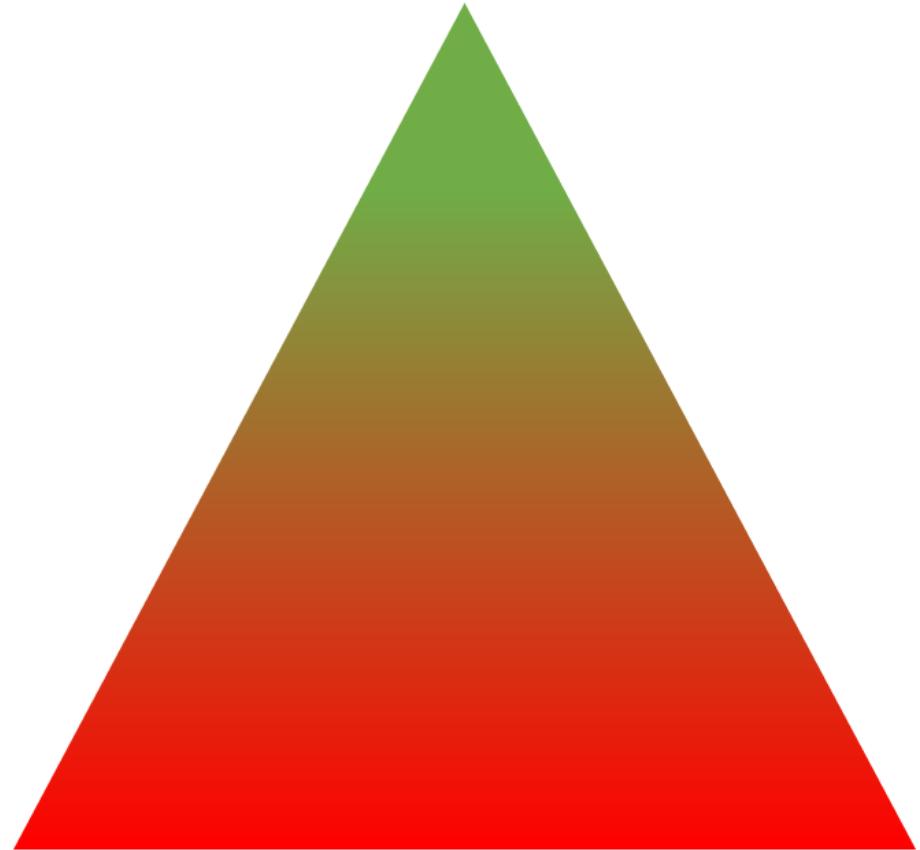
---



- 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

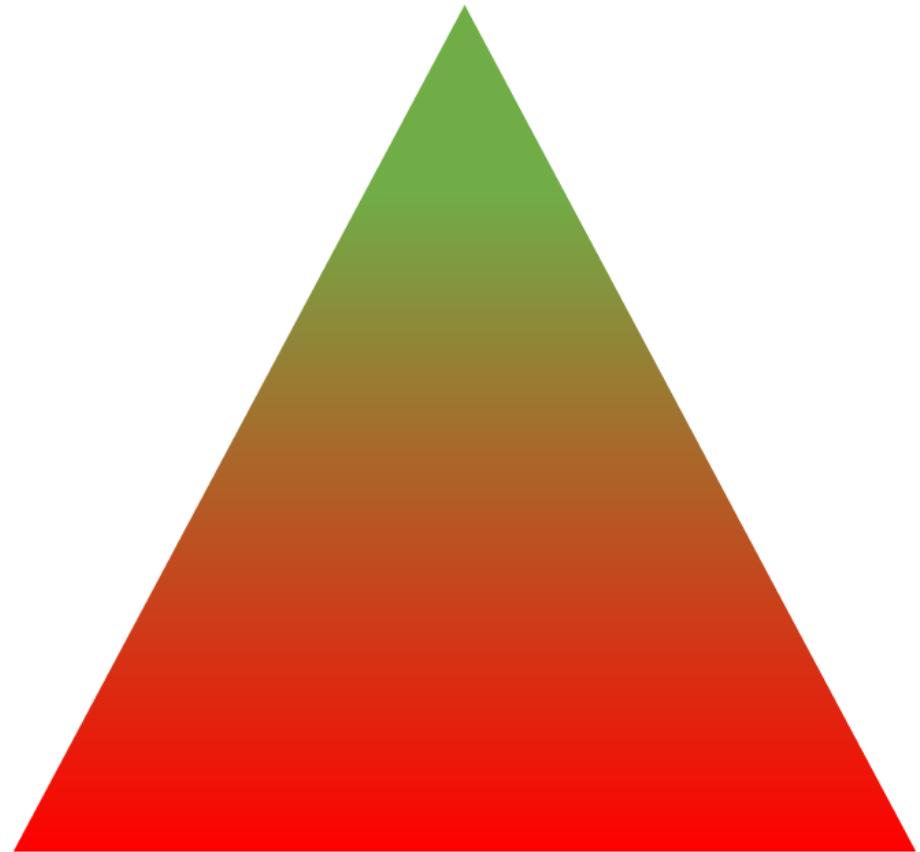
---



- 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;
```

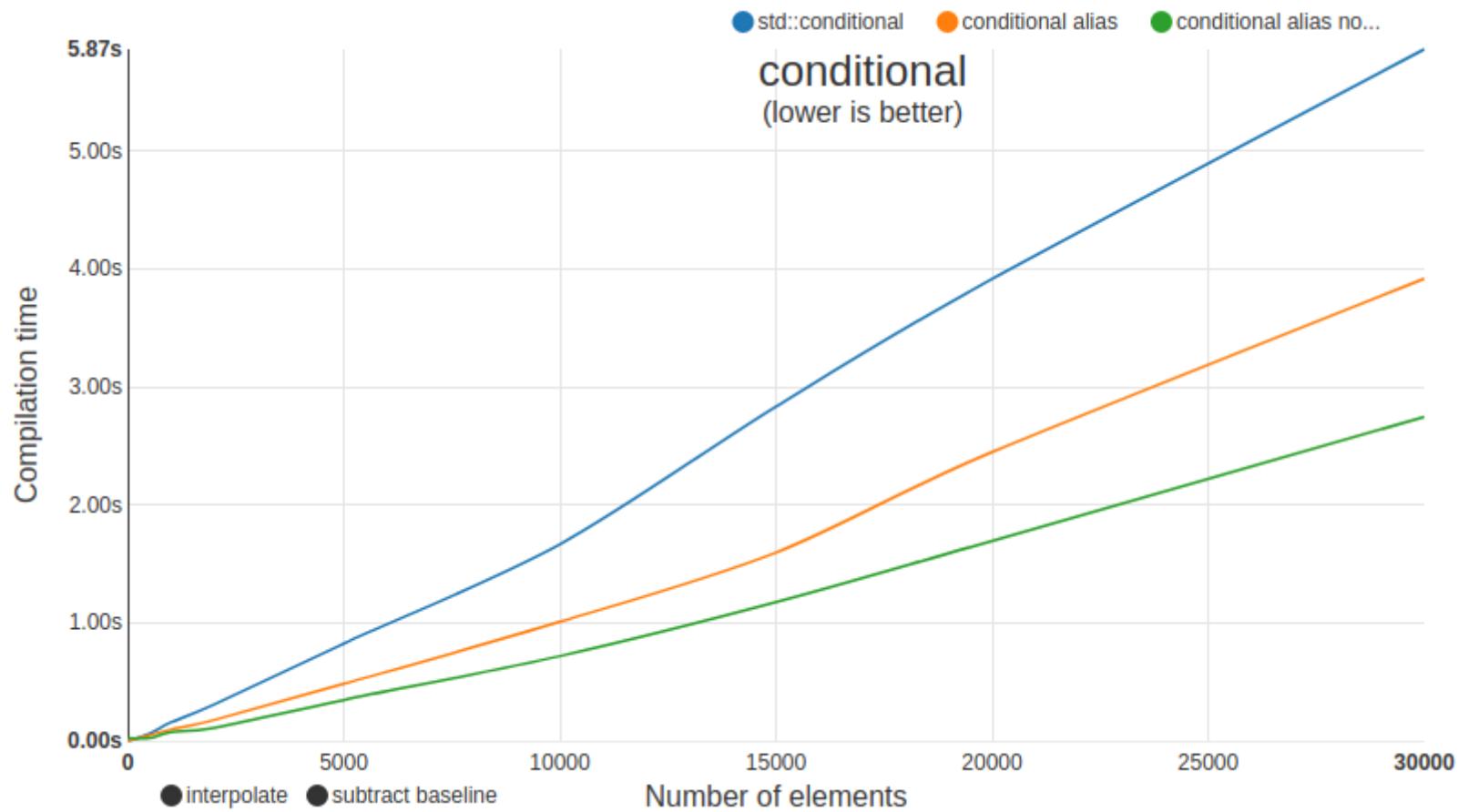
## WITH ALIAS TEMPLATE

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

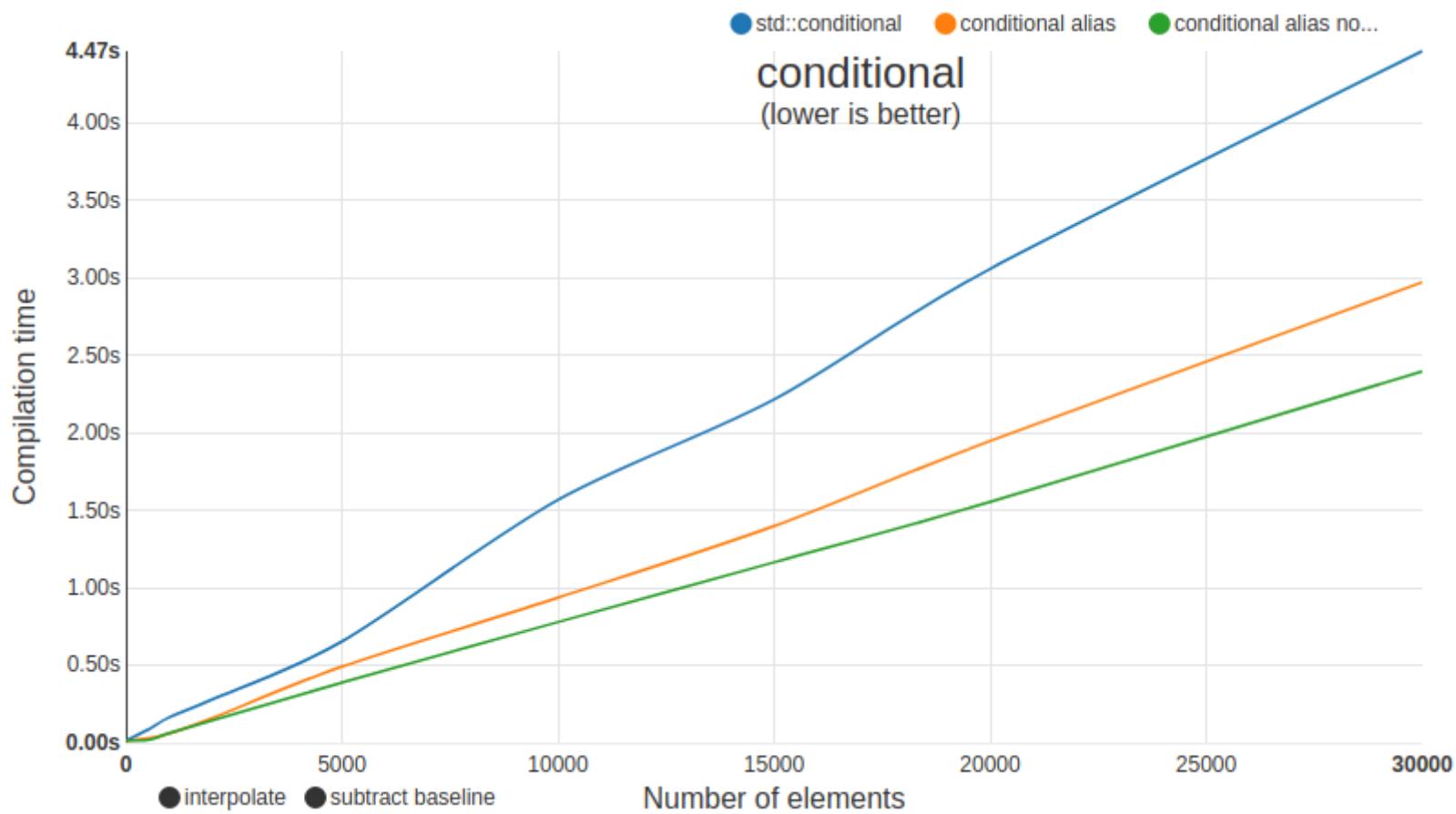
template<>
struct conditional<true> {
    template<typename T, typename F>
    using type = T;
};

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

# conditional performance (gcc-10)



# conditional performance (clang-10)



# Which template is missing in the Rule of Chiel?

---

# Which template is missing in the Rule of Chiel?

---

What about variable templates?

# Which template is missing in the Rule of Chiel?

---

What about variable templates?

- Chiel did not measure the performance of variable templates
- Let's measure it by ourselves...

# 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 {};
```

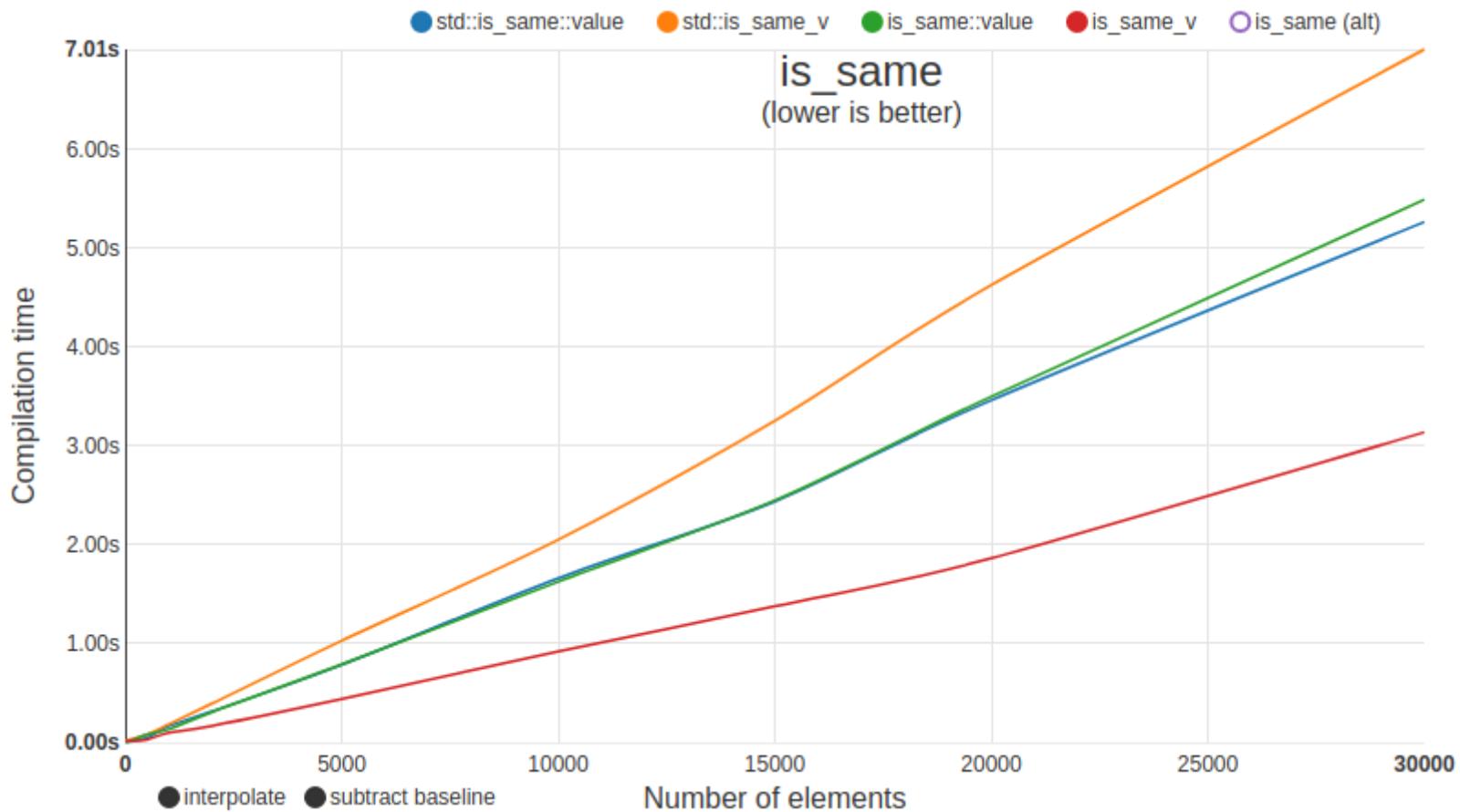
```
template<class T, class U>
inline constexpr bool is_same_v =
    is_same<T, U>::value;
```

## USING VARIABLE TEMPLATES

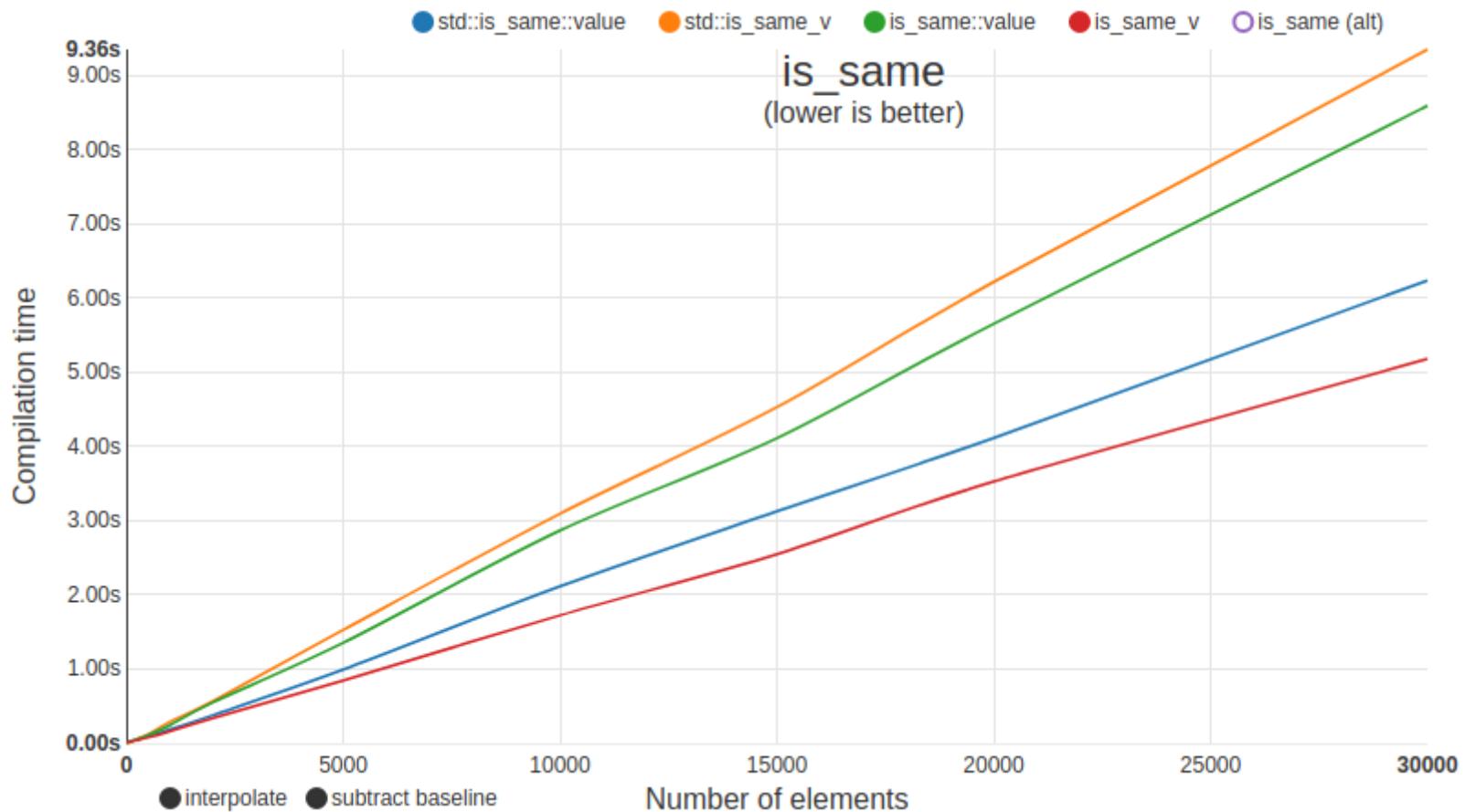
```
template<class T, class U>
inline constexpr bool is_same_v = false;  
  
template<class T>
inline constexpr bool is_same_v<T, T> = true;
```

```
template<class T, class U>
using is_same =
    std::bool_constant<is_same_v<T, U>>;
```

# `is_same` performance (gcc-10)



# `is_same` performance (clang-10)



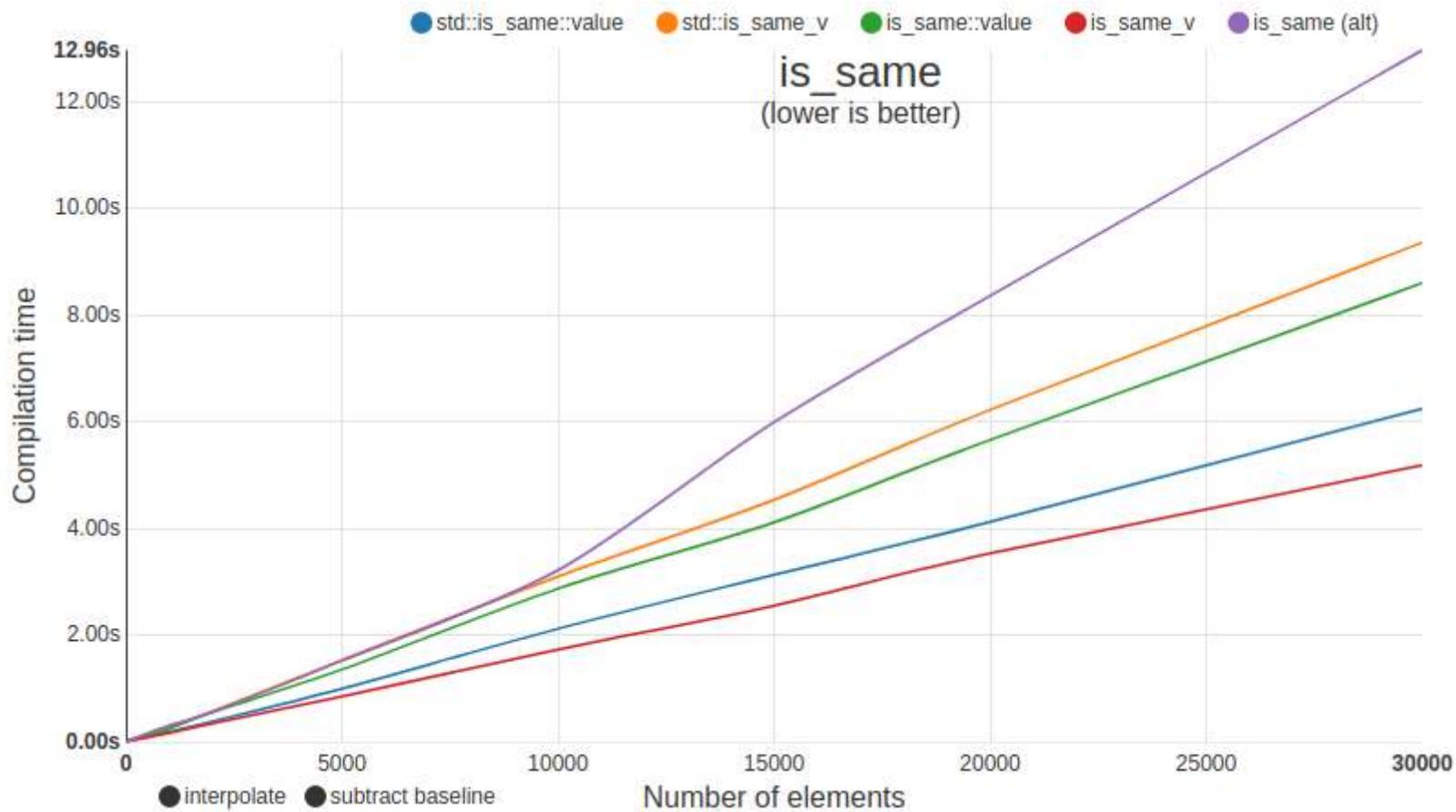
# Alternative `is_same` implementation from CppCon 2019

```
struct WrapperBase {
    static constexpr bool IsSame(void*) { return false; }
};

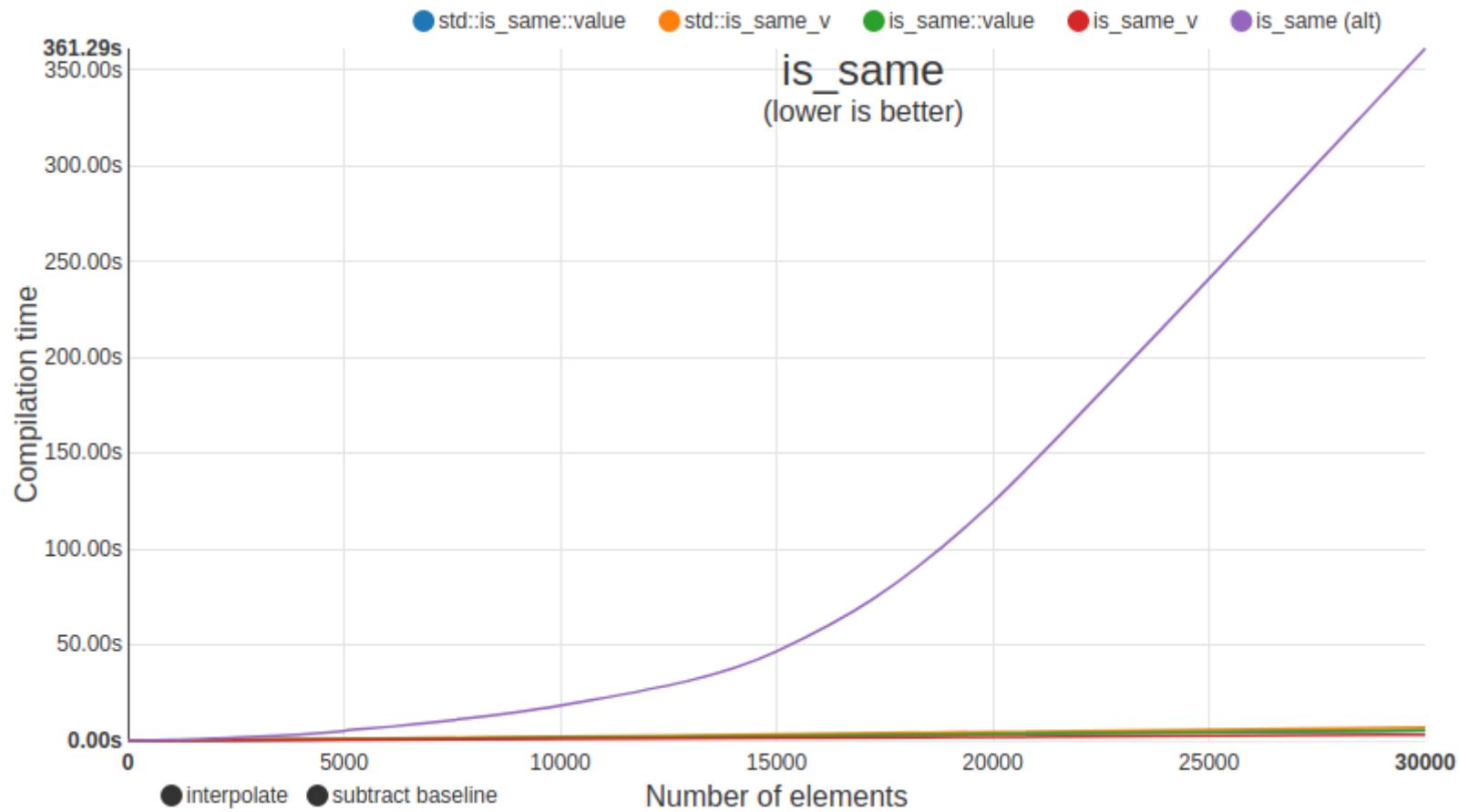
template<typename T>
struct Wrapper : WrapperBase {
    using WrapperBase::IsSame;
    static constexpr bool IsSame(Wrapper<T>*) { return true; }
};

template<class T, class U>
using is_same = std::integral_constant<bool, Wrapper<T>::IsSame((Wrapper<U>*)(nullptr))>;
```

# `is_same` performance (clang-10)



# `is_same` performance (gcc-10)



# What about C++ Concepts?

---

# What about C++ Concepts?

## SFINAE

```
template<long long N>
struct X {
    template<long long M = N>
    typename std::enable_if<M % 2 == 0, int>::type
    static constexpr foo()
    {
        return 0;
    }

    template<long long M = N>
    typename std::enable_if<M % 2 != 0, int>::type
    static constexpr foo()
    {
        return 1;
    }
};
```

## CONCEPTS

```
template<long long N>
struct X {
    static constexpr int foo()
        requires (N % 2 == 0)
    {
        return 0;
    }

    static constexpr int foo()
    {
        return 1;
    };
};
```

# What about C++ Concepts?

## IF CONSTEXPR

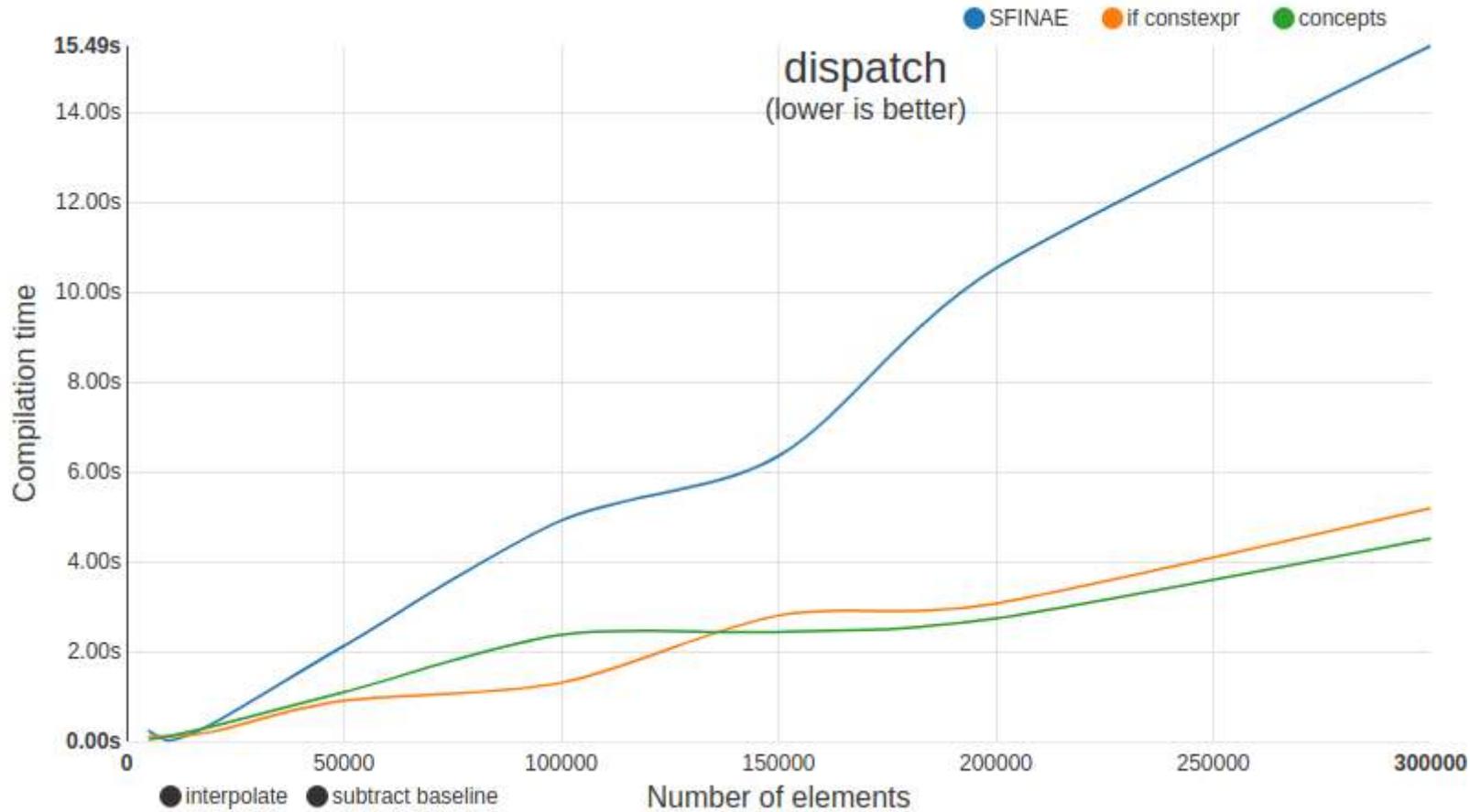
```
template<long long N>
struct X {
    static constexpr int foo()
    {
        if constexpr(N % 2 == 0)
            return 0;
        else
            return 1;
    }
};
```

## CONCEPTS

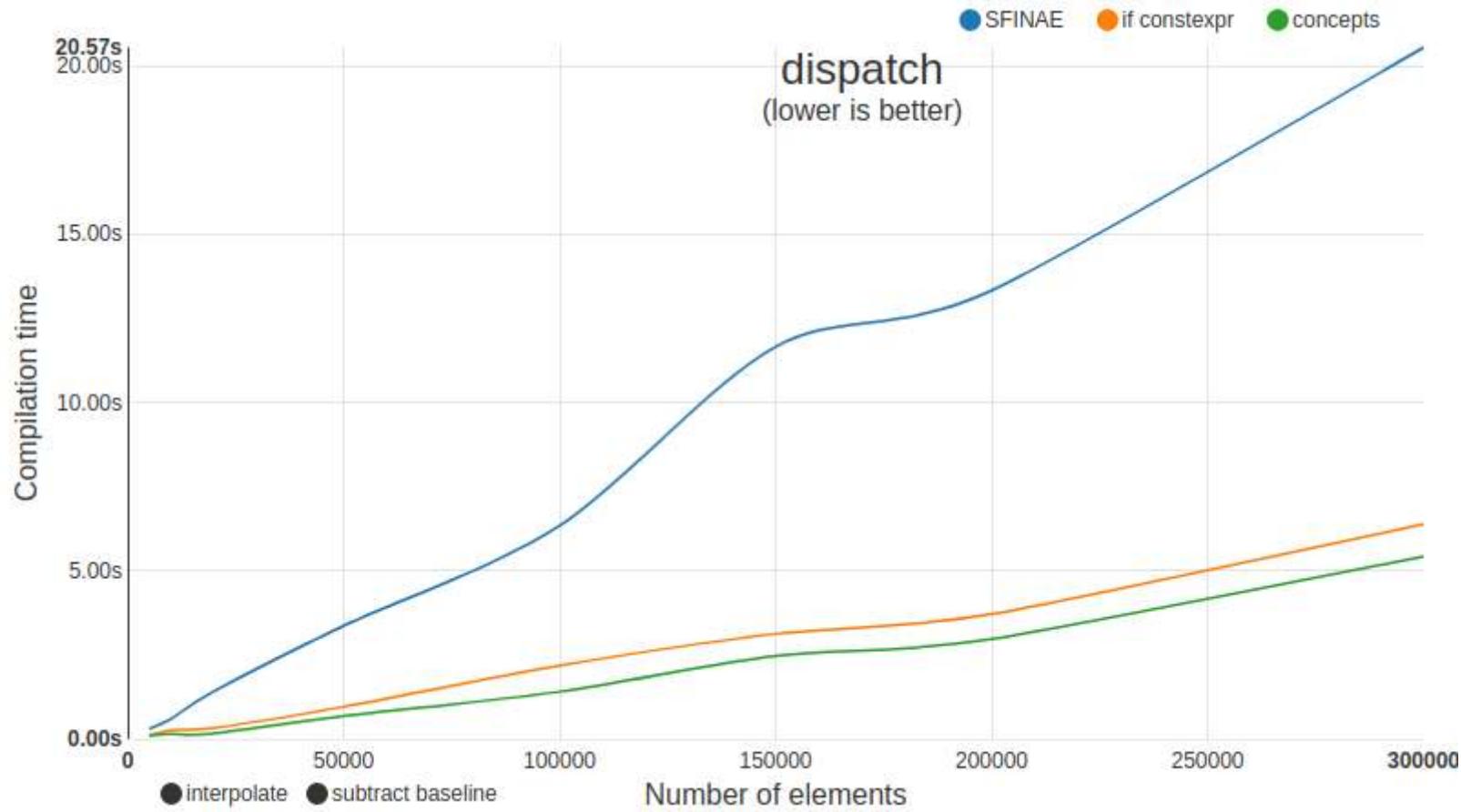
```
template<long long N>
struct X {
    static constexpr int foo()
        requires (N % 2 == 0)
    {
        return 0;
    }

    static constexpr int foo()
    {
        return 1;
    }
};
```

# Dispatch performance (gcc-10)



# Dispatch performance (clang-10)



# Where Concepts are not needed?

- 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::downcast_base_type,
                                                 typename D2::downcast_base_type>::type;
```

# Where Concepts are not needed?

- Compile-time performance impact might be *limited by using concepts only in the user interfaces*

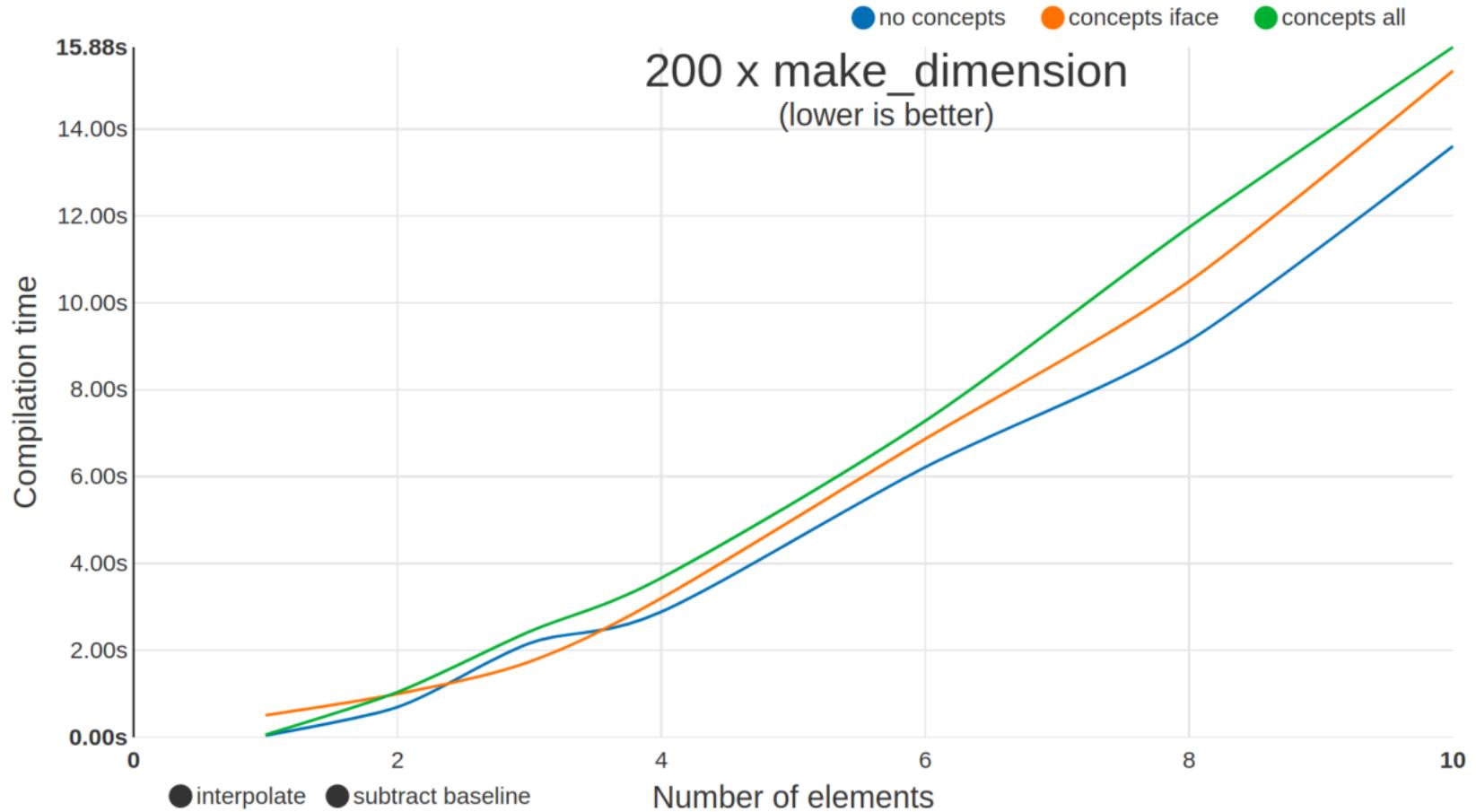
```
template<Exponent... Es>
struct dimension : downcast_base<dimension<Es...>> {};

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

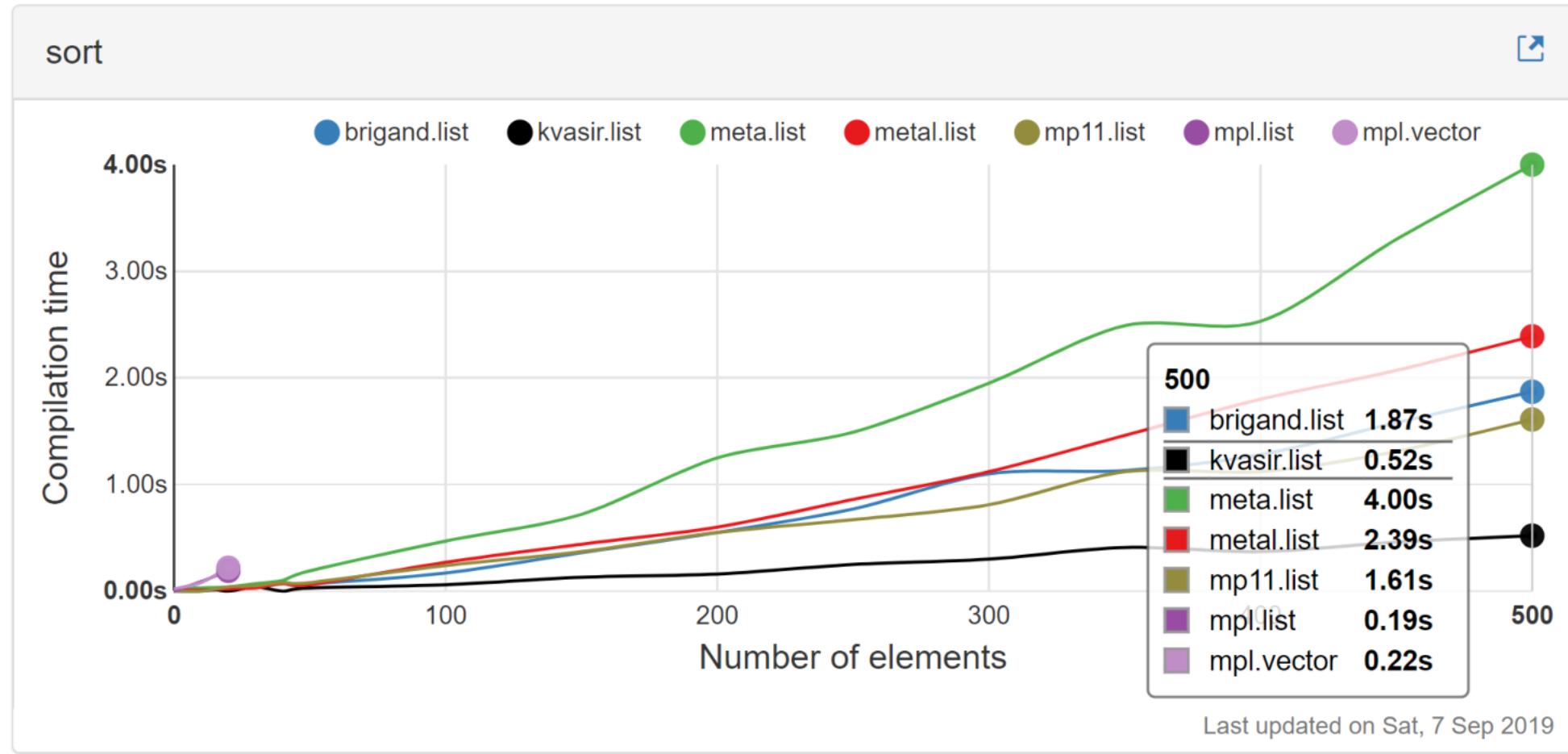
    template<typename... E1, typename... E2>
    struct dimension_divide<dimension<E1...>, dimension<E2...>>
        : dimension_multiply<dimension<E1...>, dimension<exp_invert_t<E2>...>> {
    };
}

template<Dimension D1, Dimension D2>
using dimension_divide_t = detail::dimension_divide<typename D1::downcast_base_type,
                                                    typename D2::downcast_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
- there is more than one way to do it
- **ALWAYS MEASURE** and check different compilers as your milage may vary



**CAUTION**  
**Programming**  
**is addictive**  
**(and too much fun)**