Validation and Conversion of Physical Units at Compile Time

Leveraging the power of the C++11 type system in your simulation model

Dominic Fandrey M.Sc.

32C3 Edition

2015-12-30

Abstract

Unit conversions can be a major source of errors in engineering and software development. Most of the time they just cause delays and frustration. But undiscovered they can cause unexpected behaviour in the field.

This paper illustrates how to realise a type wrapper that performs verification of physical units and generates conversion factors transparently at compile time using the C++11 feature set.

1 Lost in Space

> "The 'root cause' of the loss of the spacecraft was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software, as NASA has previously announced," said Arthur Stephenson, chairman of the Mars Climate Orbiter Mission Failure Investigation Board.

This NASA press release [6] from 1999 followed what is now likely the world's most commonly known[1] unit conversion problem and at USD 125 million it certainly was expensive [16].

2 Requirements

The concepts for this unit verification and conversion system were first implemented in a machining simulation. Machining is a subject matter rife with domain specific units. Even disregarding imperial units, unit conversions are prevalent. E.g. angular velocities are usually provided as revolutions divided by minutes instead of radians divided by seconds and angles are given in degree instead of radians.

Consider the following example from the machining simulation. The figure 1 (p. 2) illustrates the model of the machining tool:

[1] According to a Google search for "unit conversion failures"

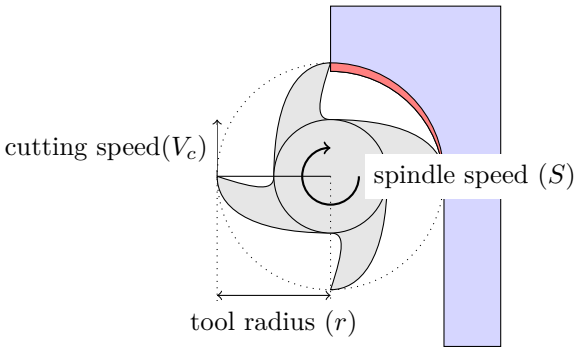

Figure 1: Top view cross section of a milling process (Fandrey [2])

The *diameter* and *cutting speed* are provided parameters, the *spindle speed* is derived from them. Noteworthy conversions that have to take place:

- The *diameter* is commonly provided in millimetre, while the *cutting speed* is provided in $^{\text{metre}}/_{\text{minute}}$.
- The *spindle speed* is an angular velocity, expected to be represented in $^{\text{revolution}}/_{\text{minute}}$ instead of $^{\text{radian}}/_{\text{minute}}$.

The following code listing illustrates the resulting conversions:

Listing 1: Weakly typed code with explicit conversions

```
/**
 * Update the angular velocity of the spindle, according to the desired
 * cutting speed and tool radius.
 *
 * @param speed
 *  The cutting speed in m/min
 * @param radius
 *  The tool radius in mm
 */
void Spindle::setCuttingSpeed(double const speed, double const radius) {
    this->av = speed * 1000 / (2 * PI * radius);
}
```

This example has the following problems:

- Input units must be meticulously documented.
- Input units are chosen to be convenient for the user of the function instead of minimising the conversions.
- There is no way to verify correct use of the interface, e.g. it is syntactically valid to provide a radius in seconds.

Which leads to the following list of requirements:

- Units should be strongly typed.
- The user should be able to provide a value like the radius in any compatible unit like 0.4 in, 394 mil or 10 mm.
- Providing an incompatible unit like $radius = 10\,\text{s}$ should result in compilation failure.

3 C++11

All these problems are solvable. In fact they have been solved before, e.g. as a comprehensive boost library [22]:

> The Boost.Units library is a C++ implementation of dimensional analysis in a general and extensible manner, treating it as a generic compile-time metaprogramming problem. With appropriate compiler optimization, no runtime execution cost is introduced, facilitating the use of this library to provide dimension checking in performance-critical code.

Because unit validation and conversion are compile-time solvable problems, this task is a good way to learn to leverage some of the features introduced by C++11 [9].

`constexpr` The key feature is the **`constexpr`** (Reis et al. [21]) keyword, which can be used to mark statements, functions and constructors as compile time solvable. This makes it possible to use complex types as constants, and perform meta programming arithmetic with arbitrary types, even floating point types like **`float`** or **`double`**.

`static_assert()` Another useful feature is **`static_assert()`** (Klarer et al. [12]), which provides a compile time version of the C assert macro.

user-defined literals A feature that will make the use of units convenient is *user-defined literals* (McIntosh et al. [17]), which allows adding type information to literals, e.g. to turn a character array into a regular expression or big number. It also allows adding type information to floating point or integer types, e.g. to turn a floating point value into metres.

template aliases A very convenient feature of C++11 is *template aliases* (Reis and Stroustrup [20]), which rely on the **`using`** syntax to achieve something akin to a templated **`typedef`**. According to Lavavej [13] using it is a great idea:

> Walter E. Brown (who is unaffiliated with Microsoft) proposed the <type_traits> alias templates in N3546 "TransformationTraits Redux", which is unquestionably the best feature voted into C++14. Anyone who dares to question this assertion is hereby sentenced to five years of hard labor writing typename decay<T>::type instead of decay_t<T>.

variadic templates C++11 introduces *variadic templates*, which are easily explained (Gregor et al. [5]):

> *Variadic templates* provide a way to express function and class templates that take an arbitrary number of arguments.

`auto`, `decltype` Both keywords deal with determining the type returned by an expression (Järvi et al. [11]):

> We suggest extending C++ with the **`decltype`** operator for querying the type of an expression, and allowing the use of keyword **`auto`** to indicate that the compiler should deduce the type of a variable from its initializer expression. …

4 Design

The basic concept was presented at the GoingNative 2012 by Bjarne Stroustrup [23]. The design proposed by Stroustrup was a value container, that took a type as an argument that would represent the dimensions of the base units. User-defined literals were to be used to convert values from different units into the base units.

The idea to present the dimensions of the base units in a type is also the basis for the presented design. Additionally linear conversion factors can be strongly typed. E.g. most factors can be represented as rationals, this has the side effect that recombination of factors does not introduce imprecisions. This approach has its limited by integer boundaries, which can however be circumvented by choosing a domain appropriate base unit or floating point constants, which are the third leg of the presented design.

Their introductions stems from the desire to treat angles as units and encoding π in a rational at sufficient precision would quickly lead to integer overflow when combining types (and thus multiplying linear factors).

The design is simplified by some things that will not be supported:

- Negative linear factors as would be required for certain temperature scales.
- Offsets, which would be required for units that do not have a common 0 point (such as temperatures, again).
- The distinction between absolute and relative values.

Above features were not implemented for the machining simulation, because they were not required. That does not mean they could not be.

The case for temperatures can still be covered by converting to kelvin using user-defined literals. I.e. validation would still work, but compile time linear factor recombination would not work for temperatures.

The distinction between absolute and relative values exists in the aforementioned boost library [22]. In the simulation design this distinction was instead made in the design of vectors, which distinguish between points (absolute) and spatial (relative) vectors.

The listing 2 (p. 5) shows the strongly typed version of listing 1 (p. 2):

Listing 2: Strongly typed code with implicit conversions

```
/**
 * Update the angular velocity of the spindle, according to the desired
 * cutting speed and tool radius.
 *
 * @param speed,radius
 *  The cutting speed and tool radius
 */
void Spindle::setCuttingSpeed(mmpmin const speed, mm const radius) {
    this->av = speed / radius * 1._rad;
}
```

This implementation has the following properties:

- The units of the `speed` and `radius` arguments are documented by the type.
- Input units are chosen to minimise conversions within the function body.
- Using an incompatible unit for `speed` or `radius` causes compilation failure.
- The `speed` and `radius` can be provided in any unit with the same base dimensions and are converted transparently.
- Just dividing the `speed` by the `radius` results in the unit $^{1}/_{\text{minute}}$, which would result in compilation failure because the expected unit is $^{\text{revolution}}/_{\text{minute}}$. Adding the radian aspect changes the type to $^{\text{radian}}/_{\text{minute}}$. The division by 2π as seen in listing 1 (p. 2) happens implicitly when the value is converted from $^{\text{radian}}/_{\text{minute}}$ to $^{\text{revolution}}/_{\text{minute}}$.

example A system of units is defined as follows:

$$n, m \in \mathbb{Z} \tag{1}$$
$$bases \in \{\mathrm{m}^{n_0}, \mathrm{s}^{n_1}, \mathrm{rad}^{n_2}\} \tag{2}$$
$$factor \in \frac{\mathbb{N}}{\mathbb{N}}, \text{ the positive subset of } \mathbb{Q} \tag{3}$$
$$constants \in \{\pi^{m_0}\} \tag{4}$$
$$system \in \{bases, factor, constants\} \tag{5}$$

The base units have the following trivial definitions in that system:

$$metre = \left\{ \left\{ \mathrm{m}^1, \mathrm{s}^0, \mathrm{rad}^0 \right\}, \frac{1}{1}, \left\{ \pi^0 \right\} \right\} \tag{6}$$
$$second = \left\{ \left\{ \mathrm{m}^0, \mathrm{s}^1, \mathrm{rad}^0 \right\}, \frac{1}{1}, \left\{ \pi^0 \right\} \right\} \tag{7}$$
$$rad = \left\{ \left\{ \mathrm{m}^0, \mathrm{s}^0, \mathrm{rad}^1 \right\}, \frac{1}{1}, \left\{ \pi^0 \right\} \right\} \tag{8}$$

The following units can be converted back to the base units:

$$millimetre = \left\{ \left\{ \mathrm{m}^1, \mathrm{s}^0, \mathrm{rad}^0 \right\}, \frac{1}{1000}, \left\{ \pi^0 \right\} \right\} \tag{9}$$

$$minute = \left\{ \left\{ \mathrm{m}^0, \mathrm{s}^1, \mathrm{rad}^0 \right\}, \frac{60}{1}, \left\{ \pi^0 \right\} \right\} \tag{10}$$

$$revolution = \left\{ \left\{ \mathrm{m}^0, \mathrm{s}^0, \mathrm{rad}^1 \right\}, \frac{2}{1}, \left\{ \pi^1 \right\} \right\} \tag{11}$$

All units with a set of *bases* can be cast to all other units with the same set of *bases*. This happens implicitly for all operations where units must be identical, i.e. assignment, addition, subtraction and modulo.

This kind of implicit cast introduces a runtime cost, which cannot be avoided. However the cost can be minimised to a single multiplication per cast. The factor is determined at compile time.

In the following example of addition the right hand side type needs to be converted to the left hand side type:

$$\text{expr: } x_0 \,\mathrm{rad} + x_1 \,\mathrm{revolution} \tag{12}$$

$$\implies \text{expr: } (x_0 + x_1 \cdot c) \,\mathrm{rad} \tag{13}$$

$$c = \left| \frac{\mathrm{revolution}}{\mathrm{rad}} \right| \tag{14}$$

$$\implies c = \left| \frac{\left\{ \left\{ \mathrm{m}^0, \mathrm{s}^0, \mathrm{rad}^1 \right\}, \frac{2}{1}, \left\{ \pi^1 \right\} \right\}}{\left\{ \left\{ \mathrm{m}^0, \mathrm{s}^0, \mathrm{rad}^1 \right\}, \frac{1}{1}, \left\{ \pi^0 \right\} \right\}} \right| \tag{15}$$

$$\implies c = \frac{\frac{2}{1} \cdot \pi^1}{\frac{1}{1} \cdot \pi^0} = 2\pi \tag{16}$$

$$\implies \text{expr: } (x_0 + x_1 \cdot 2\pi) \,\mathrm{rad} \tag{17}$$

If x_1 is a constant expression (e.g. a literal value), it can be substituted with the converted value at compile time.

If x_0 and x_1 are constant expressions, the entire expression can be substituted at compile time.

Multiplication and division have no additional runtime cost. The following expression shows how the type for an angular acceleration is calculated without the need to interact with the runtime payload:

$$\text{expr: } \frac{10000\,\text{revolution}}{1\,\text{minute}\cdot 2\,\text{second}} \tag{18}$$

$$\implies \text{expr: } \frac{10000\,\text{revolution}}{(1\cdot 2)\,(\text{minute}\cdot\text{second})} \tag{19}$$

$$\implies \text{expr: } \frac{10000\,\text{revolution}}{(1\cdot 2)\left(\begin{array}{c}\left\{\left\{\mathrm{m}^0,\mathrm{s}^1,\mathrm{rad}^0\right\},\frac{60}{1},\left\{\pi^0\right\}\right\}\\ \cdot\left\{\left\{\mathrm{m}^0,\mathrm{s}^1,\mathrm{rad}^0\right\},\frac{1}{1},\left\{\pi^0\right\}\right\}\end{array}\right)} \tag{20}$$

$$\implies \text{expr: } \frac{10000\,\text{revolution}}{(1\cdot 2)\left\{\left\{\mathrm{m}^0,\mathrm{s}^2,\mathrm{rad}^0\right\},\frac{60}{1},\left\{\pi^0\right\}\right\}} \tag{21}$$

$$\implies \text{expr: } \frac{10000\left\{\left\{\mathrm{m}^0,\mathrm{s}^0,\mathrm{rad}^1\right\},\frac{2}{1},\left\{\pi^1\right\}\right\}}{(1\cdot 2)\left\{\left\{\mathrm{m}^0,\mathrm{s}^2,\mathrm{rad}^0\right\},\frac{60}{1},\left\{\pi^0\right\}\right\}} \tag{22}$$

$$\implies \text{expr: } \left(\frac{10000}{1\cdot 2}\right)\left\{\left\{\mathrm{m}^0,\mathrm{s}^{-2},\mathrm{rad}^1\right\},\frac{2}{60},\left\{\pi^1\right\}\right\} \tag{23}$$

Note how separating the conversion factor from the dimension of the base units makes it possible to mix different derivations of base units, e.g. minute with second.

5 Glossary

instance In the scope of this paper an instance of a template is a type.

meta-function A meta-programming struct/class template that acts as a function, by taking a meta-type as an argument and returning something through a member (e.g. `value` or `type`).

meta-type A meta-programming struct/class template that carries information without a runtime payload.

The distinction between meta-functions and types is a semantic distinction, not a syntactical one.

recursion See Fandrey [3].

6 A Rational Number Meta-Type

A positive rational number meta-type is of limited complexity, so it is a good point to start:

Listing 3: A primitive rational number type

```
namespace units {

typedef unsigned long long uint;

template <uint Numerator, uint Denominator>
struct rational {
    static constexpr uint const numerator{Numerator};
    static constexpr uint const denominator{Denominator};
};

} /* namespace units */
```

operators To be useful the meta-type requires operators, `rational<>` is supposed to represent a linear factor. When two unit values are multiplied or divided, so should the linear factors of the types:

Listing 4: Meta-operators for the rational type

```
namespace units {

...

template <class Op>
struct rational_unary {
    typedef rational<Op::denominator, Op::numerator> invert;
};

template <class Op>
using rational_invert_t = typename rational_unary<Op>::invert;

template <class Lhs, class Rhs>
struct rational_binary {
    typedef rational<Lhs::numerator * Rhs::numerator,
                     Lhs::denominator * Rhs::denominator> mul;
};

template <class Lhs, class Rhs>
using rational_mul_t = typename rational_binary<Lhs, Rhs>::mul;

template <class Lhs, class Rhs>
using rational_div_t = rational_mul_t<Lhs, rational_invert_t<Rhs>>;

} /* namespace units */
```

This code works in principle, however it has some problems. Namely that both `numerator` and `denominator` grow with every multiplication and may overflow. The resulting (grossly) wrong factor would be mission fatal (the mission being to ensure correctness).

In 1996 the European Space Agency learned the hard way what may happen if integer overflow remains undetected [15]. James Gleick [4] summarised the event in the New York Times Magazine:

> It took the European Space Agency 10 years and $7 billion to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe

> overwhelming supremacy in the commercial space business.
>
> All it took to explode that rocket less than a minute into its maiden voyage last June, scattering fiery rubble across the mangrove swamps of French Guiana, was a small computer program trying to stuff a 64-bit number into a 16-bit space.

So the first step is to make sure compilation fails to enable the library user to avoid this case:

Listing 5: Prevent compilation on overflow

```
...

template <class Lhs, class Rhs>
struct rational_mul {

    static_assert((Lhs::numerator * Rhs::numerator) / Rhs::numerator ==
                      Lhs::numerator,
                  "Overflow in rational numerator multiplication");

    static_assert((Lhs::denominator * Rhs::denominator) / Rhs::denominator ==
                      Lhs::denominator,
                  "Overflow in rational denominator multiplication");

    typedef rational<Lhs::numerator * Rhs::numerator,
                     Lhs::denominator * Rhs::denominator> type;
};

template <class Lhs, class Rhs>
using rational_mul_t = typename rational_mul<Lhs, Rhs>::type;

...
```

Take note that the binary operator template was renamed to `rational_mul<>`, this was done because the **`static_assert`**`()`s are specific to multiplication, which means future binary operators should not be grouped together in a single template with multiplication.

minimising This version of the operator will at least not stab its users in the back, but frequently working around overflow, even when the base units are well-chosen, would be a great burden on library users nonetheless.

So rationals should transparently provide their minimal representation. To minimise the representation of a rational the greatest common divider of the numerator and the denominator is required.

A simple way of acquiring a GCD is Euclid's (improved) algorithm:

$$n, i \in \mathbb{Z} \tag{24}$$

$$f(n, i) = n, \qquad \text{for } i = 0 \tag{25}$$

$$f(n, i) = f(i, n \bmod i), \qquad \text{for } i \neq 0 \tag{26}$$

The implementation of the algorithm belongs in front of `rational<>`:

Listing 6: Euclid's algorithm

```
template <uint Lhs, uint Rhs>
struct euclid : euclid<Rhs, Lhs % Rhs> {};

template <uint Gcd>
struct euclid<Gcd, 0> {
    static constexpr uint const value{Gcd};
};
```

In this case the recursion is realised through inheritance (it could also be done using a member). The termination of the recursion is a specialisation of the template that provides the final value.

Alternatively a **constexpr** function can be used. In C++11 a **constexpr** function has to consist of a single return statement. C++14 relaxes these requirements significantly, allowing the creation of local variables, the use of `assert()` and loops. Non of which are required to implement Euclid's algorithm:

Listing 7: Euclid's algorithm

```
constexpr uint euclid(uint const a, uint const b) {
    return b ? euclid(b, a % b) : a;
}
```

Now the rational template needs to use it:

Listing 8: A self-minimising rational number type

```
template <uint Numerator, uint Denominator>
struct rational {
    static constexpr uint const numerator{
        Numerator / euclid(Numerator, Denominator)};
    static constexpr uint const denominator{
        Denominator / euclid(Numerator, Denominator)};
    typedef rational<numerator, denominator> type;
};
```

Note the new member `type`, it has a special property. Discarding technical limitations rationals have an infinite amount of representations for the same value. That means there is an infinite amount of `rational<>` instances, that all represent the same value. Due to minimising the type traits however, all `rational<>` instances representing the same value have the same type member. E.g. **typename** `rational<1, 2>::type`, **typename** `rational<3, 6>::type`, **typename** `rational<119, 238>::type` all return the same type.

value retrieval What remains to be achieved is a way to retrieve a floating point value from the rational. Several options present themselves, the list is not exhaustive:

- Use the `numerator` and `denominator` members directly.
- Create a meta-function that takes a target type and a `rational<>` instance.
- Add a conversion operator to `rational<>`.
- Add a *variable template* to `rational<>`.

A *variable template* [19] is very convenient and consistent with the established coding style:

Listing 9: Convert a rational value using a variable template

```
template <uint Numerator, uint Denominator>
struct rational {
    ...
    template <typename T>
    static constexpr T const value{T{numerator} / T{denominator}};
};
// Use: rational<1, 2>::value<double>
```

Unfortunately *variable templates* are a C++14 core feature, that is not yet universally available. Notably, the C++ feature table for Visual Studio 2015 [14] lists them as unsupported.

So if portability is a concern, an alternative approach must be chosen. Accessing numerator and denominator directly works out of the box, but is cumbersome. A meta-function is a viable option:

Listing 10: Convert a rational value using a meta-function

```
template <typename T, class Op>
struct rational_val {
    static constexpr T const value{T{Op::numerator} / T{Op::denominator}};
};
// Use: rational_val<double, rational<1, 2>>::value
```

The advantage of this approach is that its implementation is very straightforward and consistent. The disadvantage is that using it is unwieldy and the order of operands is not necessarily intuitive.

An alternative is to add a conversion operator:

Listing 11: Convert a rational value using a conversion operator

```
template <uint Numerator, uint Denominator>
struct rational {
    ...
    template <typename T>
    constexpr operator T() const { return T{numerator} / T{denominator}; }
};
// Use: double(rational<1, 2>{})
```

This version is short and the code is easy enough to read. The one disadvantage is that the `rational<>` needs to be constructed to use the conversion operator, which is ugly but does not add runtime cost.

The remaining code examples will assume that the code from listing 11 was used.

7 Testing

Especially in meta-programming, where debugging can be extremely annoying, it is important to test early. It is good practice to add some **`static_assert()`**s after every template definition. When the code has reached a stable state those tests can be scooped up into a unit test.

Meta-function testing should be done from the bottom up, so that the first test that fails is close to the problem.

Thus, the first thing to test is the `euclid()` function:

Listing 12: Test Euclid's algorithm

```
static_assert(euclid(3, 9) == 3,
              "Euclid's algorithm should return the GCD");

static_assert(euclid(7, 13) == 1,
              "Euclid's algorithm should return the GCD");

static_assert(euclid(21, 91) == 7,
              "Euclid's algorithm should return the GCD");
```

debugging If the test compiles everything is fine. If it doesn't it helps to look at the values. To make the compiler tell an integer value, it is necessary to create illegal code that causes a compilation error with a type constructed from the value:

Listing 13: Retrieve integers from meta-functions

```
template <int... Values>
struct ints;

static_assert(ints<euclid(21, 91)>::foo, "bar");
```

The listing 13 leverages an easy way to trigger an error message involving a certain type. Access to a declared but undefined type.

While the verbose print shows the code as written, the actual error message shows the computed type `ints<7>` for `euclid(21, 91)`:

Listing 14: The compiler output for listing 13

```
c++ -std=c++11 -Wall -Werror -pedantic -o bin/tests/units_Units.o -c src/tests/
    units_Units.cpp
src/tests/units_Units.cpp:26:15: error: implicit instantiation of undefined
    template 'ints<7>'
static_assert(ints<euclid(21, 91)>::foo, "bar");
              ^
src/tests/units_Units.cpp:24:8: note: template is declared here
struct ints;
       ^
1 error generated.
```

Your mileage may vary depending on the compiler. The output in listing 14 was produced by *Clang 3.4.1*.

`is_same<>` Before going on to testing the `rational<>` type, some common meta-programming techniques should be established. The following are available in the `std` namespace from the `<type_traits>` header.

Listing 15: `std::is_same<>` possible implementation

```
// In <type_traits>, since C++11
template <class A, class B>
struct is_same : std::false_type {};

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

The `is_same<>` meta-function is simple. The default is for it to return false (`false_type` and `true_type` have a boolean member `value`). Only if the specialised template is selected, will `is_same<>` return true.

The goal of using `is_same<>` is to compare rationals. But so far it is of limited use:

Listing 16: `std::is_same<>` to test rational

```
static_assert(!is_same<rational<1, 3>, rational<4, 12>>::value,
              "The rationals 1/3 and 4/12 are different types");

static_assert(rational<4, 12>::numerator == 1,
              "The minimised numerator of 4/12 should be 1");

static_assert(rational<4, 12>::denominator == 3,
              "The minimised denominator of 4/12 should be 3");
```

`is_same_rational<>` The obvious test at this point is whether $\frac{4}{12}$ and $\frac{1}{3}$ represent the same value:

Listing 17: `is_same_rational<>` desired use

```
static_assert(is_same_rational<rational<1, 3>, rational<4, 12>>::value,
              "The rationals 1/3 and 4/12 represent the same value");

static_assert(!is_same_rational<rational<1, 4>, rational<4, 12>>::value,
              "The rationals 1/4 and 4/12 do not represent the same value");

static_assert(is_same_rational<rational<1, 4>, rational<3, 12>>::value,
              "The rationals 1/4 and 3/12 represent the same value");

static_assert(!is_same_rational<double, rational<4, 12>>::value,
              "Double is not a rational");
```

To do that the `type` member of `rational<>` defined in listing 8 (p. 10) can be used:

Listing 18: `is_same_rational<>` implementation

```
template <class, class>
struct is_same_rational : std::false_type {};

template <uint LNum, uint LDenom, uint RNum, uint RDenom>
struct is_same_rational<rational<LNum, LDenom>, rational<RNum, RDenom>>
    : is_same<typename rational<LNum, LDenom>::type,
              typename rational<RNum, RDenom>::type> {};
```

This works because the `type` member is created from the minimised numerator and denominator.

SFINAE Evidently, using specialisation to match a certain template can become cumbersome, especially if the template has many arguments, which are not even required in the body. An alternative is to use a so called SFINAE construct.

SFINAE stands for *Substitution Failure Is Not An Error* and has already been part of C++98 [7], which was the first ISO standard version of C++ .

It refers to an aspect of template selection. Basically a C++ compiler is also an interpreter. Regular code is compiled, but first templates are interpreted to derive code from them, which in turn can be compiled.

As a result C++ templates constitute a functional programming language, which permits programming with types and integral values (Veldhuizen [25]):

> The introduction of templates to C++ added a facility whereby the compiler can act as an interpreter. This makes it possible to write programs in a subset of C++ which are interpreted at compile time.

This was an accidental development and its discovery is commonly attributed to Erwin Unruh [24].

C++ selects the most specialised template to create a type from. If any of the arguments of the template are ill-formed, it selects a less specialised version of the template. This is the part of the process called SFINAE and it only works in the template selection stage, that means SFINAE has to be triggered in the template signature. Errors in the body of a template do not affect template selection and will thus lead to compilation failure.

`void_t<>` A most instructive example of SFINAE is a meta-programming technique called `void_t<>`:

Listing 19: `std::void_t<>` possible implementation

```
// Will most likely be in <type_traits> with C++17
template <class...> using void_t = void;
```

Walter E. Brown [1] explains it, in his standard proposal:

> Given a template argument list consisting of any number of well-formed types, the alias will thus always name **`void`**. However, if even a single template argument is ill-formed, the entire alias will itself be ill-formed. As demonstrated above and in our earlier papers, this becomes usefully detectable, and hence exploitable, in any SFINAE context.

In combination with **`decltype`** it is possible to inspect the members and runtime capabilities of argument types at compile time. The following implementation of an `is_rational<>` test checks a given operand for providing the expected type traits:

Listing 20: `is_rational<>` test

```
template <class Op, class = void>
is_rational : std::false_type {};

template <class Op>
is_rational<Op, void_t<decltype(uint(Op::numerator)), decltype(uint(Op::
    denominator))>>
    : is_same<Op, rational<Op::numerator, Op::denominator>> {};
```

The case specialised with `void_t<>` is well-formed if `Op` has the `numerator` and `denominator` members and both of them can be used to construct an unsigned integer.

This way of testing for type traits is good to accept user provided compatible types. If this kind of extensibility is undesired, template specialisation can be used to enforce use of the desired template:

Listing 21: `is_rational<>` test simplified

```
template <class>
is_rational : std::false_type {};

template <uint numerator, uint denominator>
is_rational<rational<numerator, denominator>> : true_type {};
```

`std::enable_if<>` C++11 introduced the `<type_traits>` header which provides many convenient SFINAE constructs. A construct that can be useful in this case is `std::enable_if<>`:

Listing 22: `std::enable_if<>` possible implementation

```
// In <type_traits>, since C++11
template <bool Cond, typename T = void>
struct enable_if {};

template <typename T>
struct enable_if<true, T> { typedef T type; };

// In <type_traits>, since C++14
template <bool Cond, typename T = void>
using enable_if_t = typename enable_if<Cond, T>::type;
```

The `enable_if<>` template can be used to create code, which is ill-formed if a boolean expression returns false. This can be used to trigger SFINAE, which is demonstrate by the following alternative implementation of `is_same_rational<>`:

Listing 23: `is_same_rational<>` implementation using `enable_if<>`

```
template <class Lhs, class Rhs, class = void>
struct is_same_rational : false_type {};

template <class Lhs, class Rhs>
struct is_same_rational<
    Lhs, Rhs, enable_if_t<is_rational<Lhs>::value && is_rational<Rhs>::value>>
    : is_same<typename Lhs::type, typename Rhs::type> {};
```

Both the implementation from listing 18 (p. 13) and 23 make the tests from

listing 17 (p. 13) work as expected.

8 Dimensions

The core feature of a unit system is validation. I.e. the system should enforce that the value provided has the expected dimensions. Only then should conversion to the expected unit be performed.

This check however is agnostic about the actual base unit used. In fact the whole implementation is base unit agnostic, e.g. metres can be defined based on inches or inches based on metre. Either way the compiler would emit the same code. This is due to the fact that both definitions are equivalent:

$$1\,\mathrm{in} = \frac{254}{10000}\,\mathrm{m} \tag{27}$$

$$1\,\mathrm{m} = \frac{10000}{254}\,\mathrm{in} \tag{28}$$

As a result they produce the same conversion factors between units[2].

exponents So to validate the dimensions of base units it suffices to think about them in abstract terms, i.e. length, mass, time instead of metre, kilogram, second. Each dimension only needs to be represented by its exponents:

Listing 24: The desired interface for a list of exponents

```
typedef long long sint;

template <sint... Exponents>
struct exponents {};
```

This format is convenient to enter, but very cumbersome to work with. When converting units, both units must have the same exponents, which is trivial to implement.

But when multiplying or dividing units, a derived list of exponents composed of pairwise addition or subtraction has to be created[3]. So instead a chained container type can be used:

Listing 25: The implementation for a list of exponents

```
typedef long long sint;

template <sint Exponent, class Tail = void>
struct exponents_chain {
    typedef Tail tail;
    static constexpr sint const value{Exponent};
};
```

This solution is easy to recurse through, the **`void`** type is used to mark the end of the chain. As a side effect all operations dealing with chains would be able

[2] Anything else would be madness.

[3] If you find a way to recursively instantiate variadic containers in a meta-function, please tell me about it!

to be performed on the **void** type, which effectively represents the zero-length chain.

Following the last chapter the first order of business is to test the new chain type:

Listing 26: Test `exponents_chain<>`

```
static_assert(exponents_chain<23, exponents_chain<42, exponents_chain<13>>>::
    value == 23,
              "Test exponents_chain value retrieval");

static_assert(exponents_chain<23, exponents_chain<42, exponents_chain<13>>>::
    tail::value == 42,
              "Test exponents_chain value retrieval");

static_assert(exponents_chain<23, exponents_chain<42, exponents_chain<13>>>::
    tail::tail::value == 13,
              "Test exponents_chain value retrieval");
```

factory This works, and it can be processed recursively, but it is very annoying to write. A simple factory can make this much more convenient:

Listing 27: An `exponents_chain<>` factory

```
template <sint... Exponents>
struct exponents {
    typedef void type;
};

template <sint Exponent, sint... Tail>
struct exponents<Exponent, Tail...> {
    typedef exponents_chain<Exponent, typename exponents<Tail...>::type>
        type;
};

template <sint... Exponents>
using exponents_t = typename exponents<Exponents...>::type;
```

Note how each of the following tests makes use of the previously tested factory instance, so in each test only one level of direct `exponents_chain<>` use is required:

Listing 28: Test `exponents<>` factory

```
static_assert(is_same<exponents_t<>, void>::value,
              "Test exponents_chain factory");
static_assert(is_same<exponents_t<0>, exponents_chain<0>>::value,
              "Test exponents_chain factory");
static_assert(is_same<exponents_t<1, 0>, exponents_chain<1, exponents_t<0>>>::
    value,
              "Test exponents_chain factory");
static_assert(is_same<exponents_t<2, 1, 0>, exponents_chain<2, exponents_t<1,
    0>>>::value,
              "Test exponents_chain factory");
```

operators The `exponents_chain<>` meta-type needs to be used for the same operations as the `rational<>` meta-type, i.e. multiplication and division. Work-

ing with exponents this translates into addition and subtraction.

The first recursive template in this paper was the `euclid<>` meta-function defined in listing 6 (p. 10). It performs a recursive computation and returns the result when the recursion terminates.

The `exponents_chain<>` meta-type itself is recursive, so meta-functions do not just need to be recursive, they also need to return a recursively constructed type. The unary negation operand illustrates the principle:

Listing 29: Negation meta-function for `exponents_chain<>`

```
template <class Op>
struct exponents_unary {
    typedef exponents_chain<
        -Op::value, typename exponents_unary<typename Op::tail>::negate>
        negate;
};

template <>
struct exponents_unary<void> {
    typedef void negate;
};

template <class Op>
using exponents_negate_t = typename exponents_unary<Op>::negate;
```

To complete the required operands pairwise addition and subtraction are required, the recursion works just like in listing 29:

Listing 30: Addition meta-function for `exponents_chain<>`

```
template <class Lhs, class Rhs>
struct exponents_binary {
    typedef exponents_chain<
        Lhs::value + Rhs::value,
        typename exponents_binary<typename Lhs::tail,
                                  typename Rhs::tail>::add> add;
};

template <>
struct exponents_binary<void, void> {
    typedef void add;
};

template <class Lhs, class Rhs>
using exponents_add_t = typename exponents_binary<Lhs, Rhs>::add;

template <class Lhs, class Rhs>
using exponents_sub_t = exponents_add_t<Lhs, exponents_negate_t<Rhs>>;
```

Like everything else exponents handling should be tested:

Listing 31: Test meta-functions for `exponents_chain<>`

```
static_assert(is_same<exponents_t<-1, -2, -3>,
                      exponents_negate_t<exponents_t<1, 2, 3>>>::value,
              "Test unary negate operation");

static_assert(is_same<exponents_add_t<exponents_t<1, 2, 3>, exponents_t<4, 5,
    6>>,
                      exponents_t<5, 7, 9>>::value,
              "Test addition");

static_assert(is_same<exponents_sub_t<exponents_t<1, 2, 3>, exponents_t<4, 5,
    6>>,
                      exponents_t<-3, -3, -3>>::value,
              "Test subtraction");
```

9 Constants

In section 4 (p. 4) the number π is credited with the need to work with constants. It can also be useful when working with very large or very small numbers. By defining a constant 10 units can work with powers of 10, which avoids the overflow problems posed by rationals.

defining constants The first problem to solve is how users can provide floating point constants to a meta-type. Templates only accept types and integral type instances, which is why the `rational<>` meta-type was created in the first place.

The answer has two aspects, **`constexpr`** can make a floating point value compile time accessible and wrapping it in a type makes it accessible to templates:

Listing 32: Defining a constant

```
struct constant_Pi {
    static constexpr double const value{3.14159265358979323846};
};
```

A preprocessor macro could be used to simplify the creation of constant containers, but macros usually provide an extra layer of obfuscation that is undesirable.

breakdown At this point it helps to break the task down. It can be divided into meta-types and meta-functions. The following list reflects the bottom-up approach that was used in the original development:

- Take a constant to an integer power (meta-function).
- Store a list of constants (meta-type).
- Store a list of exponents (meta-type).
- Support addition and subtraction for the list of exponents (meta-function).
- Create a product of all constants to the power of their respective exponents (meta-function).
- Pack the constants and exponents together (meta-type).

- Extract the product of all constants to the powers of their exponents from a pack (meta-function).
- Provide multiplication and division for packs (meta-function).

`constant_pow<>` Taking a constant to an integer power can be solved by recursively multiplying the constant:

$$c \in \mathbb{R}, \qquad \text{the constant} \tag{29}$$
$$n \in \mathbb{Z}, \qquad \text{the exponent} \tag{30}$$
$$f(c,n) = 1, \qquad \text{for } n = 0 \tag{31}$$
$$f(c,n) = c \cdot f(c,n-1), \qquad \text{for } n > 0 \tag{32}$$
$$f(c,n) = f(c,-n)^{-1}, \qquad \text{for } n < 0 \tag{33}$$

Implementing these conditions in a template is a little less straightforward. In the following listing $n > 0$ is the default case and implements equation 32, the first specialisation implements 33 and matches $\neg(n > 0)$. It is not invoked for the case $n = 0$, because the template signature for the implementation of equation 31 is more specialised:

Listing 33: Take a constant to an integer power

```
// Case n > 0, see equation 32.
template <typename T, class Constant, sint Exponent, bool = (Exponent > 0)>
struct constant_pow {
    static constexpr T const value{
        T{Constant::value} *
        constant_pow<T, Constant, Exponent - 1>::value};
};

// Case n < 0, see equation 33.
template <typename T, class Constant, sint Exponent>
struct constant_pow<T, Constant, Exponent, false> {
    static constexpr T const value{
        T{1} / constant_pow<T, Constant, -Exponent>::value};
};

// Case n = 0, see equation 31.
template <typename T, class Constant>
struct constant_pow<T, Constant, 0, false> {
    static constexpr T const value{1};
};
```

The previous definition of `constant_Pi` can be used for the test cases:

Listing 34: Test `constant_pow<>`

```
static_assert(constant_pow<double, constant_Pi, 0>::value == 1.,
              "Test constant to the power 0");

static_assert(constant_pow<double, constant_Pi, -1>::value == 1. / constant_Pi::
    value,
              "Test constant to the power -1");

static_assert(constant_pow<double, constant_Pi, 2>::value == constant_Pi::value
    * constant_Pi::value,
              "Test constant to the power 2");

static_assert(constant_pow<double, constant_Pi, -2>::value == 1. / (constant_Pi
    ::value * constant_Pi::value),
              "Test constant to the power -2");
```

`constants_chain<>` The storage container for a list of constants can have a similar shape to `exponents_chain<>`:

Listing 35: A plain `constant_chain<>` implementation

```
template <class Head, class Tail = void>
struct constants_chain {
    typedef Tail tail;
    typedef Head type;
};
```

Because `exponents_chain<>`, defined in listing 25 (p. 16), stores an integral value it is syntactically impossible to provide illegal values. The plain `constants_chain<>` definition in listing 35 has no such guarantees, a library user is free to create chains with types that do not provide a suitable value member.

In order to support early detection, e.g. to discover typos, the implementation should reject such types and fail to compile:

Listing 36: A safer `constant_chain<>` implementation

```
template <class Head, class Tail = void,
          class = void_t<decltype(double(Head::value))>>
struct constants_chain {
    typedef Tail tail;
    typedef Head type;
};
```

This implementation uses `void_t<>` to ensure that the provided type has has a value member from which a **`double`** can be constructed.

Just like the `exponents_chain<>` factory in listing 27 (p. 17), a factory to create `constants_chain<>`s can be defined:

Listing 37: A `constants_chain<>` factory

```
template <class... Constants>
struct constants {
    typedef void type;
};

template <class Head, class... Tail>
struct constants<Head, Tail...> {
    typedef constants_chain<Head, typename constants<Tail...>::type> type;
};

template <class... Args>
using constants_t = typename constants<Args...>::type;
```

exponents According to the breakdown in section 9 (p. 19), a meta-type for storing exponents as well as meta-functions for addition and subtraction are required.

This functionality is already provided by the exponents in section 8 (p. 16).

`constants_prod<>` The next item to implement is the meta-function that recursively creates the product of all constants to the power of their exponents. The function can use `constant_pow<>` and recursively call itself to create the product:

Listing 38: Product of constants to the power of their exponents

```
template <typename T, class Constants, class Exponents>
struct constants_prod {
    static constexpr T const value{
        constant_pow<T, typename Constants::type,
                     Exponents::value>::value *
        constants_prod<T, typename Constants::tail,
                       typename Exponents::tail>::value};
};

template <typename T>
struct constants_prod<T, void, void> {
    static constexpr T const value{1};
};
```

Note that the termination of recursion requires both chains, the constants and the exponents to be **`void`** terminated and have the same length.

`constants_pack<>` The next item on the breakdown is creating a container type that accumulates constants and exponents:

Listing 39: A simple `constants_pack<>` implementation

```
template <class Constants, class Exponents>
struct constants_pack {
    typedef Constants constants;
    typedef Exponents exponents;
};

template <class Constants, sint... Exponents>
using constants_pack_t = constants_pack<Constants, exponents_t<Exponents...>>;
```

An obvious problem is that the library user can provide chains of different lengths. So far two chained list meta-types were created that use the type traits for recursion and terminate a chain with **`void`**. A utility function that compares the length of two given chains can make use of this use of common idioms:

Listing 40: Check whether two cains have the same length

```
template <class Lhs, class Rhs, class = void>
struct is_same_length : false_type {};

template <class Lhs, class Rhs>
struct is_same_length<Lhs, Rhs, void_t<typename Lhs::tail, typename Rhs::tail>>
    : is_same_length<typename Lhs::tail, typename Rhs::tail> {};

template <>
struct is_same_length<void, void> : true_type {};
```

As is usual for tests the default case returns false. A specialisation is selected if both arguments have a `tail` member type. This specialisation recursively calls itself until one of the chains runs out of its `tail`, e.g. because the last `tail` was **`void`**. In that case the default case is invoked. The last specialisation catches the case where both chains end after the same number of recursions.

This meta-function can be used in a similar manner to listing 23 (p. 15) to make a `constants_pack<>` implementation that is safer to use:

Listing 41: A safer `constants_pack<>` implementation

```
template <class Constants, class Exponents,
          class = enable_if_t<is_same_length<Constants, Exponents>::value>>
struct constants_pack {
    typedef Constants constants;
    typedef Exponents exponents;
};

template <class Constants, sint... Exponents>
using constants_pack_t = constants_pack<Constants, exponents_t<Exponents...>>;
```

`constants_pack_prod<>` The `constants_pack<>` meta-type ties the concept of constants and exponents together into a single entity. So the `constants_pack<>` should provide a single interface for all required operations.

A simple wrapper around `constants_prod<>` provides value retrieval:

Listing 42: A wrapper around `constants_prod<>`

```
template <typename, class>
struct constants_pack_prod;

template <typename T, class Constants, class Exponents>
struct constants_pack_prod<T, constants_pack<Constants, Exponents>>
    : constants_prod<T, Constants, Exponents> {};
```

template matching This code illustrates some of the finer points of template selection. A template has a signature, which the compiler must match to use the template. Only after the compiler selected the template, does it select a specialisation. At this stage SFINAE rules kick in and compilation only fails if no well-formed specialisation could be found.

The template matching is the reason that listing 42 requires a declaration. This declaration establishes the template signature, which allows the compiler to select the template. The unraveling of the `constants_pack<>` arguments can only happen in a specialisation.

Because the generic case is only declared and not implemented compilation fails when the template is selected but no matching specialisation can be found, which is the desired behaviour. Illegal code should fail to compile.

The distinction between template matching and selecting a specialisation can become important when working with typical SFINAE constructs like `void_t<>` and `enable_if_t<>`.

Listing 40 (p. 23) uses `void_t<>` in the SFINAE context, i.e. the template has already been selected and the `void_t<>` construct only influences which specialisation is selected.

This is not the case for the `enable_if_t<>` use in listing 41 (p. 23). The `enable_if_t<>` statement is part of the template declaration and thus the template signature. Even given a specialisation that matches a different set of arguments, that does not have the same length requirement, it could never be selected, because the template was already rejected during template matching[4].

So in order to support different sets of arguments for `constants_pack<>`, the declaration would have to be separated from the implementation. The condition in the declaration would be replaced by the traditional **`class = void`** (actually any type would do, this is just a convention). The condition would become part of a specialisation, contesting with other specialisations for being the most specialised well-formed implementation of the template.

operators Multiplication and division can be mapped to addition and subtraction on the involved exponents. Implementing these operations should at this point be straightforward:

[4] Unless the template user cheats by overriding the default template argument (i.e. the condition) by providing a type in its stead.

Listing 43: Operators for the `constants_pack<>` meta-type

```
template <class>
struct constants_pack_unary;

template <class Constants, class Exponents>
struct constants_pack_unary<constants_pack<Constants, Exponents>> {
    typedef constants_pack<Constants, exponents_negate_t<Exponents>>
        invert;
};

template <class... Args>
using constants_pack_invert_t = typename constants_pack_unary<Args...>::invert;

template <class, class>
struct constants_pack_binary;

template <class Constants, class LExponents, class RExponents>
struct constants_pack_binary<constants_pack<Constants, LExponents>,
                             constants_pack<Constants, RExponents>> {
    typedef constants_pack<Constants,
                           exponents_add_t<LExponents, RExponents>> mul;
};

template <class... Args>
using constants_pack_mul_t = typename constants_pack_binary<Args...>::mul;

template <class Lhs, class Rhs>
using constants_pack_div_t =
    typename constants_pack_binary<Lhs, constants_pack_invert_t<Rhs>>::mul;
```

These operators follow the established patterns. Inversion of the pack is mapped to negation of the exponents. Multiplication of two packs is mapped to addition of their exponents. The template specialisation ensures that only packs based on the same constants can be multiplied.

10 Testing Reloaded

The original implementation this paper is based on has a lot more conditions and tests this paper overlooks for the sake of scale and clarity.

Except for the multiplication meta-function for the `rational<>` meta-type in listing 5 (p. 9) illegal use of meta-types and functions has been caught at the template matching/selection level. Either implicitly, by using template unravelling or explicitly by using `enable_if_t<>` and `void_t<>`, e.g. `constants_pack<>` in listing 41 (p. 23).

The overwhelming advantage of **`static_assert()`** is its readability and the ability to produce useful error messages. Most of the safeguards built into the code so far, could be **`static_assert()`** based, which would result in code looking less cryptic and make it much easier for a library user to find out what they did wrong.

The sole reason to choose this less convenient form is that it allows testing whether an illegal construct would fail in an SFINAE context. Which means fails-to-compile tests can be performed in a **`static_assert()`**. E.g. the following listing tests whether `constants_pack<>` does not accept lists, which have different lengths:

Listing 44: Compile failure testing

```
template <class Constants, class Exponents, class = void>
struct test_constants_pack : std::false_type {};

template <class Constants, class Exponents>
struct test_constants_pack<Constants, Exponents,
                           void_t<constants_pack<Constants, Exponents>>>
    : std::true_type {};

static_assert(test_constants_pack<constants_t<constant_Pi>, exponents_t<1>>::
    value,
              "Verify that the test accepts valid cases.");
static_assert(!test_constants_pack<constants_t<constant_Pi>, exponents_t<1,
    2>>::value,
              "Constants and exponents lists with different lengths should be
                  rejected.");
static_assert(!test_constants_pack<constants_t<constant_Pi>, exponents_t<>>::
    value,
              "Constants and exponents lists with different lengths should be
                  rejected.");
```

While testing positive cases that should work is straightforward (i.e. just use the functionality), this testing of cases that *should not compile* is more difficult to achieve, but just as important. The added effort means that it should be given a much higher priority.

Note that such tests should always exhaust all negative and positive cases, to ensure that both the tested code as well as the test work as expected. It is advisable to test new tests, i.e. sabotaging the tested functionality and checking whether the test catches it.

The reason why `static_assert()` was chosen for `rational_operator_mul<>` in listing 5 (p. 9) is that overflow is a technical limitation and can be caused by regular, intended use of the library. Because the error is so non-obvious, issuing a clear error message is more important than the benefits of testing the failure to compile in a `static_assert()`.

11 The Unit Container

In the original design by Stroustrup [23], there were two templates, a `Unit<>` meta-type and a `Value<>` template. The `Unit<>` meta-type represents the exponents of the base units. The `Value<>` template takes a `Unit<>` instance as an argument and carries the payload, a value member referred to as a magnitude. I.e. a value has a unit and a magnitude.

The following code is from slide 27:

Listing 45: Unit verification according to Stroustrup [23]

```
// A unit in the MKS system
template<int M, int K, int S> struct Unit {
    enum { m=M, kg=K, s=S };
};

// A magnitude with a unit
template<typename Unit> struct Value {
    // the magnitude
    double val;
    // construct a Value from a double
    explicit Value(double d): val(d) {}
};

// metres/second type
using Speed = Value<Unit<1, 0, -1>>;
// metres/second/second type
using Acceleration = Value<Unit<1, 0, -2>>;
```

In the following design a different nomenclature is used. The `unit<>` template instance is the type of a value. I.e. the value container is called `unit<>` instead of `Value<>` and the `Unit<>` meta-type is replaced with base units, a conversion factor and constant factors. Those are represented by instances of the `exponents_chain<>`, `rational<>` and `constants_pack<>` meta-types:

Listing 46: The `unit<>` signature

```
template <typename T, class BaseUnits, class Factor,
          class ConstantsPack = constants_pack<void, void>>
struct unit {
    typedef T value_type;
    typedef BaseUnits base_units;
    typedef Factor factor;
    typedef ConstantsPack constants_pack;
    typedef unit<T, BaseUnits, Factor, ConstantsPack> type;
    T value;
    constexpr unit() : value{0} {}
    constexpr explicit unit(T const copy) : value{copy} {}
    // Insert additional constructors and operators here ...
};
```

Note that the constants pack has effectively been made optional by providing a default empty pack. This results in `constants_pack_prod<>` always returning 1.

This definition of the `unit<>` template suffices to create a first set of units for testing.

Listing 47: First test of the `unit<>` signature

```
// Define constant_Pi like in listing 32 (p. 19)
template <sint... Exponents>
using consts = constants_pack_t<constants_t<constant_Pi>, Exps ...>;

typedef unit<double, exponents_t<0, 0, 0, 0>,
             rational<1, 1>, consts<0>> scalar;
typedef unit<double, exponents_t<1, 0, 0, 0>,
             rational<1, 1>, consts<0>> metre;
typedef unit<double, exponents_t<0, 1, 0, 0>,
             rational<1, 1>, consts<0>> kilogram;
typedef unit<double, exponents_t<0, 0, 1, 0>,
             rational<1, 1>, consts<0>> second;
typedef unit<double, exponents_t<0, 0, 0, 1>,
             rational<1, 1>, consts<0>> rad;
typedef unit<double, exponents_t<1, 0, 0, 0>,
             rational<1000, 1>, consts<0>> kilometre;
typedef unit<double, exponents_t<0, 0, 1, 0>,
             rational<60 * 60, 1>, consts<0>> hour;
typedef unit<double, exponents_t<0, 0, 0, 1>,
             rational<1, 180>, consts<1>> deg;

static_assert(kilometre{}.value == 0.,
              "Test default constructor");
static_assert(kilometre{2.}.value == 2.,
              "Test explicit constructor");
```

implicit conversion The promised functionality is implicit converion between *compatible* types. This can be performed with a non-**`explicit`** copy constructor:

Listing 48: A `unit<>` constructor that performs implicit conversion

```
template <class Op>
constexpr unit(Op const copy)
    : value{copy.value *
            (T(rational_div_t<typename Op::factor, Factor>{}) *
             constants_pack_prod<
                 T, constants_pack_div_t<typename Op::constants_pack,
                                         ConstantsPack>>::value)} {}
```

This constructor multiplies the value of the given unit with the conversion factors, like in equation 14 (p. 6). The conversion is ugly to read, the following version is more elaborate:

Listing 49: A more readable `unit<>` conversion constructor

```
// INSIDE THE UNIT<> BODY

// rational<> component of the conversion factor
template <class Op>
using rational_factor_t = rational_div_t<typename Op::factor, Factor>;

// constants_pack<> component of the conversion factor
template <class Op>
using constants_div_t =
    constants_pack_div_t<typename Op::constants_pack, ConstantsPack>;

// value container for the constants_pack factor
template <class Op>
using constants_factor_t = constants_pack_prod<T, constants_div_t<Op>>;

// the constructor
template <class Op>
constexpr unit(Op const copy)
    : value{copy.value *
            (T(rational_factor_t<Op>{}) * constants_factor_t<Op>::value)} {}
```

This version of the conversion constructor is easier to comprehend, however it still has a huge drawback — it converts units with different base units, permitting illegal conversions:

Listing 50: Test `unit<>` conversion constructor

```
static_assert(kilometre{kilometre{3.}}.value == 3.,
              "Test copy construction without conversion");

static_assert(kilometre{metre{2000.}}.value == 2.,
              "Test copy construction with conversion");

static_assert(kilometre{second{2000.}}.value == 2.,
              "Test illegal conversion"); // This should not compile!!!
```

This can be prevented using template unraveling or `enable_if_t<>`:

Listing 51: A strict `unit<>` conversion constructor

```
template <class Op,
          class = enable_if_t<
              is_same<BaseUnits, typename Op::base_units>::value>>
constexpr unit(Op const copy)
    : value{copy.value *
            (T(rational_factor_t<Op>{}) * constants_factor_t<Op>::value)} {}
```

The SFINAE test for illegal conversion failure looks like this:

Listing 52: Test strict `unit<>` conversion

```
// default to false
template <typename, typename, class = void>
struct construct_copy : std::false_type {};

// return true for successful construction of To from From
template <typename To, typename From>
struct test_construct_copy<To, From,
                           void_t<decltype(To{From{}})>> : std::true_type {};

// Test strictness of conversion
static_assert(test_construct_copy<kilometre, kilometre>::value,
              "Verify that test recognises valid case");
static_assert(test_construct_copy<kilometre, metre>::value,
              "Verify that test recognises valid case");
static_assert(!test_construct_copy<kilometre, second>::value,
              "Conversion with different base units should fail");
```

This SFINAE construct can test whether one type is constructible from another. It cannot verify that such construction happens correctly, so it cannot replace regular tests like at the end of listing 49 (p. 29).

trivial operators A lot of operators require the same type for the left-hand and right-hand side operands. Given the conversion operators they are trivial to implement, because the conversion happens implicitly:

Listing 53: Trivial operators

```
// INSIDE THE UNIT<> BODY

// unary operators
constexpr type operator+() const { return *this; }
constexpr type operator-() const { return type{-this->value}; }

// trivial arithmetic operators
constexpr type operator+(type const op) const {
    return type{this->value + op.value}; }
constexpr type operator-(type const op) const {
    return type{this->value - op.value}; }

// boolean operators
constexpr bool operator==(type const op) const {
    return this->value == op.value; }
constexpr bool operator!=(type const op) const {
    return this->value != op.value; }
constexpr bool operator<(type const op) const {
    return this->value < op.value; }
constexpr bool operator<=(type const op) const {
    return this->value <= op.value; }
constexpr bool operator>(type const op) const {
    return this->value > op.value; }
constexpr bool operator>=(type const op) const {
    return this->value >= op.value; }
```

multiplication The multiplication and division operators on the other hand should not just accept convertible units, but any unit (within the system). Instead of returning the type of the current unit these operators should return a

composite type like illustrated by the equations 18 to 23 (p. 7).

To avoid messy code like in listing 48 (p. 28) these type conversions should be available as meta-functions. To enable template unraveling the `unit<>` template should be declared before the meta-functions, but the meta-functions should be available in the `unit<>` body. So the declaration and the definition of `unit<>` should be split:

Listing 54: Meta-function context for `unit<>`

```
template <typename T, class BaseUnits, class Factor,
          class ConstantsPack = constants_pack<void, void>>
struct unit;

// meta-functions go here ...

template <typename T, class BaseUnits, class Factor, class ConstantsPack>
struct unit<T, BaseUnits, Factor, ConstantsPack> {
    // unit body ...
};
```

For the implementation of meta-functions the established patterns apply, create a meta-function that performs inversion and one that performs multiplication. The remaining functionality can be tied up in alias templates:

Listing 55: Meta-functions for `unit<>`

```
template <class>
struct unit_unary;

template <typename T, class BaseUnits, class Factor, class ConstantsPack>
struct unit_unary<unit<T, BaseUnits, Factor, ConstantsPack>> {
    typedef unit<T, exponents_negate_t<BaseUnits>,
                 rational_invert_t<Factor>,
                 constants_pack_invert_t<ConstantsPack>> invert;
};

template <class... Args>
using unit_invert_t = typename unit_unary<Args...>::invert;

template <class Lhs, class Rhs>
struct unit_binary {
    typedef unit<
        typename Lhs::value_type,
        exponents_add_t<typename Lhs::base_units,
                        typename Rhs::base_units>,
        rational_mul_t<typename Lhs::factor, typename Rhs::factor>,
        constants_pack_mul_t<typename Lhs::constants_pack,
                             typename Rhs::constants_pack>> mul;
};

template <class... Args>
using unit_mul_t = typename unit_binary<Args...>::mul;

template <class Lhs, class Rhs>
using unit_div_t = unit_mul_t<Lhs, unit_invert_t<Rhs>>;
```

What the meta-functions do is straightforward enough:

- `unit_invert_t<Op>`
  - negate base unit exponents

  - invert rational factor
  - invert constant pack
- `unit_mul_t<Lhs, Rhs>`
  - add base unit exponents
  - multiply rational factors
  - multiply constant packs
- `unit_div_t<Lhs, Rhs>`
  - invert `Rhs`
  - call `unti_mul_t<>`

With the plumbing done the multiplication and division operators are simple:

Listing 56: Multiplication and division operators

```
// INSIDE THE UNIT<> BODY

// multiply with unit<> instance
template <class Op, class Result = unit_mul_t<type, Op>>
constexpr Result operator*(Op const op) const {
    return Result{this->value * op.value};
}

// divide by unit<> instance
template <class Op, class Result = unit_div_t<type, Op>>
constexpr Result operator/(Op const op) const {
    return Result{this->value / op.value};
}
```

With all the complexity moved into the type system, the function bodies are reduced to multiplying the payloads. I.e. unit multiplication and division add no runtime cost.

scalars A special case is multiplication with and division by scalar values. These can be performed without type mutations and *should* be supported. The alternative would be to create a scalar unit with all base unit exponents zeroed, instances of which would have to be constructed explicitly.

These scalar operators are as simple as the trivial operators in section 11 (p. 30):

Listing 57: Scalar multiplication and division operators

```
// INSIDE THE UNIT<> BODY

// multiply with scalar value
constexpr type operator*(T const op) const {
    return type{this->value * op};
}

// divide by scalar value
constexpr type operator/(T const op) const {
    return type{this->value / op};
}
```

These operators make statements like `metre{2.} * 5.` possible. To support `5. * metre{2.}` binary operators outside of the `unit<>` body need to be defined:

Listing 58: Left hand scalar multiplication and division operators

```
template <typename T, class... Spec>
constexpr unit<T, Spec...>
operator*(T const lhs, unit<T, Spec...> const rhs) {
    return unit<T, Spec...>{lhs * rhs.value};
}

template <typename T, class... Spec>
constexpr unit_invert_t<unit<T, Spec...>>
operator/(T const lhs, unit<T, Spec...> const rhs) {
    return unit_invert_t<unit<T, Spec...>>{lhs / rhs.value};
}
```

Unlike the operators in listing 57 (p. 32), these operators are templates. Thus the type `T` has to be matched exactly, because the compiler does not perform implicit conversions to match templates. E.g. if **`typename T = float`**, and the left hand operand is a double, template matching fails:

Listing 59: Templates require exact matches

```
src/model/Tool.cpp:113:20: error: invalid operands to binary expression ('double
     ' and 'types::rpms')
this->rotation += .5 * frame.acc * frametime * frametime;
                  ~~ ^ ~~~~~~~~~

src/model/../units/Units.hpp:1853:28: note: candidate template ignored: deduced
     conflicting types for parameter 'T' ('double' vs. 'float')
constexpr unit<T, Spec...> operator*(T const lhs, unit<T, Spec...> const rhs) {
                           ^
```

This inconsistency can be solved by defining them as **`friend`** functions from within the struct body. Friend functions are commonly declared in class bodies to allow certain functions (defined somewhere else) to access private members. But they can also be defined from within a class/struct body, where all the template's arguments and traits are available:

Listing 60: Left hand scalar multiplication and division friend operators

```
// INSIDE THE UNIT<> BODY

friend constexpr type operator*(T const lhs, type const rhs) {
    return type{lhs * rhs.value};
}

friend constexpr unit_invert_t<type>
operator/(T const lhs, type const rhs) {
    return unit_invert_t<type>{lhs / rhs.value};
}
```

Whenever the `unit<>` template is instantiated, the compiler emits the friend functions as regular functions. That is the case, even when the declaration of the friend function itself is a template.

One thing to pay attention to is that every friend function defined in a template body is defined in a way that makes its signature unique to the template instance. In listing 60 this is ensured by having a `type` argument (`type` is the trait representing the current template instance) in the signature.

What happens if the one definition rule is violated by a friend function

emitted from a template depends on the compiler. Clang 3.4.1 and 3.6.2 as well as GCC 5.2.0 ignore it (i.e. they emit implementation defined behaviour). Visual Studio 2015 RTM (Release To Manufacturing) treats it as an error.

The one definition rule was established by the C++03 [8] standard.

modulo One operator that has been omitted so far is the modulo operator. The modulo function is well defined for numbers in $\mathbb{R}$, but C++ only provides the modulo operator `%` for integral types ($\mathbb{Z} \subset \mathbb{R}$). The `<cmath>` header provides the `fmod()` function for floating point types.

So two definitions of the modulo operator, one that uses the operator and one that uses the `fmod()` function, are required:

Listing 61: Two implementations of the modulo operator

```
// INSIDE THE UNIT<> BODY

// for integral types
constexpr operator%(type const op) const {
    return type{this->value % op.value};
}

// for floating point types
constexpr operator%(type const op) const {
    return type{fmod(this->value, op.value)};
}
```

Of course compilers do not accept these two functions with identical signatures to coexist. So a selection criteria is required, e.g. the `<type_traits>` header provides the `is_integral<>` meta-function.

Another approach, which is more favourable for supporting user-defined types, is to check for the modulo operator and fall back to `fmod()`. To facilitate this a meta-function that tests for the existence of the operator needs to be defined:

Listing 62: Test for modulo operator

```
template <typename T, class = void>
struct has_op_modulo : false_type {};

template <typename T>
struct has_op_modulo<T, void_t<decltype(T{} % T{})>> : true_type {};
```

This test requires the tested type to be default constructible and provide the modulo operator to return true.

Listing 63: Two implementations of the modulo operator

```
// INSIDE THE UNIT<> BODY

template <bool NativeMod = has_op_modulo<T>::value>
constexpr enable_if_t<NativeMod, type> operator%(type const op) const {
    return type{this->value % op.value};
}

template <bool NativeMod = has_op_modulo<T>::value>
constexpr enable_if_t<!NativeMod, type> operator%(type const op) const {
    return type{fmod(this->value, op.value)};
}
```

This enables statement like `metre{1337.} % kilometre{1.} == metre{337.}`. Note that this kind of statement is only a constant expression if the underlying operator or `fmod()` are constant expressions.

12 Using the Library

This concludes the implementation of the `unit<>` template. Defining units like in listing 47 (p. 28) is the job of the library user. The `unit<>` struct template is designed to replace builtin types with small hassle and huge benefits.

literals For a more seamless experience, literals should be defined. User-defined literals provide a means of creating modifiers like those provided by the language, e.g. `5.F` or `12L`.

The only limitation is that user-defined literals have to start with an underscore (McIntosh et al. [17]):

> Literal suffix identifiers that do not start with an underscore are reserved for future standardization.

defining literals User-defined literals depend on the declaration of a literal operator:

Listing 64: Literal operators for the units in listing 47

```
constexpr metre operator "" _m(long double const value) {
    return metre{double(value)};
}
// use: 1337._m

constexpr kilometre operator "" _km(long double const value) {
    return kilometre{double(value)};
}
// use: 1.337_km
```

Note that the operator signature takes a **`long double`** instead of the desired **`double`**. This rule is described in the standard proposal [17]:

> 3. The declaration of a literal operator shall have a *parameter-declaration-clause* equivalent to one of the following:

```
char const*
unsigned long long int
long double
char const*, std::size_t
wchar_t const*, std::size_t
char16_t const*, std::size_t
char32_t const*, std::size_t
```

4. A *raw literal operator* is a literal operator with a single parameter whose type is **`char const*`** (the first case in the list above).

using literals Brief examples were alread provided in listing 64 (p. 35). To use a literal suffix just concatenate it to the literal:

```
auto const velocity = 100._km / 1._h;
```

Use **`auto`** to use the *natural* type of an expression.

User-defined literals require an exact type match. E.g. `5_m` is not caught by the **`long double`** literal:

Listing 65: Literal operator type mismatch

```
c++ -O2 -pipe -march=native  -std=c++11 -Wall -Werror -pedantic   -o bin/io/
    Parse.o -c src/io/Parse.cpp
In file included from src/io/Parse.cpp:9:
In file included from src/io/../model/Tool.hpp:10:
src/io/../model/Units.hpp:256:16: error: no matching literal operator for call
    to 'operator "" _m' with argument of type 'unsigned long long' or 'const
    char *', and no matching literal operator template
static_assert(5_m == 5000._mm, "Test");
               ^
```

As the error message in listing 65 shows, every literal number can also be caught as a string literal. This allows the creation of literals for big numbers (integers not fitting into 64 bits), or supporting different bases, e.g. base 64, without using quotes.

Literals can also be used to create new types:

Listing 66: Using literals to define new types

```
typedef decltype(1._m/1._s) mps;
typedef decltype(1._km/1._h) kmph;

void showSpeed(mps const speed) {
    std::cout << "Speed in  m/s: " << mps{speed}.value << '\n'
              << "Speed in km/h: " << kmph{speed}.value << '\n';
}
```

This code is equivalent to the following listing:

Listing 67: Using literals to define new types

```
typedef decltype(metre{}/second{}) mps;
typedef decltype(kilometre{}/hour{}) kmph;

void showSpeed(mps const speed) {
    std::cout << "Speed in  m/s: " << mps{speed}.value << '\n'
              << "Speed in km/h: " << kmph{speed}.value << '\n';
}
```

adaptive signatures In listings 66 (p. 36) and 67 the signature enforces conversion.

This can be avoided using function templates:

Listing 68: Using literals to define new types

```
// base unit for velocities
typedef exponents_t<1, 0, -1, 0> base_velocity;

// output units
typedef decltype(1._m/1._s) mps;
typedef decltype(1._km/1._h) kmph;

// match any unit that has the right base units
template <class... Args>
void showSpeed(unit<base_velocity, Args...> const speed) {
    std::cout << "Speed in  m/s: " << mps{speed}.value << '\n'
              << "Speed in km/h: " << kmph{speed}.value << '\n';
}
```

This approach also works in combination with **`inline`** and **`constexpr`**.

It also shows how to extract plain values from a unit typed value. Note the explicit conversion in listing 66 (p. 36), even though `speed` already had the desired unit. When extracting the magnitude from a unit typed value, the required output unit should always be stated explicitly. This makes the code more readable and safer to maintain. E.g. the same code still works in listing 68, where the input unit is constrained to velocities, but otherwise unknown.

13 Summary

The following recap of all the previous sections concludes this article:

1. Units are troubling and mistakes potentially dangerous.
2. Units must be strongly typed without causing additional runtime cost.
3. C++11 [9] has much better meta-programming support.
4. Units are sets of properties, describing base units and conversion factors.
5. Types and functions are a semantic distinction in meta-programming.
6. Create meta-types and recursive meta-functions.
7. Test everything, test bottom up, test early, Test Everything!
8. Meta-types can be recursive, too.
9. Break problems down into atomic pieces and solve them one at a time.
10. Do More Testing!
11. Tie it together into a unit template.

12 Use literals, avoid unnecessary conversions.
13 See recursion in section 5 (p. 7).

The complete source code is available from https://github.com/lonkamikaze/units.

References

[1] Walter E. Brown. Transformationtrait alias void_t. 2014. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3911.pdf.

[2] Dominic Fandrey. Mesh based dynamic octree modelling, 2013.

[3] Dominic Fandrey. Validation and conversion of physical units at compile time - leveraging the power of the c++11 type system in your simulation model. In *Proceedings of the 32nd Chaos Communication Congress (32C3)*, 2015.

[4] James Gleick. A bug and a crash. *New York Times Magazine*, 1996. URL http://www.around.com/ariane.html.

[5] Douglas Gregor, Jaakko Järvi, and Gary Powell. Variadic templates (revision 3). 2006. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2080.pdf.

[6] Douglas Isbell and Don Savage. Mars climate orbiter failure board releases report, 1999. URL http://spaceflight.nasa.gov/spacenews/releases/1999/h99-134.html.

[7] ISO/IEC. 14882:1998 information technology - programming languages - c++, 1998.

[8] ISO/IEC. 14882:2003 information technology - programming languages - c++, 2003.

[9] ISO/IEC. 14882:2011 information technology - programming languages - c++, 2011.

[10] ISO/IEC. 14882:2014 information technology - programming languages - c++, 2014.

[11] Jaakko Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype and auto (revision 4). 2004. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1705.pdf.

[12] Robert Klarer, Dr. John Maddock, Beman Dawes, and Howard Hinnant. Proposal to add static assertions to the core language (revision 3). 2004. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html.

[13] Stephan T. Lavavej. C++11/14 stl features, fixes, and breaking changes in vs 2013. *Visual C++ Team Blog*, 2013. URL http://blogs.msdn.com/b/vcblog/archive/2013/06/28/c-11-14-stl-features-fixes-and-breaking-changes-in-vs-2013.aspx.

[14] Stephan T. Lavavej. C++11/14/17 features in vs 2015 rtm. *Visual C++ Team Blog*, 2015. URL http://blogs.msdn.com/b/vcblog/archive/2015/06/19/c-11-14-17-features-in-vs-2015-rtm.aspx.

[15] Prof. Jacques-Louis Lions, Dr. Lennart Lübeck, Mr. Jean-Luc Fauquembergue, Mr. Gilles Kahn, Prof. Dr. Ing. Wolfgang Kubbat, Dr. Ing. Stefan Levedag, Dr. Ing. Leonardo Mazzini, Mr. Didier Merle, and Dr. Colin O'Halloran. Ariane 5 flight 501 failure, 1996. URL http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[16] Robin Lloyd. Metric mishap caused loss of nasa orbiter. *CNN.com*, 1999. URL http://edition.cnn.com/TECH/space/9909/30/mars.metric.02/.

[17] Ian McIntosh, Michael Wong, Raymond Mak, Robert Klarer, Jens Maurer, Alisdair Meredith, Bjarne Stroustrup, and David Vandevoorde. User-defined literals (aka. extensible literals (revision 5)). 2008. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2765.pdf.

[18] Media Relations Office. Nasa's mars climate orbiter believed to be lost, 1999. URL http://spaceflight.nasa.gov/spacenews/releases/1999/jpl-092399.html.

[19] Gabriel Dos Reis. Variable templates (revision 1). 2013. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3651.pdf.

[20] Gabriel Dos Reis and Bjarne Stroustrup. Templates aliases (revision 3). 2007. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2258.pdf.

[21] Gabriel Dos Reis, Bjarne Stroustrup, and Jens Maurer. Generalized constant expressions — revision 5. 2007. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2235.pdf.

[22] Matthias Christian Schabel and Steven Watanabe. *Boost.Units 1.1.0*, 2015. URL http://www.boost.org/doc/libs/1_59_0/doc/html/boost_units.html.

[23] Bjarne Stroustrup. Keynote - bjarne stroustrup: C++11 style. In *GoingNative*, 2012. URL https://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style.

[24] Erwin Unruh. Temple metaprogrammierung, 2002. URL http://www.erwin-unruh.de/meta.html.

[25] Todd Veldhuizen. Template metaprograms. 1995.