Units.h manual

Units.h is brought to you by science.cplusplus.engineering and can be downloaded from the Patreon page at https://www.patreon.com/sciencecplusplusengineering. Units.h is not free, but we don't put any copy protection on the code. Instead we simply trust that if you think that Units.h is good, you will go to the Patreon page and join. It doesn't cost much and it pays for me to maintain this software. By

Units.h is a C++ library which deals with physical dimensions and units at compile time. A set of classes represent the units themselves and another class, called Physical, represents a physical value with a unit. You should note that you will never create an object from one of the units classes – the units classes simply serve as the template parameter of a Physical and you shouldn't need to use any of their methods. You just interact with the Physical class.

By using the Unit classes and Physical objects in this way we get some awesome advantages. Here are some of the things Units.h can do:

1) Units are known at compile time, so if you accidentally add two Physicals with different units you get a compile time error, not a runtime bug.
2) Units are tracked through all calculations. When you add subtract, multiply, divide or raise to a power, you get a result that is in the correct resulting units.
3) Values are automatically converted between units when needed and only when needed.
4) Using units takes up zero additional memory compared to the same floating point value type.
5) Using Physicals should be just as fast as using the same floating point value types. It may even be faster, as some unit conversions can be performed at compile time and they are only performed when needed.
6) Unit strings are automatically added to text output (but can be omitted if needed).
7) Values are kept in the unit that you define them as, not converted to SI behind the scenes.
8) Complex units are permitted, without being reduced down to their base SI unit. E.g. kg kg^{-1} and mm^6 m^{-3} are permitted units, which are distinct from, but practically equivalent to unitless and μm^3.
9) Physicals with distinct units but the same dimensionality can be added, subtracted and equated. For example you can add m s^{-1} to mm^2 kg m^{-1} g^{-1} ns^{-1}. If you choose.

Tutorial

Let's jump right in. Units.h is a header only template library. To use it just #include Units.h. You will need to be using the C++20 standard, or the C++17 standard with uchar8_t turned on. In our first program, we are going to calculate a speed from a length and a time and output it to the console.

```
#include <iostream>
#include <Units.h>

int main()
{
    sci::Physical<sci::Metre<>, double> length(21.0);
    sci::Physical<sci::Second<>, double> time(5.0);
```

```
    std::cout << "Speed = " << length / time;

    return 0;
}
```

You will see that we created a specific type of object known as a Physical to hold our length and our time. Physical is a templated class, receiving two templated parameters - the unit type and the value type. The units used here are Metre and Second. Metre and Second are also templated classes, but in this case we use the default template parameters (hence the empty angle brackets) and we will discuss the parameters more later. The value type is double. You could equally use float or std::complex, or in fact any other type for which the usual mathematical operators and std math functions work. This could even be your own class.

The output from our first quick example should be

```
Speed = 4.2 m s-1
```

You will notice that our output shows the correct units, which is the first advantage of Units.h – all units are automatically tracked. This happens at compile time and does not affect runtime. Let's modify our code to tidy things up a little. We are also going to add in another calculation, this time of the area of a square. We can calculate and area of a square in two different ways, multiply the length by itself, or raise the length to the power two. From the area, we can recalculate the length by square rooting.

```
#include <iostream>
#include <Units.h>

typedef sci::Physical<sci::Metre<>, double> metre;
typedef sci::Physical<sci::Second<>, double> second;

int main()
{
    metre length(21.0);
    second time(5.0);
    auto area1 = length * length;
    auto area2 = sci::pow<2>(length);
    std::cout << "Speed = " << length / time << std::endl;
    std::cout << "Area1 = " << area1 << std::endl;
    std::cout << "Area2 = " << area2 << std::endl;
    std::cout << "Length1 = " << sci::sqrt(area1) << std::endl;
    std::cout << "Length2 = " << sci::sqrt(area2) << std::endl;

    return 0;
}
```

You will note that we tidied up the long declaration needed for a metre, by using typedefs. There is no need to do this, but it definitely makes life easier. My personal preference is to do this using camelback with a lower case first letter, so I know this is a typedef, not a class. You may wish to create your own header file containing unit definitions and just keep adding to it as you need

The output of this example is

```
Speed = 4.2 m s-1
Area1 = 441 m m
Area2 = 441 m2
Length1 = 21 m1/2 m1/2
Length2 = 21 m
```

Note here the difference in units in area 1 and area 2. Units.h distinguishes between multiplying identical units vs. raising them to a power. The unit metre metre is actually a different (but almost entirely equivalent) class to metre squared. You can use them identically and do assignments between them without worrying at all. The only time you will really see the difference is when outputting the unit string. You might ask why we make this distinction? It is because often a field of science will have standard units that are not compounded down. For example I work in the field of meteorology some units that we use regularly are kg kg^{-1} (mass of water per mass of air) $mm^6 m^{-3}$ (radar reflectivity) and μm^2 cm^{-3} (aerosol surface area per unit volume). The users of these units do not want them reduced down to their simplest forms – so by default, Units.h does not reduce down multiplications or divisions into powers when outputting unit strings. You will, however notice that Units.h does combine powers and roots, so Length2 is given the unit m, not (m2)1/2.

There may be times when you do want to reduce multiplied units to a power. You can do this manually by casting or assigning your Physical to the appropriate type as in our next example. If two units are equivalent, then you can cast between them and assign between them. Your compiler is probably smart enough to do this at compile time, so it will probably have no runtime cost. If your units have the same dimensionality, but different units, e.g. millimetres squared and metres squared, then Units.h will do the conversion for you.

```
#include <iostream>
#include <Units.h>

typedef sci::Physical<sci::Metre<>, double> metre;
typedef sci::Physical<sci::Second<>, double> second;
typedef sci::Physical<sci::Metre<2>, double> metreSquared;
typedef sci::Physical<sci::Metre<3, sci::micro>, double>
microMetreCubed;
typedef decltype(sci::Physical<sci::Metre<6, sci::milli>,
    double>()/sci::Physical<sci::Metre<3>,double>())
    radarReflectivity;
//below is an alternate way to define radarReflectivity
//using the DividedUnit class
//typedef sci::Physical<sci::DividedUnit<sci::Metre<6, sci::milli>,
//  sci::Metre<3>>, double> radarReflectivity;

int main()
{
    metre length(21.0);
    second time(5.0);
    auto area1 = length * length;
    auto area2 = sci::pow<2>(length);
    std::cout << "Speed = " << length / time << std::endl;
    std::cout << "Area1 = " << metreSquared(area1) << std::endl;
    std::cout << "Area2 = " << area2 << std::endl;
```

```
    std::cout << "Length1 = " << metre(sci::sqrt(area1)) <<
        std::endl;
    std::cout << "Length2 = " << sci::sqrt(area2) << std::endl;
    microMetreCubed volume = sci::pow<3>(length);
    std::cout << "Volume = " << volume << std::endl;
    std::cout << radarReflectivity(1) << " = " <<
        microMetreCubed(radarReflectivity(1)) << std::endl;

    return 0;
}
```

This example shows how we can raise our units to powers, apply exponents and convert between equivalent dimensions.

This is the first time we have used our template parameters for one of our unit classes. The first template parameter raises the unit to an integer power, the second template parameter gives us an exponent for our unit. You can see that our typedef for metreSquared uses the first template parameter, in this case the power is 2. To typedef microMetreCubed we use 3 for the first template parameter and sci::micro for the second parameter.

You can also see that we built up our radarReflectivity typedef using decltype. There is an alternate method to do this using a class called DividedUnit which is shown in the comments. There are similar classes called MultipliedUnit and, PoweredUnit and RootedUnit.

The output of this example is

```
Speed = 4.2 m s-1
Area1 = 441 m2
Area2 = 441 m2
Length1 = 21 m
Length2 = 21 m
Volume = 9.261e+21 µm3
1 mm6 m-3 = 1 µm3
```

We have converted our length and area units to something more presentable. You can also see how we calculated a volume in metres cubed and it was automatically converted to micrometres cubed upon assignment to a Physical with those units. Finally, you can see that the radar unit we described above as $mm^6\ m^{-3}$, is equivalent to μm^3.

Let's now see how Units.h stops us introducing difficult to find runtime bugs into our code, instead giving us compile time errors which we can easily locate and fix. The following example shows a number of these scenarios.

```
#include <iostream>
#include <Units.h>

typedef sci::Physical<sci::Metre<>, double> metre;
typedef sci::Physical<sci::Second<>, double> second;
typedef sci::Physical<sci::Unitless, double> unitless;
typedef sci::Physical<sci::Radian<>, double> radian;
typedef decltype(radian() / second()) radianPerSecond;
```

```
int main()
{
	metre length(21.0);

	second time = length; //error1

	auto badResult = time + length; //error2

	length = 5.0; //error3
	//length = metre(5.0)

	length = length / 2.0; //error4
	//length = length / unitless(2.0);

	auto logLength = sci::log10(length); //error5
	//auto logLength = sci::log10(length / metre(1.0));

	auto waveAmplitude = sci::cos(time); //error6
	//radianPerSecond angularFrequency(2.0);
	//auto waveAmplitude = sci::cos(angularFrequency * time);

	return 0;
}
```

We will go through each of the compilation errors in turn.

Error 1: It makes no physical sense to convert a length into a time, they have different dimensions.

Error 2: It is physically meaningless to add a time and a length.

Error 3: A Physical cannot be assigned by a raw double. Instead you must call the Physical constructor so that you explicitly state the unit of the value you are using.

Error 4: maths calculations cannot mix physicals and raw floating point numbers. You must use the Physical constructor to explicitly provide the unit for the value or indicate that it is unitless.

Error 5: Although this is something often done in the real world, the logarithm function should only be called on dimensionless quantities. Convert to dimensionless first.

Error 6: Units.h checks that the input for trigonometric functions is an angle. Again, ensure the unit is correct by multiplying/dividing by an angular frequency, wavelength or other appropriate parameter.

As well as SI units, we can also use a range of non-SI units.

```
#include <iostream>
#include <Units.h>

typedef sci::Physical<sci::Metre<>, double> metre;
typedef sci::Physical<sci::Inch<>, double> inch;
typedef sci::Physical<sci::Kelvin<>, double> kelvin;
typedef sci::Physical<sci::Rankine<>, double> rankine;
typedef sci::Physical<sci::Degree<>, double> degree;
```

```
typedef sci::Physical<sci::Unitless, double> unitless;

int main()
{
	metre lengthSi(21.0);
	inch lengthNonSi = lengthSi;

	kelvin temperatureSi(273.15);
	rankine temperatureNonSi = temperatureSi;

	degree angle(90);

	std::cout << lengthSi << " = " << lengthNonSi << "\n";
	std::cout << temperatureSi << " = " << temperatureNonSi << "\n";
	std::cout << "cos of " << angle << " = " << sci::cos(angle);
	return 0;
}
```

The output from this should be:

```
21 m = 826.772 "
273.15 K = 491.67 °Ra
cos of 90 ° = 6.12323e-17
```

This is exactly the conversion we would expect and we can see that (within a small rounding error) the trigonometric functions can deal appropriately with angles in degrees or any other unit. If a unit that you require is missing, then Units.h includes a macro called MAKE_SCALED_UNIT to easily add additional units.

You should note that two units are conspicuous by their absence are Celsius and Fahrenheit. Units.h does not support unis that have an offset. This is because when we subtract two values with an offset, the offset disappears. So for example a temperature of 10 °C converts to 283.15 K, however, the temperature difference between 27 °C and 37 °C = 10 °C, should convert into a temperature difference of 10 K. Units.h cannot know whether the offset of 273.15 should be applied or not. Although it is possible to implement offsets, doing so adds significant additional complexity and may increase memory use and reduce the ability to perform compile time calculations. Hence Units.h chooses not to support Celsius and Fahrenheit – instead use Kelvin and Rankine.

There are likely to be times when you wish to separate the value and the unit. A common example would be when outputting a table of values. Generally the unit would be in a header at the top of the table and the values in each row of the table would show no unit. You may also want to get the full length word version of a unit, rather than just the sybol representation. Units.h provides the ability to get the unit string separately using the getShortUnitString() and getlongUnitString() methods of either a Physical or a unit. You can get the value of a Physical using the value() method, but note that the value() method takes a template parameter to specify the unit that you wish to get the value as – it does not automatically guess the appropriate unit in order to avoid introducing bugs if the Physical's unit is incorrectly recalled by the user.

```
#include <iostream>
#include <Units.h>
#include<vector>

typedef sci::Physical<sci::Metre<1,sci::kilo>, double> kilometre;
```

```
typedef sci::Physical<sci::Mile<>, double> mile;
typedef sci::Physical<sci::Hour<>, double> hour;
typedef sci::Physical<sci::Second<>, double> second;
typedef decltype(kilometre() / hour()) kilometrePerHour;

int main()
{
	kilometre europeanTrackLength(4.3);
	mile ukTrackLength(2.8);
	std::vector<int> carNumber{ 61, 66, 17 };
	std::vector<second> ukLapTimes{ second(130.0), second(128.4),
		second(142.6) };
	std::vector<second> europeanLapTimes{ second(134.1), second(126.1),
		second(140.8) };

	std::cout << "car\tspeed-uk\tspeed-Europe\n";
	std::cout << "number\t" <<
		kilometrePerHour::getShortUnitString<std::string>();
	std::cout << "\t\t" <<
		kilometrePerHour::getShortUnitString<std::string>() << "\n";
	for (size_t i = 0; i < 3; ++i)
	{
		std::cout << carNumber[i] << "\t";
		std::cout << (ukTrackLength /
			ukLapTimes[i]).value<kilometrePerHour>() << "\t\t";
		std::cout << (europeanTrackLength /
			europeanLapTimes[i]).value<kilometrePerHour>() << "\n";
	}

	return 0;
}
```

This should give an output like

```
car     speed-uk        speed-Europe
number  km hr-1         km hr-1
61      124.786         115.436
66      126.341         122.76
17      113.76          109.943
```

You can see that although the code is a little longer, by being explicit with the units during the value() call, we ensure that we don't forget that our ukTrackLength variable was in miles. This ensures we always output the correct units. Although this may be a trivial bug to spot in this short code, in longer codes it is sometimes much more difficult to check what units a variable was using – multiple disasters in history have been generated by such unit problems.

If you have followed these examples then you now know almost all of how Units.h works. You can go include it in your own code and should be able to use its abilities to make your life easier and avoid bugs. Happy coding.

Reference

This section provides a reference to the official API for Units.h.

#defines for unit strings

There are a number of #defines you can use which affect how strings are output. You must #define these before you #include Units.h or you can add them to your compiler's list of preprocessor definitions.

Unit strings often use non ASCII characters. For example µ for micro, or Ω for ohm. The value used to represent these characters varies depending upon your character encoding. If you are using Linux or Mac, then your code will usually use UTF8 encoded Unicode characters and you will probably not use the ASCII characters. If you are using Windows and you are using single byte strings (i.e. std::string), then you are probably using code page strings (often incorrectly referred to as ASCII or ANSI strings). There are a number of different code page encodings for different regions around the world and Units.h defines the characters for use in the CP-1253 "Western Latin" code page encoding and are defined as follows.

```
#define ALTERNATE_MICRO "\xe6"

#define ALTERNATE_OMEGA "\xea"

#define ALTERNATE_PER_MILLE "0/00"

#define ALTERNATE_BASIS_POINT "0/000"

#define ALTERNATE_DEGREE "\xf8"

#define ALTERNATE_ARCMINUTE "'"

#define ALTERNATE_ARCSECOND "\""

#define ALTERNATE_ANGSTROM_SHORT "\x8e"

#define ALTERNATE_ANGSTROM_LONG "\x8e" "ngstrom"

#define ALTERNATE_RANKINE "\xf8" "Ra"
```

If you are in a locale which uses a difference code page, then you may wish to change these alternate strings. To do so, before Units.h is #included, you must add the line

```
#define UNITS_H_NOT_CP1253
```

Then you can set alternate versions of the #defines listed above. If you omit any of the #defines, you will get warnings about the character not being representable. After all the #defines, you may #include Units.h.

sci namespace

Anything listed below will be in the sci namespace and forms part of the API. Some functions are within a unitsPrivate namespace and are absolutely not part of the API. Some items are withi the sci namespace, but not documented below. These are subject to change and are not part of the API – you should use these at your own risk as they may not have been thoroughly tested and may change without warning in later versions of the code.

Unit Constants

The following constants are defined to represent the SI unit prefixes. They can be used as the second template parameter for most of the SI unit classes.

const int64_t yotta = 24;

const int64_t zetta = 21;

const int64_t exa = 18;

const int64_t peta = 15;

const int64_t tera = 12;

const int64_t giga = 9;

const int64_t mega = 6;

const int64_t kilo = 3;

const int64_t hecto = 2;

const int64_t deca = 1;

const int64_t deci = -1;

const int64_t centi = -2;

const int64_t milli = -3;

const int64_t micro = -6;

const int64_t nano = -9;

const int64_t pico = -12;

const int64_t femto = -15;

const int64_t atto = -18;

const int64_t zepto = -21;

const int64_t yocto = -24;

`template<int64_t EXPONENT> struct ExponentTraits`

This traits style struct provides properties of the exponent prefixes. It contains the following methods and constants

`bool validSi;` This will be true if EXPONENT is one of the SI values listed above, or false otherwise.

`template<class STRING> static STRING getName();` This will return the name of the prefix in the STRING type. If STRING is not std::string, std::wstring, std::basic_string<uchar8_t>, std::u16string nor std::u32string, or if EXPONENT is not one of the SI values listed above, then a compilation error will occur.

`template<class STRING> static STRING getPrefix();` This will return the symbol prefix in the STRING type. If STRING is not std::string, std::wstring, std::basic_string<uchar8_t>, std::u16string nor std::u32string, or if EXPONENT is not one of the SI values listed above, then a compilation error will occur.

Units

The unit structs are used as template parameters for the Physical class. The different unit structs do not all inherit from a common base class, but can be interchanged as template parameters by virtue of having a common structure. Users should never instantiate any unit objects, nor access any of the unit members.

The following unit structs represent SI units and accept two template parameters, the first is the power to which the unit is raised, the second is the exponent. By default the power is 1 and the exponent is zero. So the unit sci::Metre<> would represent a metre. The unit sci::Metre<2, sci::milli> would represent an millimetre squared or 1e-6 metres squared. Note the order by which the power and exponent are applied are as you would expect in general scientific usage. All the official SI base and derived units are available as represented by their own class. The class names are:

Ampere, Kelvin, Second, Metre, Gram, Kilogram, Candela, Mole, Radian, Steradian, Hertz, Newton, Pascal, Joule, Watt, Coulomb, Volt, Farad, Ohm, Seimens, Weber, Tesla, Henry, Lumen, Lux, Becquerel, Grey, Seivert, Katal.

It should be noted that classes exist for both gram and kilogram. This is done for consistency with both the SI (where the SI unit is kilogram) and the rest of the library (where the exponent is provided as a template parameter, hence avoiding nonsensical units such as millikilograms). The Kilogram class does not accept the second template parameter. Hence, sci::Gram<1, sci::kilo> is functionally equivalent to sci::Kilogram<> and a milli gram would be defined as sci::Gram<1,sci::milli>.

As well as the SI units, a range of scaled units are provided. The only important difference with respect to the API is that conversions using scaled units will always go via a base SI unit, whereas SI conversions are direct. For example a conversion from kilometres to megametres will be performed by a direct division of 1000. There will not be a multiplication by 1000 to convert to metres followed by a division by 1,000,000 to convert to megametres. However, when converting from yards to inches, there will first be a conversion from yards to metres, then a conversion from metres to inches. This may induce small floating point rounding errors into the calculation. The scaled unit classes still have the same template parameters, the first being the power, the second being the exponent. This means it is possible to create some unusual units such as milliinch or hectare squared. The user may choose to use them in this manner or not as they see fit.

The following scaled unit classes are available:

Degree, ArcMinute, ArcSecond, Turn, Quadrant, Sectant, Hexacontade, BinaryDegree, Gradian, Rankine, Angstrom, AstronomicalUnit, LightYear, Parsec, NauticalMile, Hectare, Tonne, Litre, AtomicMassUnit, SiderealDay, ElementaryCharge, ElectronVolt, Minute, Hour, Day, Mile, Furlong, Chain, Rod, Fathom, Yard, Foot, Inch, Acre, GallonImperial, GallonUs, FluidOunceImperial, FluidOunceUs, PintImperial, PintUs, Ton, Hundredweight, stone, Pound, Ounce.

Custom Scaled Units

You may wish to add your own scaled units, that have not been included in Units.h. You can do this using the MAKE_SCALED_UNIT macro. The macro definition is

MAKE_SCALED_UNIT(CLASS_NAME, BASE_CLASS, BASE_CLASS_POWER, BASE_TO_SCALED_DIVIDER, SHORTNAME, LONGNAME)

You need to set CLASS_NAME to the name of the class you wish to create, BASE_CLASS is the name of the SI unit class which has the same dimensionality – this may be a compound unit (see below). BASE_CLASS_POWER is the power you wish to raise the base class to. For example for an area unit, you would set the BASE_CLASS to sci::Metre and the BASE_CLASS_POWER to 2. Base to scaled divider is the number you would divide the SI equivalent value by to get your unit. Or equivalently what 1 of your units would be when converted to SI units. For example a yard is 0.9144 metres, hence BASE_TO_SCALED_DIVIDER is 0.9144. SHORTNAME and LONGNAME are the short and long names for the unit. The long name is the name you would use in common speech. For example “yard” or “degree”. The short name is the abbreviation or symbol that would be used when writing out values. For example “yd” or “°”. You must include the quote marks around the names.

Unitless units

All maths within Units.h must take place between Physicals – raw floating point numbers cannot be used and dimensionless results are presented as Physicals too. A number of “unitless units” exist for use with dimensionless physicals. None have any template parameters.

sci::Unitless is the basic unitless class. The others unitless units are sci::Percent, sci::Permille and sci::BasisPoint, each respectively representing 100, 1000 and 10000 times a Unitless.

Combination Units

There are four combination units you can use. These are

```
template<class ENCODEDUNIT, int8_t POW_NUMERATOR, int8_t POW_DENOMINATOR>
struct PoweredUnit
```

```
template<class ENCODEDUNIT, int ROOT> struct RootedUnit
```

```
template<class ENCODEDUNIT1, class ENCODEDUNIT2> struct MultipliedUnit
```

```
template<class ENCODEDUNIT1, class ENCODEDUNIT2> struct DividedUnit
```

Powered unit provide a unit multiplied by a power. Unlike the power available as a template parameter in unit classes, this power can be a fraction. So for example to generate a metre to the power 3/2, you can use sci::PoweredUnit<sci::Metre<>, 3, 2>. One can create a rooted unit by either inverting the power in sci::PoweredUnit, or sci::Rooted unit is provided for convenience. For example to create a cube root of a metre you can use sci::RootedUnit<sci::Metre<>, 3>, which would be functionally equivalent to sci::PoweredUnit<sci::Metre<>, 1, 3>.

sci::MultipliedUnit allows two units to be multiplied together. These are provided as the two template parameters. So to create a newton metre you can use sci::MultipliedUnit<sci::Newton<>,sci::Metre<>>. If you wish to divide two units, sci::DividedUnit will divide ENCODEDUNIT1 by ENCODEDUNIT2. So, to create a metre per second, you can use sci::DividedUnit<sci::Metre<>, sci::Second<>>. Of course, this is again a convenience as you could also create a metre per second by multiplying a metre with a second to the power -1.

It is worth noting that when you are creating Physicals, you may find it easier to create combined units by using decltype along with standard operators. For example decltype(sci::Physical<sci::Metre<>,double>()/sci::Physical<sci::Second<>,double>()) will give you a

Physical with a metre per second unit. Using decltype or combination types directly is a matter of style only and the two should be functionally equivalent.

It is important to note that all units are functionally equivalent, however you create them. The only difference may be the floating point rounding or the unit string output. Values can always be converted between units provided the units have the same dimensionality – i.e. provided the powers of the 8 SI units are identical. For assignment and mathematical operations it does not matter if you create a metre squared by providing the power in the sci::Metre template parameters, using sci::MultipliedUnit with sci::Metre for both template parameters, using sci::PoweredUnit or using decltype. All will be equivalent classes which can be assigned to one another. The only difference will be in the output of the unit string.

Physical

Physical is where you will generally interact with the units. A Physical represents a value with a unit.

The class declaration is

```
template < class ENCODED_UNIT, class VALUE_TYPE> class Physical
```

The ENCODED_UNIT template parameter can be any of the unit types from the previous section. It can be SI units, scaled units or combined units. The VALUE_TYPE can be either double or float. In theory other parameters which act as numbers could be used in limited circumstances, but this is not a supported feature.

Physical contains two typedefs.

`sci::Physical::unit` provides the type of the unit used in the declaration.

`sci::Physical::valueType` provides the type of the value used in the declaration.

There are three constructors for Physical

`sci::Physical::Physical()` is the default constructor and does not initialise the value of the Physical.

`sci::Physical::Physical(VALUE_TYPE v)` constructs the Physical and assigns its value. This constructor is declared explicit so that the compiler cannot implicitly convert floating point values to Physicals. This avoids bugs where a value may be in the incorrect units when implicitly converted.

`template<class U, class V> sci::Physical:: Physical(const Physical<U, V> &other)` constructs a physical from another physical. The other Physical will be converted into the new units as required.

The following assignment operators are provided, =, +=, -=, *=, /=

These all behave in the expected manner. They require the other parameter to be another Physical (which may have a different VALUE_TYPE). The other parameter cannot be a raw floating point number. You must use one of the unitless units to utilise values with no units. Compile time checks are performed to ensure the two Physicals are compatible and conversion will take place before the operation. So for example you can assign a millimetre to a metre and the value will be converted from millimetres to metre as part of the assignment. As with built in assignment operators, these operators return a reference to the Physical. The += and -= operators will only work with the other Physical having the same dimensionality and the *= and /= operators will only work with the other Physical having a unitless unit. Anything else will give compilation errors.

There are two functions for retrieving the string representation of a Physical's unit

```
template<class STRING> STRING sci::Physical::getShortUnitString(const
STRING &exponentPrefix = STRING(), const STRING &exponentSuffix =
STRING())
```

```
template<class STRING> STRING sci::Physical::getLongUnitString()
```

The function getShortUnitString provides the string as it would be written in short notation, e.g. m s-1 or N m. The function getLongUnitString returns the unit as you would say it, e.g. metre per second or newton metre. The STRING template parameter may be std::string, std::wstring, std::u16string, std::u32string or std::basic_string<u8_type>. The function getShortUnitString accepts a prefix and suffix to go around exponents. These may be used to provide formatting flags to indicate the text is superscript. For getShortUnitString you may provide a prefix and suffix to bracket the power. This may help you to encode the string before passing it to a different library for parsing or displaying the text.

There is a function to retrieve the value of the physical

```
template <class REQUIRED> constexpr VALUE_TYPE sci::Physicalvalue() const
```

This function requires you to specify the required unit as the REQUIRED template parameter. Requiring the template parameter ensures any unit conversion is taken care of when the value is accessed and avoids bugs if the unit of the Physical is not recalled correctly by the user or is changed.

In order to check if two units are compatible you may use the function

```
template<class OTHER> static constexpr bool
sci::Physical::compatibleWith()
```

This will return true if the two units have the same dimensionality or false if not.

Operators

Physicals have overloads for the standard mathematical and comparison operators: +, -, *, /, (unary)-, >, <, >=, <=, ==, !=. When we use the standard mathematical operators we will get an appropriate Physical as output, i.e. when we multiply or divide we will get a Physical with a new unit and when we add or subtract a unit, there is a compile time check for unit compatibility and the result will have the same unit as the left hand Physical of the operator. Hence, 1 metre plus 100 millimetres will give a result of 1.1 metres. In addition type promotion is performed, so that if one physical uses a double a the value type and the other uses a float, the result will use a double.

Comparison operators also perform a compile time unit compatibility check and return a bool as expected. If the two Physicals do not have equivalent units, then Physical to the right of the operator is converted to that to the left of the operator before comparison. So 100 millimetres would be regarded as less than 1 metre, as you would expect.

Math functions

All the standard C++08 math functions are provided, but are within the sci namespace rather than the std namespace. These include sci::log, sci::log10, sci::exp, sci::sqrt, sci:sin, sci::cos, sci::tan, sci::asin, sci::acos, sci::atan, sci::atan2, sci::sinh, sci::cosh, sci::tanh, sci::abs. There is no version of

frexp and modf as these functions are considered nonsensical to perform on a Physical. The pow, ceil and floor functions are a separate case and are documented in the subsequent sections.

In addition to the functions above, Units.h provides a sci::ln function, which is functionally identical to sci::log, with a more intuitive name.

All the functions perform as would be expected – appropriate units must be used and will be returned. i.e. the inputs of sci::log, sci::log10, sci::ln, sci::exp, sci::asin, sci::acos and sci::atan, must be unitless, the two inputs of sci::atan2 must divide to give a unitless Physical, and the inputs of sci::sin, sci::cos, sci::tan, sci::sinh, sci::cosh and sci::tanh must have angular units. The output of sci::asin, sci::acos and sci::atan are Radians, but can trivially be converted to any other angular unit by assignment.

Power and root

Units.h provides both a power and root function. The sci::pow function, can accept a unitless base and a unitless power, giving a unitless result as one might expect. The sci::root function works in a similar manner. In addition, you may raise a non-unitless Physical to a power which is a rational fraction or find the equivalent root. These functions accept the numerator and denominator of the power/root as template parameters, this allows the return unit to be determined at compile time. So for example, you could do

```
sci::Physical<sci::Metre<>> length = 16.0;
auto parameter = sci::pow<2,3>(length);
std::cout << parameter;
```

Which would give the result something like

```
3.34960421 m2/3
```

The sci::root function works in a similar manner.

Ceil and Floor

When determining ceil and floor, the unit in which we are working needs to be specified. In the same way that sci::Physical::value, requires the user to be explicit with the unit required, so does sci::ceil and sci::floor. The unit is provided as a template parameter to the function. The output of these functions is a sci::Physical with the unit specified. For example

```
sci::Physical<sci::Metre<>> length = 16.347;
std::cout << sci::floor<sci::Metre<>>(length) << std::endl;
std::cout << sci::floor<sci::Metre<1, sci::centi>>(length) << std::endl;
```

would give output of

```
16 m
1637 cm
```

Stream Operators

As we have seen in the example code, there are overloads for the stream operators << and >>. The input stream operator will read a text representation of a number into your Physical. WARNING – this is the only time that you can convert a number to a Physical without explicitly stating the units of the number. The number will be assumed to be in the same units as the Physical being used. Some thought was given about simply omitting this operator to avoid possible bugs, but it seems too

useful to leave out and you will always require some unit checks or standards when reading from a stream anyway. When using the output operator, the value of the unit is written to the stream, followed by a space and then the short version of the unit.

UnitlessType struct

The UnitlessType<T> templated struct provides a way to get a unitless type for calculations involving either Physicals or raw doubles and floats.. For T set as any Physical, the typedef sci::UnitlessType<T>::type will be sci::Physical<sci::Unitless>. For any other type, such as a double or float sci::UnitlessType<T>::type will be the same as T. This is particularly useful when averaging. For example the templated function

```
template<class T>
T mean( T value1, T value2 )
{
	return (value1+value2)/T(2);
}
```

will work fine for doubles or floats, but would generally not compile if T was a sci::Physical. The issue is that the denominator is given a unit, when it should be unitless. To fix this, we can change the code to

```
template<class T>
T mean( T value1, T value2 )
{
	return (value1+value2)/sci::UnitlessType::type(2);
}
```

Numeric limits

We add a specialization of the std::numeric_limits class for sci::Physical. This behaves exactly how you would expect. So, for example, if you wish to create a NaN with the units of metre you can use the code

```
auto metreNan =
	std::numeric_limits<sci::Physical<sci::Metre<>>::quiet_NaN();
```

Note in this case we do use the std namespace, not the sci namespace. The helper functions sci::isnan, sci::isinf, sci::isfinite and sci::isnormal are included to check Physicals.