

---

# Boost.Units 1.1.0

Matthias C. Schabel

Steven Watanabe

Copyright © 2003 -2008 Matthias Christian Schabel, 2007-2010 Steven Watanabe

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE\_1\_0.txt or copy at [http://www.boost.org/LICENSE\\_1\\_0.txt](http://www.boost.org/LICENSE_1_0.txt))

## Table of Contents

Introduction .....	1
Quick Start .....	2
Dimensional Analysis .....	4
Units .....	6
Quantities .....	8
Examples .....	10
Utilities .....	39
Reference .....	40
Installation .....	496
FAQ .....	496
Acknowledgements .....	497
Help Wanted .....	498
Version Info .....	498
Release Notes .....	498
TODO .....	501

## Introduction

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. Support for units and quantities (defined as a unit and associated value) for arbitrary unit system models and arbitrary value types is provided, as is a fine-grained general facility for unit conversions. Complete SI and CGS unit systems are provided, along with systems for angles measured in degrees, radians, gradians, and revolutions and systems for temperatures measured in Kelvin, degrees Celsius and degrees Fahrenheit. The library architecture has been designed with flexibility and extensibility in mind; demonstrations of the ease of adding new units and unit conversions are provided in the examples.

In order to enable complex compile-time dimensional analysis calculations with no runtime overhead, Boost.Units relies heavily on the [Boost Metaprogramming Library](#) (MPL) and on template metaprogramming techniques, and is, as a consequence, fairly demanding of compiler compliance to ISO standards. At present, it has been successfully compiled and tested on the following compilers/platforms :

1. g++ 4.0.1 on Mac OSX 10.4
2. Intel CC 9.1, 10.0, and 10.1 on Mac OSX 10.4
3. g++ 3.4.4, 4.2.3, and 4.3.0 on Windows XP
4. Microsoft Visual C++ 7.1, 8.0, and 9.0 on Windows XP
5. Comeau 4.3.10.1 beta2 on Windows XP

6. Metrowerks CodeWarrior 9.2 on Windows XP.
7. Sun CC 5.9 on Solaris and Linux

The following compilers/platforms are known **not** to work :

1. g++ 3.3.x
2. Microsoft Visual C++ 6.0 on Windows XP
3. Microsoft Visual C++ 7.0 on Windows XP
4. Metrowerks CodeWarrior 8.0 on Windows XP.
5. All versions of Borland.

## Quick Start

Before discussing the basics of the library, we first define a few terms that will be used frequently in the following :

- **Base dimension** : A base dimension is loosely defined as a measurable entity of interest; in conventional dimensional analysis, base dimensions include length ([L]), mass ([M]), time ([T]), etc... but there is no specific restriction on what base dimensions can be used. Base dimensions are essentially a tag type and provide no dimensional analysis functionality themselves.
- **Dimension** : A collection of zero or more base dimensions, each potentially raised to a different rational power. For example,  $\text{length} = [L]^1$ ,  $\text{area} = [L]^2$ ,  $\text{velocity} = [L]^1/[T]^1$ , and  $\text{energy} = [M]^1 [L]^2/[T]^2$  are all dimensions.
- **Base unit** : A base unit represents a specific measure of a dimension. For example, while length is an abstract measure of distance, the meter is a concrete base unit of distance. Conversions are defined using base units. Much like base dimensions, base units are a tag type used solely to define units and do not support dimensional analysis algebra.
- **Unit** : A set of base units raised to rational exponents, e.g.  $\text{m}^1$ ,  $\text{kg}^1$ ,  $\text{m}^1/\text{s}^2$ .
- **System** : A unit system is a collection of base units representing all the measurable entities of interest for a specific problem. For example, the SI unit system defines seven base units : length ([L]) in meters, mass ([M]) in kilograms, time ([T]) in seconds, current ([I]) in amperes, temperature ([theta]) in kelvin, amount ([N]) in moles, and luminous intensity ([J]) in candelas. All measurable entities within the SI system can be represented as products of various integer or rational powers of these seven base units.
- **Quantity** : A quantity represents a concrete amount of a unit. Thus, while the meter is the base unit of length in the SI system, 5.5 meters is a quantity of length in that system.

To begin, we present two short tutorials. [Tutorial1](#) demonstrates the use of [SI](#) units. After including the appropriate system headers and the headers for the various SI units we will need (all SI units can be included with [boost/units/systems/si.hpp](#)) and for quantity I/O ([boost/units/io.hpp](#)), we define a function that computes the work, in joules, done by exerting a force in newtons over a specified distance in meters and outputs the result to `std::cout`. The [quantity](#) class accepts a second template parameter as its value type; this parameter defaults to `double` if not otherwise specified. To demonstrate the ease of using user-defined types in dimensional calculations, we also present code for computing the complex impedance using `std::complex<double>` as the value type :

```

#include <complex>
#include <iostream>

#include <boost/typeof/std/complex.hpp>

#include <boost/units/systems/si/energy.hpp>
#include <boost/units/systems/si/force.hpp>
#include <boost/units/systems/si/length.hpp>
#include <boost/units/systems/si/electric_potential.hpp>
#include <boost/units/systems/si/current.hpp>
#include <boost/units/systems/si/resistance.hpp>
#include <boost/units/systems/si/io.hpp>

using namespace boost::units;
using namespace boost::units::si;

quantity<energy>
work(const quantity<force>& F, const quantity<length>& dx)
{
    return F * dx; // Defines the relation: work = force * distance.
}

int main()
{
    /// Test calculation of work.
    quantity<force>      F(2.0 * newton); // Define a quantity of force.
    quantity<length>     dx(2.0 * meter); // and a distance,
    quantity<energy>     E(work(F,dx));  // and calculate the work done.

    std::cout << "F = " << F << std::endl
               << "dx = " << dx << std::endl
               << "E = " << E << std::endl
               << std::endl;

    /// Test and check complex quantities.
    typedef std::complex<double> complex_type; // double real and imaginary parts.

    // Define some complex electrical quantities.
    quantity<electric_potential, complex_type> v = complex_type(12.5, 0.0) * volts;
    quantity<current, complex_type> i = complex_type(3.0, 4.0) * amperes;
    quantity<resistance, complex_type> z = complex_type(1.5, -2.0) * ohms;

    std::cout << "V = " << v << std::endl
               << "I = " << i << std::endl
               << "Z = " << z << std::endl
               // Calculate from Ohm's law voltage = current * resistance.
               << "I * Z = " << i * z << std::endl
               // Check defined V is equal to calculated.
               << "I * Z == V? " << std::boolalpha << (i * z == v) << std::endl
               << std::endl;

    return 0;
}

```

The intent and function of the above code should be obvious; the output produced is :

```

F   = 2 N
dx  = 2 m
E   = 4 J

V   = (12.5, 0) V
I   = (3, 4) A
Z   = (1.5, -2) Ohm
I*Z = (12.5, 0) V
I*Z == V? true

```

While this library attempts to make simple dimensional computations easy to code, it is in no way tied to any particular unit system (SI or otherwise). Instead, it provides a highly flexible compile-time system for dimensional analysis, supporting arbitrary collections of base dimensions, rational powers of units, and explicit quantity conversions. It accomplishes all of this via template metaprogramming techniques. With modern optimizing compilers, this results in zero runtime overhead for quantity computations relative to the same code without unit checking.

## Dimensional Analysis

The concept of [dimensional analysis](#) is normally presented early on in introductory physics and engineering classes as a means of determining the correctness of an equation or computation by propagating the physical measurement [units](#) of various quantities through the equation along with their numerical values. There are a number of standard unit systems in common use, the most prominent of which is the [Système International](#) (also known as SI or MKS (meter-kilogram-second), which was a metric predecessor to the SI system named for three of the base units on which the system is based). The SI is the only official international standard unit system and is widely utilized in science and engineering. Other common systems include the [CGS](#) (centimeter-gram-second) system and the [English](#) system still in use in some problem domains in the United States and elsewhere. In physics, there also exist a number of other systems that are in common use in specialized subdisciplines. These are collectively referred to as [natural units](#). When quantities representing different measurables are combined, dimensional analysis provides the means of assessing the consistency of the resulting calculation. For example, the sum of two lengths is also a length, while the product of two lengths is an area, and the sum of a length and an area is undefined. The fact that the arguments to many functions (such as exp, log, etc...) must be dimensionless quantities can be easily demonstrated by examining their series expansions in the context of dimensional analysis. This library facilitates the enforcement of this type of restriction in code involving dimensioned quantities where appropriate.

In the following discussion we view dimensional analysis as an abstraction in which an arbitrary set of [units](#) obey the rules of a specific algebra. We will refer to a pair of a base dimension and a rational exponent as a **fundamental dimension**, and a list composed of an arbitrary number of fundamental dimensions as a **composite dimension** or, simply, **dimension**. In particular, given a set of  $p$  fundamental dimensions denoted by  $\{D_1, D_2, \dots, D_p\}$  and a set of  $p$  rational exponents  $\{R_1, R_2, \dots, R_p\}$ , any possible (composite) dimension can be written as  $D = \{D_1^{R_1}, D_2^{R_2}, \dots, D_p^{R_p}\}$ .

Composite dimensions obey the algebraic rules for dimensional analysis. In particular, for any scalar value,  $S$ , and composite dimensions  $D_x = \{\langle D_1, R_1 \rangle, \langle D_2, R_2 \rangle, \dots, \langle D_n, R_n \rangle\}$  and  $D_y = \{\langle D_1, R'_1 \rangle, \langle D_2, R'_2 \rangle, \dots, \langle D_m, R'_m \rangle\}$ , where  $n \leq m \leq p$ , we have:

$$\begin{aligned}
D_x + D_y &= D_x \quad \text{iff} \quad D_x = D_y \\
D_x - D_y &= D_x \quad \text{iff} \quad D_x = D_y \\
D_x \cdot D_y &= \{\langle D_1, R_1 + R'_1 \rangle, \langle D_2, R_2 + R'_2 \rangle, \dots, \langle D_n, R_n + R'_n \rangle, \langle D_{n+1}, R'_{n+1} \rangle, \dots, \langle D_m, R'_m \rangle\} \\
D_x / D_y &= \{\langle D_1, R_1 - R'_1 \rangle, \langle D_2, R_2 - R'_2 \rangle, \dots, \langle D_n, R_n - R'_n \rangle, \langle D_{n+1}, -R'_{n+1} \rangle, \dots, \langle D_m, -R'_m \rangle\} \\
D_x^S &= \{\langle D_1, S \cdot R_1 \rangle, \langle D_2, S \cdot R_2 \rangle, \dots, \langle D_n, S \cdot R_n \rangle\}
\end{aligned}$$

Users of a dimensional analysis library should be able to specify an arbitrary list of base dimensions to produce a composite dimension. This potentially includes repeated tags. For example, it should be possible to express energy as  $M \cdot L^2 / T^2$ ,  $M \cdot L / T \cdot L / T$ ,  $L / T \cdot M \cdot L / T$ , or any other permutation of mass, length, and time having aggregate exponents of 1, 2, and -2, respectively. In order to be able to perform computations on arbitrary sets of dimensions, all composite dimensions must be reducible to an unambiguous final composite dimension, which we will refer to as a **reduced dimension**, for which

1. fundamental dimensions are consistently ordered

2. dimensions with zero exponent are elided. Note that reduced dimensions never have more than  $P$  base dimensions, one for each distinct fundamental dimension, but may have fewer.

In our implementation, base dimensions are associated with tag types. As we will ultimately represent composite dimensions as typelists, we must provide some mechanism for sorting base dimension tags in order to make it possible to convert an arbitrary composite dimension into a reduced dimension. For this purpose, we assign a unique integer to each base dimension. The [base\\_dimension](#) class (found in [boost/units/base\\_dimension.hpp](#)) uses the curiously recurring template pattern (CRTP) technique to ensure that ordinals specified for base dimensions are unique:

```
template<class Derived, long N> struct base_dimension { ... };
```

With this, we can define the base dimensions for length, mass, and time as:

```
/// base dimension of length
struct length_base_dimension : base_dimension<length_base_dimension,1> { };
/// base dimension of mass
struct mass_base_dimension : base_dimension<mass_base_dimension,2> { };
/// base dimension of time
struct time_base_dimension : base_dimension<time_base_dimension,3> { };
```

It is important to note that the choice of order is completely arbitrary as long as each tag has a unique enumerable value; non-unique ordinals are flagged as errors at compile-time. Negative ordinals are reserved for use by the library. To define composite dimensions corresponding to the base dimensions, we simply create MPL-conformant typelists of fundamental dimensions by using the [dim](#) class to encapsulate pairs of base dimensions and [static\\_rational](#) exponents. The [make\\_dimension\\_list](#) class acts as a wrapper to ensure that the resulting type is in the form of a reduced dimension:

```
typedef make_dimension_list<
    boost::mpl::list< dim< length_base_dimension,static_rational<1> > >
>::type    length_dimension;

typedef make_dimension_list<
    boost::mpl::list< dim< mass_base_dimension,static_rational<1> > >
>::type    mass_dimension;

typedef make_dimension_list<
    boost::mpl::list< dim< time_base_dimension,static_rational<1> > >
>::type    time_dimension;
```

This can also be easily accomplished using a convenience typedef provided by [base\\_dimension](#):

```
typedef length_base_dimension::dimension_type    length_dimension;
typedef mass_base_dimension::dimension_type      mass_dimension;
typedef time_base_dimension::dimension_type      time_dimension;
```

so that the above code is identical to the full typelist definition. Composite dimensions are similarly defined via a typelist:

```
typedef make_dimension_list<
    boost::mpl::list< dim< length_base_dimension,static_rational<2> > >
>::type    area_dimension;

typedef make_dimension_list<
    boost::mpl::list< dim< mass_base_dimension,static_rational<1> >,
                    dim< length_base_dimension,static_rational<2> >,
                    dim< time_base_dimension,static_rational<-2> > >
>::type    energy_dimension;
```

A convenience class for composite dimensions with integer powers is also provided:

```
typedef derived_dimension<length_base_dimension,2>::type    area_dimension;
typedef derived_dimension<mass_base_dimension,1,
                        length_base_dimension,2,
                        time_base_dimension,-2>::type    energy_dimension;
```

## Units

We define a **unit** as a set of base units each of which can be raised to an arbitrary rational exponent. Thus, the SI unit corresponding to the dimension of force is kg m s<sup>-2</sup>, where kg, m, and s are base units. We use the notion of a **unit system** such as SI to specify the mapping from a dimension to a particular unit so that instead of specifying the base units explicitly, we can just ask for the representation of a dimension in a particular system.

Units are, like dimensions, purely compile-time variables with no associated value. Units obey the same algebra as dimensions do; the presence of the unit system serves to ensure that units having identical reduced dimension in different systems (like feet and meters) cannot be inadvertently mixed in computations.

There are two distinct types of systems that can be envisioned:

- **Homogeneous systems** : Systems which hold a linearly independent set of base units which can be used to represent many different dimensions. For example, the SI system has seven base dimensions and seven base units corresponding to them. It can represent any unit which uses only those seven base dimensions. Thus it is a homogeneous\_system.
- **Heterogeneous systems** : Systems which store the exponents of every base unit involved are termed heterogeneous. Some units can only be represented in this way. For example, area in m ft is intrinsically heterogeneous, because the base units of meters and feet have identical dimensions. As a result, simply storing a dimension and a set of base units does not yield a unique solution. A practical example of the need for heterogeneous units, is an empirical equation used in aviation:  $H = (r/C)^2$  where H is the radar beam height in feet and r is the radar range in nautical miles. In order to enforce dimensional correctness of this equation, the constant, C, must be expressed in nautical miles per foot<sup>(1/2)</sup>, mixing two distinct base units of length.

Units are implemented by the [unit](#) template class defined in [boost/units/unit.hpp](#) :

```
template<class Dim,class System> class unit;
```

In addition to supporting the compile-time dimensional analysis operations, the +, -, \*, and / runtime operators are provided for [unit](#) variables. Because the dimension associated with powers and roots must be computed at compile-time, it is not possible to provide overloads for `std::pow` that function correctly for [units](#). These operations are supported through free functions [pow](#) and [root](#) that are templated on integer and [static\\_rational](#) values and can take as an argument any type for which the utility classes [power\\_typeof\\_helper](#) and [root\\_typeof\\_helper](#) have been defined.

## Base Units

Base units are defined much like base dimensions.

```
template<class Derived, class Dimensions, long N> struct base_unit { ... };
```

Again negative ordinals are reserved.

As an example, in the following we will implement a subset of the SI unit system based on the fundamental dimensions given above, demonstrating all steps necessary for a completely functional system. First, we simply define a unit system that includes type definitions for commonly used units:

```
struct meter_base_unit : base_unit<meter_base_unit, length_dimension, 1> { };
struct kilogram_base_unit : base_unit<kilogram_base_unit, mass_dimension, 2> { };
struct second_base_unit : base_unit<second_base_unit, time_dimension, 3> { };

typedef make_system<
    meter_base_unit,
    kilogram_base_unit,
    second_base_unit>::type mks_system;

/// unit typedefs
typedef unit<dimensionless_type, mks_system>          dimensionless;

typedef unit<length_dimension, mks_system>            length;
typedef unit<mass_dimension, mks_system>              mass;
typedef unit<time_dimension, mks_system>              time;

typedef unit<area_dimension, mks_system>              area;
typedef unit<energy_dimension, mks_system>            energy;
```

The macro [BOOST\\_UNITS\\_STATIC\\_CONSTANT](#) is provided in [boost/units/static\\_constant.hpp](#) to facilitate ODR- and thread-safe constant definition in header files. We then define some constants for the supported units to simplify variable definitions:

```
/// unit constants
BOOST_UNITS_STATIC_CONSTANT(meter, length);
BOOST_UNITS_STATIC_CONSTANT(meters, length);
BOOST_UNITS_STATIC_CONSTANT(kilogram, mass);
BOOST_UNITS_STATIC_CONSTANT(kilograms, mass);
BOOST_UNITS_STATIC_CONSTANT(second, time);
BOOST_UNITS_STATIC_CONSTANT(seconds, time);

BOOST_UNITS_STATIC_CONSTANT(square_meter, area);
BOOST_UNITS_STATIC_CONSTANT(square_meters, area);
BOOST_UNITS_STATIC_CONSTANT(joule, energy);
BOOST_UNITS_STATIC_CONSTANT(joules, energy);
```

If support for textual output of units is desired, we can also specialize the [base\\_unit\\_info](#) class for each fundamental dimension tag:

```
template<> struct base_unit_info<test::meter_base_unit>
{
    static std::string name()          { return "meter"; }
    static std::string symbol()        { return "m"; }
};
```

and similarly for `kilogram_base_unit` and `second_base_unit`. A future version of the library will provide a more flexible system allowing for internationalization through a facet/locale-type mechanism. The `name()` and `symbol()` methods of [base\\_unit\\_info](#) provide full and short names for the base unit. With these definitions, we have the rudimentary beginnings of our unit system, which can be used to determine reduced dimensions for arbitrary unit calculations.

## Scaled Base Units

Now, it is also possible to define a base unit as being a multiple of another base unit. For example, the way that `kilogram_base_unit` is actually defined by the library is along the following lines

```
struct gram_base_unit : boost::units::base_unit<gram_base_unit, mass_dimension, 1> {};
typedef scaled_base_unit<gram_base_unit, scale<10, static_rational<3> > > kilogram_base_unit;
```

This basically defines a kilogram as being  $10^3$  times a gram.

There are several advantages to this approach.

- It reflects the real meaning of these units better than treating them as independent units.
- If a conversion is defined between grams or kilograms and some other units, it will automatically work for both kilograms and grams, with only one specialization.
- Similarly, if the symbol for grams is defined as "g", then the symbol for kilograms will be "kg" without any extra effort.

## Scaled Units

We can also scale a [unit](#) as a whole, rather than scaling the individual base units which comprise it. For this purpose, we use the metafunction [make\\_scaled\\_unit](#). The main motivation for this feature is the metric prefixes defined in [boost/units/systems/si/prefixes.hpp](#).

A simple example of its usage would be.

```
typedef make_scaled_unit<si::time, scale<10, static_rational<-9> > >::type nanosecond;
```

`nanosecond` is a specialization of [unit](#), and can be used in a quantity normally.

```
quantity<nanosecond> t(1.0 * si::seconds);
std::cout << t << std::endl; // prints 1e9 ns
```

## Quantities

A **quantity** is defined as a value of an arbitrary value type that is associated with a specific unit. For example, while meter is a unit, 3.0 meters is a quantity. Quantities obey two separate algebras: the native algebra for their value type, and the dimensional analysis algebra for the associated unit. In addition, algebraic operations are defined between units and quantities to simplify the definition of quantities; it is effectively equivalent to algebra with a unit-valued quantity.

Quantities are implemented by the [quantity](#) template class defined in [boost/units/quantity.hpp](#):

```
template<class Unit, class Y = double> class quantity;
```

This class is templated on both unit type (`Unit`) and value type (`Y`), with the latter defaulting to double-precision floating point if not otherwise specified. The value type must have a normal copy constructor and copy assignment operator. Operators `+`, `-`, `*`, and `/` are provided for algebraic operations between scalars and units, scalars and quantities, units and quantities, and between quantities. In addition, integral and rational powers and roots can be computed using the [pow<R>](#) and [root<R>](#) functions. Finally, the standard set of boolean comparison operators (`==`, `!=`, `<`, `<=`, `>`, and `>=`) are provided to allow comparison of quantities from the same unit system. All operators simply delegate to the corresponding operator of the value type if the units permit.



## Heterogeneous Operators

For most common value types, the result type of arithmetic operators is the same as the value type itself. For example, the sum of two double precision floating point numbers is another double precision floating point number. However, there are instances where this is not the case. A simple example is given by the [natural numbers](#) where the operator arithmetic obeys the following rules (using the standard notation for [number systems](#)):

- $N + N \rightarrow N$
- $N - N \rightarrow \mathbb{Z}$
- $N \cdot N \rightarrow N$
- $N/N \rightarrow \mathbb{Q}$

This library is designed to support arbitrary value type algebra for addition, subtraction, multiplication, division, and rational powers and roots. It uses Boost.Typeof to deduce the result of these operators. For compilers that support `typeof`, the appropriate value type will be automatically deduced. For compilers that do not provide language support for `typeof` it is necessary to register all the types used. For the case of natural numbers, this would amount to something like the following:

```
BOOST_TYPEOF_REGISTER_TYPE(natural);
BOOST_TYPEOF_REGISTER_TYPE(integer);
BOOST_TYPEOF_REGISTER_TYPE(rational);
```

## Conversions

Conversion is only meaningful for quantities as it implies the presence of at least a multiplicative scale factor and, possibly, and affine linear offset. Macros for simplifying the definition of conversions between units can be found in [boost/units/conversion.hpp](#) and [boost/units/absolute.hpp](#) (for affine conversions with offsets).

The macro [BOOST\\_UNITS\\_DEFINE\\_CONVERSION\\_FACTOR](#) specifies a scale factor for conversion from the first unit type to the second. The first argument must be a [base unit](#). The second argument can be either a [base unit](#) or a [unit](#).

Let's declare a simple base unit:

```
struct foot_base_unit : base_unit<foot_base_unit, length_dimension, 10> { };
```

Now, we want to be able to convert feet to meters and vice versa. The foot is defined as exactly 0.3048 meters, so we can write the following

```
BOOST_UNITS_DEFINE_CONVERSION_FACTOR(foot_base_unit, meter_base_unit, double, 0.3048);
```

Alternately, we could use the SI length typedef:

```
BOOST_UNITS_DEFINE_CONVERSION_FACTOR(foot_base_unit, SI::length, double, 0.3048);
```

Since the SI unit of length is the meter, these two definitions are equivalent. If these conversions have been defined, then converting between scaled forms of these units will also automatically work.

The macro [BOOST\\_UNITS\\_DEFAULT\\_CONVERSION](#) specifies a conversion that will be applied to a base unit when no direct conversion is possible. This can be used to make arbitrary conversions work with a single specialization:

```
struct my_unit_tag : boost::units::base_unit<my_unit_tag, boost::units::force_type, 1> {};  
// define the conversion factor  
BOOST_UNITS_DEFINE_CONVERSION_FACTOR(my_unit_tag, SI::force, double, 3.14159265358979323846);  
// make conversion to SI the default.  
BOOST_UNITS_DEFAULT_CONVERSION(my_unit_tag, SI::force);
```

## Construction and Conversion of Quantities

This library is designed to emphasize safety above convenience when performing operations with dimensioned quantities. Specifically, construction of quantities is required to fully specify both value and unit. Direct construction from a scalar value is prohibited (though the static member function [from\\_value](#) is provided to enable this functionality where it is necessary. In addition, a [quantity\\_cast](#) to a reference allows direct access to the underlying value of a [quantity](#) variable. An explicit constructor is provided to enable conversion between dimensionally compatible quantities in different unit systems. Implicit conversions between unit systems are allowed only when the reduced units are identical, allowing, for example, trivial conversions between equivalent units in different systems (such as SI seconds and CGS seconds) while simultaneously enabling unintentional unit system mismatches to be caught at compile time and preventing potential loss of precision and performance overhead from unintended conversions. Assignment follows the same rules. An exception is made for quantities for which the unit reduces to dimensionless; in this case, implicit conversion to the underlying value type is allowed via class template specialization. Quantities of different value types are implicitly convertible only if the value types are themselves implicitly convertible. The [quantity](#) class also defines a `value()` member for directly accessing the underlying value.

To summarize, conversions are allowed under the following conditions :

- implicit conversion of `quantity<Unit,Y>` to `quantity<Unit,Z>` is allowed if `Y` and `Z` are implicitly convertible.
- assignment between `quantity<Unit,Y>` and `quantity<Unit,Z>` is allowed if `Y` and `Z` are implicitly convertible.
- explicit conversion between `quantity<Unit1,Y>` and `quantity<Unit2,Z>` is allowed if `Unit1` and `Unit2` have the same dimensions and if `Y` and `Z` are implicitly convertible.
- implicit conversion between `quantity<Unit1,Y>` and `quantity<Unit2,Z>` is allowed if `Unit1` reduces to exactly the same combination of base units as `Unit2` and if `Y` and `Z` are convertible.
- assignment between `quantity<Unit1,Y>` and `quantity<Unit2,Z>` is allowed under the same conditions as implicit conversion.
- `quantity<Unit,Y>` can be directly constructed from a value of type `Y` using the static member function [from\\_value](#). Doing so, naturally, bypasses any type-checking of the newly assigned value, so this method should be used only when absolutely necessary.

Of course, any time implicit conversion is allowed, an explicit conversion is also legal.

Because dimensionless quantities have no associated units, they behave as normal scalars, and allow implicit conversion to and from the underlying value type or types that are convertible to/from that value type.

## Examples

### Dimension Example

([dimension.cpp](#))

By using MPL metafunctions and the template specializations for operations on composite dimensions (defined in [boost/units/dimension.hpp](#)) it is possible to perform compile time arithmetic according to the dimensional analysis rules described [above](#) to produce new composite dimensions :

```
typedef mpl::times<length_dimension, mass_dimension>::type    LM_type;
typedef mpl::divides<length_dimension, time_dimension>::type  L_T_type;
typedef static_root<
    mpl::divides<energy_dimension, mass_dimension>::type,
    static_rational<2>
>::type    V_type;
```

outputting (with symbol demangling, implemented in [utility.hpp](#))

```
length_dimension  = list<dim<length_base_dimension, static_rational<11, 11> >, dimensionless_type>
mass_dimension    = list<dim<mass_base_dimension, static_rational<11, 11> >, dimensionless_type>
time_dimension    = list<dim<time_base_dimension, static_rational<11, 11> >, dimensionless_type>
energy_dimension  = list<dim<length_base_dimension, static_rational<21, 11> >, list<dim<mass_base_dimension, static_rational<-21, 11> >, dimensionless_type> > >
LM_type           = list<dim<length_base_dimension, static_rational<11, 11> >, list<dim<mass_base_dimension, static_rational<11, 11> >, dimensionless_type> >
L_T_type          = list<dim<length_base_dimension, static_rational<11, 11> >, list<dim<time_base_dimension, static_rational<-11, 11> >, dimensionless_type> >
V_type            = list<dim<length_base_dimension, static_rational<11, 11> >, list<dim<time_base_dimension, static_rational<-11, 11> >, dimensionless_type> >
```

## Unit Example

([unit.cpp](#))

This example demonstrates the use of the simple but functional unit system implemented in [test\\_system.hpp](#)

```
const length      L;
const mass        M;
// needs to be namespace-qualified because of global time definition
const boost::units::test::time T;
const energy      E;
```

We can perform various algebraic operations on these units, resulting in the following output:

```
L                = m
L+L              = m
L-L              = m
L/L              = dimensionless
meter*meter      = m^2
M*(L/T)*(L/T)    = m^2 kg s^-2
M*(L/T)^2        = m^2 kg s^-2
L^3              = m^3
L^(3/2)          = m^(3/2)
2vM              = kg^(1/2)
(3/2)vM          = kg^(2/3)
```

## Quantity Example

([quantity.cpp](#))

This example demonstrates how to use quantities of our toy unit system :

```
quantity<length> L = 2.0*meters;           // quantity of length
quantity<energy> E = kilograms*pow<2>(L/seconds); // quantity of energy
```

giving us the basic quantity functionality :

```
L                = 2 m
L+L              = 4 m
L-L              = 0 m
L*L              = 4 m^2
L/L              = 1 dimensionless
L*meter          = 2 m^2
kilograms*(L/seconds)*(L/seconds) = 4 m^2 kg s^-2
kilograms*(L/seconds)^2      = 4 m^2 kg s^-2
L^3               = 8 m^3
L^(3/2)           = 2.82843 m^(3/2)
2vL               = 1.41421 m^(1/2)
(3/2)vL           = 1.5874 m^(2/3)
```

As a further demonstration of the flexibility of the system, we replace the double value type with a `std::complex<double>` value type (ignoring the question of the meaningfulness of complex lengths and energies) :

```
quantity<length,std::complex<double> > L(std::complex<double>(3.0,4.0)*meters);
quantity<energy,std::complex<double> > E(kilograms*pow<2>(L/seconds));
```

and find that the code functions exactly as expected with no additional work, delegating operations to `std::complex<double>` and performing the appropriate dimensional analysis :

```
L                = (3,4) m
L+L              = (6,8) m
L-L              = (0,0) m
L*L              = (-7,24) m^2
L/L              = (1,0) dimensionless
L*meter          = (3,4) m^2
kilograms*(L/seconds)*(L/seconds) = (-7,24) m^2 kg s^-2
kilograms*(L/seconds)^2      = (-7,24) m^2 kg s^-2
L^3               = (-117,44) m^3
L^(3/2)           = (2,11) m^(3/2)
2vL               = (2,1) m^(1/2)
(3/2)vL           = (2.38285,1.69466) m^(2/3)
```

## Kitchen Sink Example using SI units

([kitchen\\_sink.cpp](#))

This example provides a fairly extensive set of tests covering most of the [quantity](#) functionality. It uses the SI unit system defined in [boost/units/systems/si.hpp](#).

If we define a few units and associated quantities,

```

/// scalar
const double    s1 = 2;

const long      x1 = 2;
const static_rational<4,3> x2;

/// define some units
force          u1 = newton;
energy         u2 = joule;

/// define some quantities
quantity<force>    q1(1.0*u1);
quantity<energy>   q2(2.0*u2);

```

the various algebraic operations between scalars, units, and quantities give

```

S1 :      2
X1 :      2
X2 :      (4/3)
U1 :      N
U2 :      J
Q1 :      1 N
Q2 :      2 J

```

Scalar/unit operations :

```

U1*S1 : 2 N
S1*U1 : 2 N
U1/S1 : 0.5 N
S1/U1 : 2 m^-1 kg^-1 s^2

```

Unit/unit operations and integral/rational powers of units :

```

U1+U1 : N
U1-U1 : N
U1*U1 : m^2 kg^2 s^-4
U1/U1 : dimensionless
U1*U2 : m^3 kg^2 s^-4
U1/U2 : m^-1
U1^X   : m^2 kg^2 s^-4
X1vU1 : m^(1/2) kg^(1/2) s^-1
U1^X2  : m^(4/3) kg^(4/3) s^(-8/3)
X2vU1 : m^(3/4) kg^(3/4) s^(-3/2)

```

Scalar/quantity operations :

```

Q1*S1 : 2 N
S1*Q1 : 2 N
Q1/S1 : 0.5 N
S1/Q1 : 2 m^-1 kg^-1 s^2

```

Unit/quantity operations :

```

U1*Q1 : 1 m^2 kg^2 s^-4
Q1*U1 : 1 m^2 kg^2 s^-4
U1/Q1 : 1 dimensionless
Q1/U1 : 1 dimensionless

```

Quantity/quantity operations and integral/rational powers of quantities :

```

+Q1      : 1 N
-Q1      : -1 N
Q1+Q1    : 2 N
Q1-Q1    : 0 N
Q1*Q1    : 1 m^2 kg^2 s^-4
Q1/Q1    : 1 dimensionless
Q1*Q2    : 2 m^3 kg^2 s^-4
Q1/Q2    : 0.5 m^-1
Q1^X1    : 1 m^2 kg^2 s^-4
X1vQ1    : 1 m^(1/2) kg^(1/2) s^-1
Q1^X2    : 1 m^(4/3) kg^(4/3) s^(-8/3)
X2vQ1    : 1 m^(3/4) kg^(3/4) s^(-3/2)

```

Logical comparison operators are also defined between quantities :

```

/// check comparison tests
quantity<length>    l1(1.0*meter),
                    l2(2.0*meters);

```

giving

```

l1 == l2    false
l1 != l2    true
l1 <= l2    true
l1 < l2     true
l1 >= l2    false
l1 > l2     false

```

Implicit conversion is allowed between dimensionless quantities and their corresponding value types :

```

/// check implicit unit conversion from dimensionless to value_type
const double    dimless = (q1/q1);

```

A generic function for computing mechanical work can be defined that takes force and distance arguments in an arbitrary unit system and returns energy in the same system:

```

/// the physical definition of work - computed for an arbitrary unit system
template<class System, class Y>
quantity<unit<energy_dimension, System>, Y>
work(quantity<unit<force_dimension, System>, Y> F,
      quantity<unit<length_dimension, System>, Y> dx)
{
    return F*dx;
}

```

```

/// test calculation of work
quantity<force>      F(1.0*newton);
quantity<length>     dx(1.0*meter);
quantity<energy>     E(work(F,dx));

```

which functions as expected for SI quantities :

```

F  = 1 N
dx = 1 m
E  = 1 J

```

The ideal gas law can also be implemented in SI units :

```

/// the ideal gas law in si units
template<class Y>
quantity<si::amount, Y>
idealGasLaw(const quantity<si::pressure, Y>& P,
            const quantity<si::volume, Y>& V,
            const quantity<si::temperature, Y>& T)
{
    using namespace boost::units::si;

    using namespace constants::codata;
    return (P*V/(R*T));
}

```

```

/// test ideal gas law
quantity<temperature> T = (273.+37.)*kelvin;
quantity<pressure>    P = 1.01325e5*pascals;
quantity<length>      r = 0.5e-6*meters;
quantity<volume>       V = (4.0/3.0)*3.141592*pow<3>(r);
quantity<amount>       n(idealGasLaw(P,V,T));

```

with the resulting output :

```

r = 5e-07 m
P = 101325 Pa
V = 5.23599e-19 m^3
T = 310 K
n = 2.05835e-17 mol
R = 8.314472 m^2 kg s^-2 K^-1 mol^-1 (rel. unc. = 1.8e-06)

```

Trigonometric and inverse trigonometric functions can be implemented for any unit system that provides an angular base dimension. For radians, these functions are found in [boost/units/cmath.hpp](#). These behave as one expects, with trigonometric functions taking an angular quantity and returning a dimensionless quantity, while the inverse trigonometric functions take a dimensionless quantity and return an angular quantity :

Defining a few angular quantities,

```

/// test trig stuff
quantity<plane_angle>          theta = 0.375*radians;
quantity<dimensionless>       sin_theta = sin(theta);
quantity<plane_angle>          thetap = asin(sin_theta);

```

yields

```

theta          = 0.375 rd
sin(theta)     = 0.366273 dimensionless
asin(sin(theta)) = 0.375 rd

```

Dealing with complex quantities is trivial. Here is the calculation of complex impedance :

```

quantity<electric_potential,complex_type> v = complex_type(12.5,0.0)*volts;
quantity<current,complex_type>            i = complex_type(3.0,4.0)*amperes;
quantity<resistance,complex_type>         z = complex_type(1.5,-2.0)*ohms;

```

giving

```

V   = (12.5,0) V
I   = (3,4) A
Z   = (1.5,-2) Ohm
I*Z = (12.5,0) V

```

## User-defined value types

User-defined value types that support the appropriate arithmetic operations are automatically supported as quantity value types. The operators that are supported by default for quantity value types are unary plus, unary minus, addition, subtraction, multiplication, division, equal-to, not-equal-to, less-than, less-or-equal-to, greater-than, and greater-or-equal-to. Support for rational powers and roots can be added by overloading the [power\\_typeof\\_helper](#) and [root\\_typeof\\_helper](#) classes. Here we implement a user-defined measurement class that models a numerical measurement with an associated measurement error and the appropriate algebra and demonstrates its use as a quantity value type; the full code is found in [measurement.hpp](#).

Then, defining some measurement [quantity](#) variables

```

quantity<length,measurement<double> >
    u(measurement<double>(1.0,0.0)*meters),
    w(measurement<double>(4.52,0.02)*meters),
    x(measurement<double>(2.0,0.2)*meters),
    y(measurement<double>(3.0,0.6)*meters);

```

gives

```

x+y-w      = 0.48(+/-0.632772) m
w*x        = 9.04(+/-0.904885) m^2
x/y        = 0.666667(+/-0.149071) dimensionless

```

If we implement the overloaded helper classes for rational powers and roots then we can also compute rational powers of measurement quantities :



```
w*y^2/(u*x)^2 = 10.17(+/-3.52328) m^-1
w/(u*x)^(1/2) = 3.19612(+/-0.160431) dimensionless
```

## Conversion Example

([conversion.cpp](#))

This example demonstrates the various allowed conversions between SI and CGS units. Defining some quantities

```
quantity<si::length>      L1 = quantity<si::length,int>(int(2.5)*si::meters);
quantity<si::length,int>  L2(quantity<si::length,double>(2.5*si::meters));
```

illustrates implicit conversion of quantities of different value types where implicit conversion of the value types themselves is allowed. N.B. The conversion from double to int is treated as an explicit conversion because there is no way to emulate the exact behavior of the built-in conversion. Explicit constructors allow conversions for two cases:

- explicit casting of a [quantity](#) to a different value\_type :

```
quantity<si::length,int> L3 = static_cast<quantity<si::length,int>>(L1);
```

- and explicit casting of a [quantity](#) to a different unit :

```
quantity<cgs::length>     L4 = static_cast<quantity<cgs::length>>(L1);
```

giving the following output :

```
L1 = 2 m
L2 = 2 m
L3 = 2 m
L4 = 200 cm
L5 = 5 m
L6 = 4 m
L7 = 200 cm
```

A few more explicit unit system conversions :

```
quantity<si::volume>      vs(1.0*pow<3>(si::meter));
quantity<cgs::volume>     vc(vs);
quantity<si::volume>      vs2(vc);

quantity<si::energy>      es(1.0*si::joule);
quantity<cgs::energy>     ec(es);
quantity<si::energy>      es2(ec);

quantity<si::velocity>    v1 = 2.0*si::meters/si::second,
                           v2(2.0*cgs::centimeters/cgs::second);
```

which produces the following output:

```

volume (m^3)   = 1 m^3
volume (cm^3)  = 1e+06 cm^3
volume (m^3)   = 1 m^3

energy (joules) = 1 J
energy (ergs)   = 1e+07 erg
energy (joules) = 1 J

velocity (2 m/s) = 2 m s^-1
velocity (2 cm/s) = 0.02 m s^-1

```

## User Defined Types

([quaternion.cpp](#))

This example demonstrates the use of `boost::math::quaternion` as a value type for [quantity](#) and the converse. For the first case, we first define specializations of [power\\_typeof\\_helper](#) and [root\\_typeof\\_helper](#) for powers and roots, respectively:

```

/// specialize power_typeof helper
template<class Y, long N, long D>
struct power_typeof_helper<boost::math::quaternion<Y>, static_rational<N, D> >
{
    // boost::math::quaternion only supports integer powers
    BOOST_STATIC_ASSERT(D==1);

    typedef boost::math::quaternion<
        typename power_typeof_helper<Y, static_rational<N, D> >::type
    > type;

    static type value(const boost::math::quaternion<Y>& x)
    {
        return boost::math::pow(x, static_cast<int>(N));
    }
};

```

```

/// specialize root_typeof helper
template<class Y, long N, long D>
struct root_typeof_helper<boost::math::quaternion<Y>, static_rational<N, D> >
{
    // boost::math::quaternion only supports integer powers
    BOOST_STATIC_ASSERT(N==1);

    typedef boost::math::quaternion<
        typename root_typeof_helper<Y, static_rational<N, D> >::type
    > type;

    static type value(const boost::math::quaternion<Y>& x)
    {
        return boost::math::pow(x, static_cast<int>(D));
    }
};

```

We can now declare a [quantity](#) of a quaternion:

```
typedef quantity<length, quaternion<double> >      length_dimension;  
  
length_dimension    L(quaternion<double>(4.0, 3.0, 2.0, 1.0)*meters);
```

so that all operations that are defined in the `quaternion` class behave correctly. If rational powers were defined for this class, it would be possible to compute rational powers and roots with no additional changes.

```
+L      = (4, 3, 2, 1) m  
-L      = (-4, -3, -2, -1) m  
L+L     = (8, 6, 4, 2) m  
L-L     = (0, 0, 0, 0) m  
L*L     = (2, 24, 16, 8) m^2  
L/L     = (1, 0, 0, 0) dimensionless  
L^3     = (-104, 102, 68, 34) m^3
```

Now, if for some reason we preferred the [quantity](#) to be the value type of the `quaternion` class we would have :

```
typedef quaternion<quantity<length> >      length_dimension;  
  
length_dimension    L(4.0*meters, 3.0*meters, 2.0*meters, 1.0*meters);
```

Here, the unary plus and minus and addition and subtraction operators function correctly. Unfortunately, the multiplication and division operations fail because `quaternion` implements them in terms of the `*=` and `/=` operators, respectively, which are incapable of representing the heterogeneous unit algebra needed for quantities (an identical problem occurs with `std::complex<T>`, for the same reason). In order to compute rational powers and roots, we need to specialize [power\\_typeof\\_helper](#) and [root\\_typeof\\_helper](#) as follows:

```

/// specialize power typeof helper for quaternion<quantity<Unit,Y> >
template<class Unit,long N,long D,class Y>
struct power_typeof_helper<
    boost::math::quaternion<quantity<Unit,Y> >,
    static_rational<N,D> >
{
    typedef typename power_typeof_helper<
        Y,
        static_rational<N,D>
    >::type value_type;

    typedef typename power_typeof_helper<
        Unit,
        static_rational<N,D>
    >::type unit_type;

    typedef quantity<unit_type,value_type> quantity_type;
    typedef boost::math::quaternion<quantity_type> type;

    static type value(const boost::math::quaternion<quantity<Unit,Y> >& x)
    {
        const boost::math::quaternion<value_type> tmp =
            pow<static_rational<N,D> >(boost::math::quaternion<Y>(
                x.R_component_1().value(),
                x.R_component_2().value(),
                x.R_component_3().value(),
                x.R_component_4().value()));

        return type(quantity_type::from_value(tmp.R_component_1()),
                    quantity_type::from_value(tmp.R_component_2()),
                    quantity_type::from_value(tmp.R_component_3()),
                    quantity_type::from_value(tmp.R_component_4()));
    }
};

```

```

/// specialize root typeof helper for quaternion<quantity<Unit,Y> >
template<class Unit,long N,long D,class Y>
struct root_typeof_helper<
    boost::math::quaternion<quantity<Unit,Y> >,
    static_rational<N,D> >
{
    typedef typename root_typeof_helper<
        Y,
        static_rational<N,D>
    >::type value_type;

    typedef typename root_typeof_helper<
        Unit,
        static_rational<N,D>
    >::type unit_type;

    typedef quantity<unit_type,value_type> quantity_type;
    typedef boost::math::quaternion<quantity_type> type;

    static type value(const boost::math::quaternion<quantity<Unit,Y> >& x)
    {
        const boost::math::quaternion<value_type> tmp =
            root<static_rational<N,D> >(boost::math::quaternion<Y>(
                x.R_component_1().value(),
                x.R_component_2().value(),
                x.R_component_3().value(),
                x.R_component_4().value()));
    }
};

```

```

    return type(quantity_type::from_value(tmp.R_component_1()),
               quantity_type::from_value(tmp.R_component_2()),
               quantity_type::from_value(tmp.R_component_3()),
               quantity_type::from_value(tmp.R_component_4()));
}
};

```

giving:

```

+L      = ( 4 m, 3 m, 2 m, 1 m)
-L      = (-4 m, -3 m, -2 m, -1 m)
L+L     = ( 8 m, 6 m, 4 m, 2 m)
L-L     = ( 0 m, 0 m, 0 m, 0 m)
L^3     = (-104 m^3, 102 m^3, 68 m^3, 34 m^3)

```

## Complex Example

([complex.cpp](#))

This example demonstrates how to implement a replacement `complex` class that functions correctly both as a quantity value type and as a quantity container class, including heterogeneous multiplication and division operations and rational powers and roots. Naturally, heterogeneous operations are only supported on compilers that implement `typeof`. The primary differences are that binary operations are not implemented using the `op=` operators and use the utility classes [add\\_typeof\\_helper](#), [subtract\\_typeof\\_helper](#), [multiply\\_typeof\\_helper](#), and [divide\\_typeof\\_helper](#). In addition, [power\\_typeof\\_helper](#) and [root\\_typeof\\_helper](#) are defined for both cases :

```

namespace boost {

namespace units {

/// replacement complex class
template<class T>
class complex
{
public:
    typedef complex<T>    this_type;

    complex(const T& r = 0, const T& i = 0) : r_(r), i_(i) { }
    complex(const this_type& source) : r_(source.r_), i_(source.i_) { }

    this_type& operator=(const this_type& source)
    {
        if (this == &source) return *this;

        r_ = source.r_;
        i_ = source.i_;

        return *this;
    }

    T& real()          { return r_; }
    T& imag()          { return i_; }

    const T& real() const { return r_; }
    const T& imag() const { return i_; }

    this_type& operator+=(const T& val)
    {
        r_ += val;
        return *this;
    }

    this_type& operator-=(const T& val)
    {
        r_ -= val;
        return *this;
    }

    this_type& operator*=(const T& val)
    {
        r_ *= val;
        i_ *= val;
        return *this;
    }

    this_type& operator/=(const T& val)
    {
        r_ /= val;
        i_ /= val;
        return *this;
    }

    this_type& operator+=(const this_type& source)
    {
        r_ += source.r_;
        i_ += source.i_;
        return *this;
    }
}

```

```

    this_type& operator--=(const this_type& source)
    {
        r_ -= source.r_;
        i_ -= source.i_;
        return *this;
    }

    this_type& operator*=(const this_type& source)
    {
        *this = *this * source;
        return *this;
    }

    this_type& operator/=(const this_type& source)
    {
        *this = *this / source;
        return *this;
    }

private:
    T    r_, i_;
};

}

}

#if BOOST_UNITS_HAS_BOOST_TYPEOF
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

BOOST_TYPEOF_REGISTER_TEMPLATE(boost::units::complex, 1)

#endif

namespace boost {

namespace units {

template<class X>
complex<typename unary_plus_typeof_helper<X>::type>
operator+(const complex<X>& x)
{
    typedef typename unary_plus_typeof_helper<X>::type type;

    return complex<type>(x.real(), x.imag());
}

template<class X>
complex<typename unary_minus_typeof_helper<X>::type>
operator-(const complex<X>& x)
{
    typedef typename unary_minus_typeof_helper<X>::type type;

    return complex<type>(-x.real(), -x.imag());
}

template<class X, class Y>
complex<typename add_typeof_helper<X, Y>::type>
operator+(const complex<X>& x, const complex<Y>& y)
{
    typedef typename boost::units::add_typeof_helper<X, Y>::type type;

```

```

    return complex<type>(x.real()+y.real(),x.imag()+y.imag());
}

template<class X,class Y>
complex<typename boost::units::subtract_typeof_helper<X,Y>::type>
operator-(const complex<X>& x,const complex<Y>& y)
{
    typedef typename boost::units::subtract_typeof_helper<X,Y>::type    type;

    return complex<type>(x.real()-y.real(),x.imag()-y.imag());
}

template<class X,class Y>
complex<typename boost::units::multiply_typeof_helper<X,Y>::type>
operator*(const complex<X>& x,const complex<Y>& y)
{
    typedef typename boost::units::multiply_typeof_helper<X,Y>::type    type;

    return complex<type>(x.real()*y.real() - x.imag()*y.imag(),
                        x.real()*y.imag() + x.imag()*y.real());
}

// fully correct implementation has more complex return type
//
//     typedef typename boost::units::multiply_typeof_helper<X,Y>::type xy_type;
//
//     typedef typename boost::units::add_typeof_helper<
//         xy_type,xy_type>::type          xy_plus_xy_type;
//     typedef typename
//         boost::units::subtract_typeof_helper<xy_type,xy_type>::type
//         xy_minus_xy_type;
//
//     BOOST_STATIC_ASSERT((boost::is_same<xy_plus_xy_type,
//                                     xy_minus_xy_type>::value == true));
//
//     return complex<xy_plus_xy_type>(x.real()*y.real()-x.imag()*y.imag(),
//                                     x.real()*y.imag()+x.imag()*y.real());
//
}

template<class X,class Y>
complex<typename boost::units::divide_typeof_helper<X,Y>::type>
operator/(const complex<X>& x,const complex<Y>& y)
{
    // naive implementation of complex division
    typedef typename boost::units::divide_typeof_helper<X,Y>::type type;

    return complex<type>((x.real()*y.real()+x.imag()*y.imag())/
                        (y.real()*y.real()+y.imag()*y.imag()),
                        (x.imag()*y.real()-x.real()*y.imag())/
                        (y.real()*y.real()+y.imag()*y.imag()));
}

// fully correct implementation has more complex return type
//
//     typedef typename boost::units::multiply_typeof_helper<X,Y>::type xy_type;
//     typedef typename boost::units::multiply_typeof_helper<Y,Y>::type yy_type;
//
//     typedef typename boost::units::add_typeof_helper<xy_type, xy_type>::type
//         xy_plus_xy_type;
//     typedef typename boost::units::subtract_typeof_helper<
//         xy_type,xy_type>::type xy_minus_xy_type;
//
//     typedef typename boost::units::divide_typeof_helper<
//         xy_plus_xy_type,yy_type>::type          xy_plus_xy_over_yy_type;
//     typedef typename boost::units::divide_typeof_helper<

```



```

//      xy_minus_xy_type,yy_type>::type      xy_minus_xy_over_yy_type;
//
// BOOST_STATIC_ASSERT((boost::is_same<xy_plus_xy_over_yy_type,
//                                     xy_minus_xy_over_yy_type>::value == true));
//
// return complex<xy_plus_xy_over_yy_type>(
//     (x.real()*y.real()+x.imag()*y.imag())/
//     (y.real()*y.real()+y.imag()*y.imag()),
//     (x.imag()*y.real()-x.real()*y.imag())/
//     (y.real()*y.real()+y.imag()*y.imag()));
// }

template<class Y>
complex<Y>
pow(const complex<Y>& x,const Y& y)
{
    std::complex<Y> tmp(x.real(),x.imag());

    tmp = std::pow(tmp,y);

    return complex<Y>(tmp.real(),tmp.imag());
}

template<class Y>
std::ostream& operator<< (std::ostream& os,const complex<Y>& val)
{
    os << val.real() << " + " << val.imag() << " i";

    return os;
}

/// specialize power typeof helper for complex<Y>
template<class Y,long N,long D>
struct power_typeof_helper<complex<Y>,static_rational<N,D> >
{
    typedef complex<
        typename power_typeof_helper<Y,static_rational<N,D> >::type
    > type;

    static type value(const complex<Y>& x)
    {
        const static_rational<N,D> rat;

        const Y    m = Y(rat.numerator())/Y(rat.denominator());

        return boost::units::pow(x,m);
    }
};

/// specialize root typeof helper for complex<Y>
template<class Y,long N,long D>
struct root_typeof_helper<complex<Y>,static_rational<N,D> >
{
    typedef complex<
        typename root_typeof_helper<Y,static_rational<N,D> >::type
    > type;

    static type value(const complex<Y>& x)
    {
        const static_rational<N,D> rat;

        const Y    m = Y(rat.denominator())/Y(rat.numerator());

```

```

        return boost::units::pow(x,m);
    }
};

/// specialize power typeof helper for complex<quantity<Unit,Y> >
template<class Y,class Unit,long N,long D>
struct power_typeof_helper<complex<quantity<Unit,Y> >,static_rational<N,D> >
{
    typedef typename
        power_typeof_helper<Y,static_rational<N,D> >::type        value_type;
    typedef typename
        power_typeof_helper<Unit,static_rational<N,D> >::type    unit_type;
    typedef quantity<unit_type,value_type>                        quantity_type;
    typedef complex<quantity_type>                                type;

    static type value(const complex<quantity<Unit,Y> >& x)
    {
        const complex<value_type> tmp =
            pow<static_rational<N,D> >(complex<Y>(x.real().value(),
                                                    x.imag().value()));

        return type(quantity_type::from_value(tmp.real()),
                    quantity_type::from_value(tmp.imag()));
    }
};

/// specialize root typeof helper for complex<quantity<Unit,Y> >
template<class Y,class Unit,long N,long D>
struct root_typeof_helper<complex<quantity<Unit,Y> >,static_rational<N,D> >
{
    typedef typename
        root_typeof_helper<Y,static_rational<N,D> >::type        value_type;
    typedef typename
        root_typeof_helper<Unit,static_rational<N,D> >::type    unit_type;
    typedef quantity<unit_type,value_type>                        quantity_type;
    typedef complex<quantity_type>                                type;

    static type value(const complex<quantity<Unit,Y> >& x)
    {
        const complex<value_type> tmp =
            root<static_rational<N,D> >(complex<Y>(x.real().value(),
                                                    x.imag().value()));

        return type(quantity_type::from_value(tmp.real()),
                    quantity_type::from_value(tmp.imag()));
    }
};

} // namespace units

} // namespace boost

```

With this replacement complex class, we can declare a complex variable :

```

typedef quantity<length,complex<double> >    length_dimension;

length_dimension    L(complex<double>(2.0,1.0)*meters);

```

to get the correct behavior for all cases supported by [quantity](#) with a complex value type :

```

+L      = 2 + 1 i m
-L      = -2 + -1 i m
L+L     = 4 + 2 i m
L-L     = 0 + 0 i m
L*L     = 3 + 4 i m^2
L/L     = 1 + 0 i dimensionless
L^3     = 2 + 11 i m^3
L^(3/2) = 2.56713 + 2.14247 i m^(3/2)
3vL     = 1.29207 + 0.201294 i m^(1/3)
(3/2)vL = 1.62894 + 0.520175 i m^(2/3)

```

and, similarly, complex with a [quantity](#) value type

```

typedef complex<quantity<length> >      length_dimension;

length_dimension    L(2.0*meters,1.0*meters);

```

gives

```

+L      = 2 m + 1 m i
-L      = -2 m + -1 m i
L+L     = 4 m + 2 m i
L-L     = 0 m + 0 m i
L*L     = 3 m^2 + 4 m^2 i
L/L     = 1 dimensionless + 0 dimensionless i
L^3     = 2 m^3 + 11 m^3 i
L^(3/2) = 2.56713 m^(3/2) + 2.14247 m^(3/2) i
3vL     = 1.29207 m^(1/3) + 0.201294 m^(1/3) i
(3/2)vL = 1.62894 m^(2/3) + 0.520175 m^(2/3) i

```

## Performance Example

([performance.cpp](#))

This example provides an ad hoc performance test to verify that zero runtime overhead is incurred when using [quantity](#) in place of `double`. Note that performance optimization and testing is not trivial, so some care must be taken in profiling. It is also critical to have a compiler capable of optimizing the many template instantiations and inline calls effectively to achieve maximal performance. Zero overhead for this test has been verified using gcc 4.0.1, and icc 9.0, 10.0, and 10.1 on Mac OS 10.4 and 10.5, and using msvc 8.0 on Windows XP.

## Radar Beam Height

([radar\\_beam\\_height.cpp](#))

This example demonstrates the implementation of two non-SI units of length, the nautical mile :

```

namespace nautical {

struct length_base_unit :
    boost::units::base_unit<length_base_unit, length_dimension, 1>
{
    static std::string name()          { return "nautical mile"; }
    static std::string symbol()        { return "nmi"; }
};

typedef boost::units::make_system<length_base_unit>::type system;

/// unit typedefs
typedef unit<length_dimension,system>    length;

static const length mile,miles;

} // namespace nautical

// helper for conversions between nautical length and si length
BOOST_UNITS_DEFINE_CONVERSION_FACTOR(nautical::length_base_unit,
                                     boost::units::si::meter_base_unit,
                                     double, 1.852e3);

```

and the imperial foot :

```

namespace imperial {

struct length_base_unit :
    boost::units::base_unit<length_base_unit, length_dimension, 2>
{
    static std::string name()          { return "foot"; }
    static std::string symbol()        { return "ft"; }
};

typedef boost::units::make_system<length_base_unit>::type system;

/// unit typedefs
typedef unit<length_dimension,system>    length;

static const length foot,feet;

} // imperial

// helper for conversions between imperial length and si length
BOOST_UNITS_DEFINE_CONVERSION_FACTOR(imperial::length_base_unit,
                                     boost::units::si::meter_base_unit,
                                     double, 1.0/3.28083989501312);

```

These units include conversions between themselves and the meter. Three functions for computing radar beam height from radar range and the local earth radius are defined. The first takes arguments in one system and returns a value in the same system :

```
template<class System, typename T>
quantity<unit<boost::units::length_dimension, System>, T>
radar_beam_height(const quantity<unit<length_dimension, System>, T>& radar_range,
                  const quantity<unit<length_dimension, System>, T>& earth_radius,
                  T k = 4.0/3.0)
{
    return quantity<unit<length_dimension, System>, T>
        (pow<2>(radar_range)/(2.0*k*earth_radius));
}
```

The second is similar, but is templated on return type, so that the arguments are converted to the return unit system internally :

```
template<class return_type, class System1, class System2, typename T>
return_type
radar_beam_height(const quantity<unit<length_dimension, System1>, T>& radar_range,
                  const quantity<unit<length_dimension, System2>, T>& earth_radius,
                  T k = 4.0/3.0)
{
    // need to decide which system to use for calculation
    const return_type rr(radar_range),
                    er(earth_radius);

    return return_type(pow<2>(rr)/(2.0*k*er));
}
```

Finally, the third function is an empirical approximation that is only valid for radar ranges specified in nautical miles, returning beam height in feet. This function uses the heterogeneous unit of nautical miles per square root of feet to ensure dimensional correctness :

```
quantity<imperial::length>
radar_beam_height(const quantity<nautical::length>& range)
{
    return quantity<imperial::length>
        (pow<2>(range/(1.23*nautical::miles/root<2>(imperial::feet))));
}
```

With these, we can compute radar beam height in various unit systems :

```
const quantity<nautical::length> radar_range(300.0*miles);
const quantity<si::length> earth_radius(6371.0087714*kilo*meters);

const quantity<si::length> beam_height_1(radar_beam_height(quantity<si::length>(radar_range), earth_radius));
const quantity<nautical::length> beam_height_2(radar_beam_height(radar_range, quantity<nautical::length>(earth_radius)));
const quantity<si::length> beam_height_3(radar_beam_height< quantity<si::length> >(radar_range, earth_radius));
const quantity<nautical::length> beam_height_4(radar_beam_height< quantity<nautical::length> >(radar_range, earth_radius));
```

giving

```

radar range      : 300 nmi
earth radius     : 6.37101e+06 m
beam height 1    : 18169.7 m
beam height 2    : 9.81085 nmi
beam height 3    : 18169.7 m
beam height 4    : 9.81085 nmi
beam height approx : 59488.4 ft
beam height approx : 18132.1 m

```

## Heterogeneous Unit Example

(heterogeneous\_unit.cpp)

Mixed units and mixed unit conversions.

This code:

```

quantity<si::length>      L(1.5*si::meter);
quantity<cgs::mass>       M(1.0*cgs::gram);

std::cout << L << std::endl
          << M << std::endl
          << L*M << std::endl
          << L/M << std::endl
          << std::endl;

std::cout << 1.0*si::meter*si::kilogram/pow<2>(si::second) << std::endl
          << 1.0*si::meter*si::kilogram/pow<2>(si::second)/si::meter
          << std::endl << std::endl;

std::cout << 1.0*cgs::centimeter*si::kilogram/pow<2>(si::second) << std::endl
          << 1.0*cgs::centimeter*si::kilogram/pow<2>(si::second)/si::meter
          << std::endl << std::endl;

```

gives

```

1.5 m
1 g
1.5 m g
1.5 m g^-1

1 N
1 kg s^-2

1 cm kg s^-2
1 cm m^-1 kg s^-2

```

Arbitrary conversions also work:

```

quantity<si::area>      A(1.5*si::meter*cgs::centimeter);

std::cout << 1.5*si::meter*cgs::centimeter << std::endl
          << A << std::endl
          << std::endl;

```

yielding

```
1.5 cm m
0.015 m^2
```

## Absolute and Relative Temperature Example

([temperature.cpp](#))

This example demonstrates using of absolute temperatures and relative temperature differences in Fahrenheit and converting between these and the Kelvin temperature scale. This issue touches on some surprisingly deep mathematical concepts (see [Wikipedia](#) for a basic review), but for our purposes here, we will simply observe that it is important to be able to differentiate between an absolute temperature measurement and a measurement of temperature difference. This is accomplished by using the [absolute](#) wrapper class.

First we define a system using the predefined fahrenheit base unit:

```
typedef temperature::fahrenheit_base_unit::unit_type    temperature;
typedef get_system<temperature>::type                  system;

BOOST_UNITS_STATIC_CONSTANT(degree, temperature);
BOOST_UNITS_STATIC_CONSTANT(degrees, temperature);
```

Now we can create some quantities:

```
quantity<absolute<fahrenheit::temperature> >    T1p(
    32.0*absolute<fahrenheit::temperature>());
quantity<fahrenheit::temperature>                T1v(
    32.0*fahrenheit::degrees);

quantity<absolute<si::temperature> >             T2p(T1p);
quantity<si::temperature>                        T2v(T1v);
```

Note the use of [absolute](#) to wrap a unit. The resulting output is:

```
{ 32 } F
{ 273.15 } K
{ 273.15 } K
[ 32 ] F
[ 17.7778 ] K
[ 17.7778 ] K
```

## Runtime Conversion Factor Example

([runtime\\_conversion\\_factor.cpp](#))

The Boost.Units library does not require that the conversion factors be compile time constants, as is demonstrated in this example:

```
using boost::units::base_dimension;
using boost::units::base_unit;

static const long currency_base = 1;

struct currency_base_dimension : base_dimension<currency_base_dimension, 1> {};

typedef currency_base_dimension::dimension_type currency_type;

template<long N>
struct currency_base_unit :
    base_unit<currency_base_unit<N>, currency_type, currency_base + N> {};

typedef currency_base_unit<0> us_dollar_base_unit;
typedef currency_base_unit<1> euro_base_unit;

typedef us_dollar_base_unit::unit_type us_dollar;
typedef euro_base_unit::unit_type euro;

// an array of all possible conversions
double conversion_factors[2][2] = {
    {1.0, 1.0},
    {1.0, 1.0}
};

double get_conversion_factor(long from, long to) {
    return(conversion_factors[from][to]);
}

void set_conversion_factor(long from, long to, double value) {
    conversion_factors[from][to] = value;
    conversion_factors[to][from] = 1.0 / value;
}

BOOST_UNITS_DEFINE_CONVERSION_FACTOR_TEMPLATE((long N1)(long N2),
    currency_base_unit<N1>,
    currency_base_unit<N2>,
    double, get_conversion_factor(N1, N2));
```

## Units with Non-base Dimensions

([non\\_base\\_dimension.cpp](#))

It is also possible to define base units that have derived rather than base dimensions:



```

struct imperial_gallon_tag :
    base_unit<imperial_gallon_tag, volume_dimension, 1> { };

typedef make_system<imperial_gallon_tag>::type imperial;

typedef unit<volume_dimension,imperial> imperial_gallon;

struct us_gallon_tag : base_unit<us_gallon_tag, volume_dimension, 2> { };

typedef make_system<us_gallon_tag>::type us;

typedef unit<volume_dimension,us> us_gallon;

```

## Output for Composite Units

([composite\\_output.cpp](#))

If a unit has a special name and/or symbol, the free functions `name_string` and `symbol_string` can be overloaded directly.

```

std::string name_string(const cgs::force&)
{
    return "dyne";
}

std::string symbol_string(const cgs::force&)
{
    return "dyn";
}

```

In this case, any unit that reduces to the overloaded unit will be output with the replacement symbol.

Special names and symbols for the SI and CGS unit systems are found in [boost/units/systems/si/io.hpp](#) and [boost/units/systems/cgs/io.hpp](#), respectively. If these headers are not included, the output will simply follow default rules using the appropriate fundamental dimensions. Note that neither of these functions is defined for quantities because doing so would require making assumptions on how the corresponding value type should be formatted.

Three ostream formatters, `symbol_format`, `name_format`, and `typename_format` are provided for convenience. These select the textual representation of units provided by `symbol_string` or `name_string` in the first two cases, while the latter returns a demangled typename for debugging purposes. Formatting of scaled unit is also done correctly.

## Automatically Scaled Units

It is often desirable to scale a [unit](#) automatically, depending on its value, to keep the integral part in a limited range, usually between 1 and 999.

For example, using [engineering notation prefixes](#),

```

"1234.5 m" is more helpfully displayed as "1.234 km"
"0.000000001234 m" is more clearly displayed as "1.2345 nanometer".

```

The iostream manipulators `engineering_prefixes` or `binary_prefixes` make this easy.

```
using boost::units::binary_prefix;
using boost::units::engineering_prefix;
using boost::units::no_prefix;

quantity<length> l = 2.345 * meters; // A quantity of length, in units of meters.
cout << engineering_prefix << l << endl; // Outputs "2.345 m".
l = 1000.0 * l; // Increase it by 1000, so expect a k prefix.
// Note that a double 1000.0 is required - an integer will fail to compile.
cout << engineering_prefix << l << endl; // Output autoprefixed with k to "2.345 km".

quantity<energy> e = kilograms * pow<2>(1 / seconds); // A quantity of energy.
cout << engineering_prefix << e << endl; // 5.49902 MJ
cout << name_format << engineering_prefix << e << endl; // 5.49902 megaJoule
```

(The complete set of [engineering and scientific multiples](#) is not used (not centi or deci for example), but only powers of ten that are multiples of three,  $10^3$ ).

Similarly, the equivalent [binary prefixes](#) used for displaying computing kilobytes, megabytes, gigabytes...

These are the  $2^{10} = 1024$ ,  $2^{20} = 1\,048\,576$ ,  $2^{30}$  ... multiples.

(See also [Prefixes for binary multiples](#))

This scale is specified in IEC 60027-2, Second edition, 2000-11, Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics).

```
// Don't forget that the units name or symbol format specification is persistent.
cout << symbol_format << endl; // Resets the format to the default symbol format.

quantity<byte_base_unit::unit_type> b = 2048. * byte_base_unit::unit_type();
cout << engineering_prefix << b << endl; // 2.048 kb
cout << symbol_format << binary_prefix << b << endl; // "2 Kib"
```

But note that scalar dimensionless values, like int, float and double, are **not** prefixed automatically by the engineering\_prefix or binary\_prefix iostream manipulators.

```
const double s1 = 2345.6;
const long x1 = 23456;
cout << engineering_prefix << s1 << endl; // 2345.6
cout << engineering_prefix << x1 << endl; // 23456

cout << binary_prefix << s1 << endl; // 2345.6
cout << binary_prefix << x1 << endl; // 23456
```

You can output the name or symbol of a unit (rather than the most common quantity of a unit).

```
const length L; // A unit of length (but not a quantity of length).
cout << L << endl; // Default length unit is meter,
// but default is symbol format so output is just "m".
cout << name_format << L << endl; // default length name is "meter".
```

Note too that all the formatting flags are persistent, so that if you set engineering\_prefix, then it applies to all future outputs, until you select binary\_prefix, or explicitly switch autoprefix off. You can specify no prefix (the default of course) in two ways:

```
no_prefix(cout); // Clear any prefix flag.
cout << no_prefix << endl; // Clear any prefix flag using `no_prefix` manipulator.
```

And you can get the format flags for diagnosing problems.

```
cout << boost::units::get_autoprefix(cout) << endl; // 8 is `autoprefix_binary` from `enum auto-  
prefix_mode`.  
cout << boost::units::get_format(cout) << endl; // 1 is `name_fmt` from `enum format_mode`.
```

## Conversion Factor

This code demonstrates the use of the `conversion_factor` free function to determine the scale factor between two units.

([conversion\\_factor.cpp](#))

```
double dyne_to_newton =  
    conversion_factor(cgs::dyne, si::newton);  
std::cout << dyne_to_newton << std::endl;  
  
double force_over_mass_conversion =  
    conversion_factor(si::newton/si::kilogram, cgs::dyne/cgs::gram);  
std::cout << force_over_mass_conversion << std::endl;  
  
double momentum_conversion =  
    conversion_factor(cgs::momentum(), si::momentum());  
std::cout << momentum_conversion << std::endl;  
  
double momentum_over_mass_conversion =  
    conversion_factor(si::momentum()/si::mass(), cgs::momentum()/cgs::gram);  
std::cout << momentum_over_mass_conversion << std::endl;  
  
double acceleration_conversion =  
    conversion_factor(cgs::gal, si::meter_per_second_squared);  
std::cout << acceleration_conversion << std::endl;
```

Produces

```
1e-005  
100  
1e-005  
100  
0.01
```

## Runtime Units

([runtime\\_unit.cpp](#))

This example shows how to implement an interface that allow different units at runtime while still maintaining type safety for internal calculations.

```

namespace {

using namespace boost::units;
using imperial::foot_base_unit;

std::map<std::string, quantity<si::length> > known_units;

}

quantity<si::length> calculate(const quantity<si::length>& t)
{
    return(boost::units::hypot(t, 2.0 * si::meters));
}

int main()
{
    known_units["meter"] = 1.0 * si::meters;
    known_units["centimeter"] = .01 * si::meters;
    known_units["foot"] =
        conversion_factor(foot_base_unit::unit_type(), si::meter) * si::meter;

    std::string output_type("meter");
    std::string input;

    while((std::cout << "> " ) && (std::cin >> input))
    {
        if(!input.empty() && input[0] == '#')
        {
            std::getline(std::cin, input);
        }
        else if(input == "exit")
        {
            break;
        }
        else if(input == "help")
        {
            std::cout << "type \"exit\" to exit\n"
                "type \"return 'unit'\" to set the return units\n"
                "type \"'number' 'unit'\" to do a simple calculation"
                << std::endl;
        }
        else if(input == "return")
        {
            if(std::cin >> input)
            {
                if(known_units.find(input) != known_units.end())
                {
                    output_type = input;
                    std::cout << "Done." << std::endl;
                }
                else
                {
                    std::cout << "Unknown unit \"" << input << "\"\n"
                        << std::endl;
                }
            }
            else
            {
                break;
            }
        }
        else
        {

```

```
try
{
    double value = boost::lexical_cast<double>(input);

    if(std::cin >> input)
    {
        if(known_units.find(input) != known_units.end())
        {
            std::cout << static_cast<double>(
                calculate(value * known_units[input]) /
                known_units[output_type])
                << ' ' << output_type << std::endl;
        }
        else
        {
            std::cout << "Unknown unit \"" << input << "\" "
                << std::endl;
        }
    }
    else
    {
        break;
    }
}
catch(...)
{
    std::cout << "Input error" << std::endl;
}
}
```

## Interoperability with Boost.Lambda

(lambda.cpp)

The header `boost/units/lambda.hpp` provides overloads and specializations needed to make Boost.Units usable with the Boost.Lambda library.

```

int main(int argc, char **argv) {

    using namespace std;
    namespace bl = boost::lambda;
    namespace bu = boost::units;
    namespace si = boost::units::si;

    ////////////////////////////////////////////////////
    // Mechanical example: linear accelerated movement
    ////////////////////////////////////////////////////

    // Initial condition variables for acceleration, speed, and displacement
    bu::quantity<si::acceleration> a = 2.0 * si::meters_per_second_squared;
    bu::quantity<si::velocity> v = 1.0 * si::meters_per_second;
    bu::quantity<si::length> s0 = 0.5 * si::meter;

    // Displacement over time
    boost::function<bu::quantity<si::length> (bu::quantity<si::time>) >
        s = 0.5 * bl::var(a) * bl::_1 * bl::_1
            + bl::var(v) * bl::_1
            + bl::var(s0);

    cout << "Linear accelerated movement:" << endl
         << "a = " << a << ", v = " << v << ", s0 = " << s0 << endl
         << "s(1.0 * si::second) = " << s(1.0 * si::second) << endl
         << endl;

    // Change initial conditions
    a = 1.0 * si::meters_per_second_squared;
    v = 2.0 * si::meters_per_second;
    s0 = -1.5 * si::meter;

    cout << "a = " << a << ", v = " << v << ", s0 = " << s0 << endl
         << "s(1.0 * si::second) = " << s(1.0 * si::second) << endl
         << endl;

    ////////////////////////////////////////////////////
    // Electrical example: oscillating current
    ////////////////////////////////////////////////////

    // Constants for the current amplitude, frequency, and offset current
    const bu::quantity<si::current> iamp = 1.5 * si::ampere;
    const bu::quantity<si::frequency> f = 1.0e3 * si::hertz;
    const bu::quantity<si::current> i0 = 0.5 * si::ampere;

    // The invocation of the sin function needs to be postponed using
    // bind to specify the oscillation function. A lengthy static_cast
    // to the function pointer referencing boost::units::sin() is needed
    // to avoid an "unresolved overloaded function type" error.
    boost::function<bu::quantity<si::current> (bu::quantity<si::time>) >
        i = iamp
            * bl::bind(static_cast<bu::dimensionless_quantity<si::sys::J
tem, double>::type (*)(&const bu::quantity<si::plane_angle>&)>(bu::sin),
                        2.0 * pi * si::radian * f * bl::_1)
            + i0;

    cout << "Oscillating current:" << endl
         << "iamp = " << iamp << ", f = " << f << ", i0 = " << i0 << endl
         << "i(1.25e-3 * si::second) = " << i(1.25e-3 * si::second) << endl
         << endl;
}

```

```

////////////////////////////////////
// Geometric example: area calculation for a square
////////////////////////////////////

// Length constant
const bu::quantity<si::length> l = 1.5 * si::meter;

// Again an ugly static_cast is needed to bind pow<2> to the first
// function argument.
boost::function<bu::quantity<si::area> (bu::quantity<si::length>) >
    A = bl::bind(static_cast<bu::quantity<si::area> (*)>(const bu::quant
ity<si::length>&)>(bu::pow<2>),
                bl::_1);

cout << "Area of a square:" << endl
     << "A(" << l <<") = " << A(l) << endl << endl;

////////////////////////////////////
// Thermal example: temperature difference of two absolute temperatures
////////////////////////////////////

// Absolute temperature constants
const bu::quantity<bu::absolute<si::temperature> >
    Tref = 273.15 * bu::absolute<si::temperature>();
const bu::quantity<bu::absolute<si::temperature> >
    Tamb = 300.00 * bu::absolute<si::temperature>();

boost::function<bu::quantity<si::temperature> (bu::quantity<bu::absolute<si::temperature> >,
                                                bu::quantity<bu::absolute<si::temperature> >)>
    dT = bl::_2 - bl::_1;

cout << "Temperature difference of two absolute temperatures:" << endl
     << "dT(" << Tref << ", " << Tamb << ") = " << dT(Tref, Tamb) << endl
     << endl;

return 0;
}

```

## Utilities

Relatively complete SI and CGS unit systems are provided in [boost/units/systems/si.hpp](http://boost.org/libs/units/doc/html/boost_units_systems_si.html) and [boost/units/systems/cgs.hpp](http://boost.org/libs/units/doc/html/boost_units_systems_cgs.html), respectively.

## Metaprogramming Classes

```
template<long N> struct ordinal<N>;

template<typename T,typename V> struct get_tag< dim<T,V> >;
template<typename T,typename V> struct get_value< dim<T,V> >;
template<class S,class DT> struct get_system_tag_of_dim<S,DT>;
template<typename Seq> struct make_dimension_list<Seq>;
template<class DT> struct fundamental_dimension<DT>;
template<class DT1,int E1,...> struct composite_dimension<DT1,E1,...>;

template<class Dim,class System> struct get_dimension< unit<Dim,System> >;
template<class Unit,class Y> struct get_dimension< quantity<Unit,Y> >;
template<class Dim,class System> struct get_system< unit<Dim,System> >;
template<class Unit,class Y> struct get_system< quantity<Unit,Y> >;

struct dimensionless_type;
template<class System> struct dimensionless_unit<System>;
template<class System,class Y> struct dimensionless_quantity<System,Y>;

struct implicitly_convertible;
struct trivial_conversion;
template<class T,class S1,class S2> struct base_unit_converter<T,S1,S2>;

template<class Q1,class Q2> class conversion_helper<Q1,Q2>;
```

## Metaprogramming Predicates

```
template<typename T,typename V> struct is_dim< dim<T,V> >;
template<typename T,typename V> struct is_empty_dim< dim<T,V> >;

template<typename Seq> struct is_dimension_list<Seq>;

template<class S> struct is_system< homogeneous_system<S> >;
template<class S> struct is_system< heterogeneous_system<S> >;
template<class S> struct is_homogeneous_system< homogeneous_system<S> >;
template<class S> struct is_heterogeneous_system< heterogeneous_system<S> >;

template<class Dim,class System> struct is_unit< unit<Dim,System> >;
template<class Dim,class System> struct is_unit_of_system< unit<Dim,System>,System >;
template<class Dim,class System> struct is_unit_of_dimension< unit<Dim,System>,Dim >;

template<class Unit,class Y> struct is_quantity< quantity<Unit,Y> >;
template<class Dim,class System,class Y> struct is_quantity_of_system< quantity<unit<Dim,system>,Y>,System >;
template<class Dim,class System,class Y> struct is_quantity_of_dimension< quantity<unit<Dim,system>,Y>,Dim >;

template<class System> struct is_dimensionless< unit<dimensionless_type,System> >;
template<class System> struct is_dimensionless_unit< unit<dimensionless_type,System> >;
template<class System,class Y> struct is_dimensionless< quantity<unit<dimensionless_type,system>,Y> >;
template<class System,class Y> struct is_dimensionless_quantity< quantity<unit<dimensionless_type,system>,Y>,System>,Y> >;
```

## Reference

### Units Reference



## Header `<boost/units/absolute.hpp>`

Absolute units (points rather than vectors).

Operations between absolute units, and relative units like temperature differences.

```
BOOST_UNITS_DEFINE_CONVERSION_OFFSET(From, To, type_, value_)
```

```
namespace boost {
  namespace units {
    template<typename Y> class absolute;

    // add a relative value to an absolute one
    template<typename Y>
      absolute< Y > operator+(const absolute< Y > & aval, const Y & rval);

    // add a relative value to an absolute one
    template<typename Y>
      absolute< Y > operator+(const Y & rval, const absolute< Y > & aval);

    // subtract a relative value from an absolute one
    template<typename Y>
      absolute< Y > operator-(const absolute< Y > & aval, const Y & rval);

    // subtracting two absolutes gives a difference
    template<typename Y>
      Y operator-(const absolute< Y > & aval1, const absolute< Y > & aval2);

    // creates a quantity from an absolute unit and a raw value
    template<typename D, typename S, typename T>
      quantity< absolute< unit< D, S > >, T >
      operator*(const T & t, const absolute< unit< D, S > > &);

    // creates a quantity from an absolute unit and a raw value
    template<typename D, typename S, typename T>
      quantity< absolute< unit< D, S > >, T >
      operator*(const absolute< unit< D, S > > &, const T & t);

    // Print an absolute unit.
    template<typename Char, typename Traits, typename Y>
      std::basic_ostream< Char, Traits > &
      operator<<(std::basic_ostream< Char, Traits > & os,
        const absolute< Y > & aval);
  }
}
```

## Class template absolute

boost::units::absolute

## Synopsis

```
// In header: <boost/units/absolute.hpp>

template<typename Y>
class absolute {
public:
    // types
    typedef absolute< Y > this_type;
    typedef Y          value_type;

    // construct/copy/destroy
    absolute();
    absolute(const value_type &);
    absolute(const this_type &);
    absolute& operator=(const this_type &);

    // public member functions
    const value_type & value() const;
    const this_type & operator+=(const value_type &);
    const this_type & operator-=(const value_type &);
};
```

### Description

A wrapper to represent absolute units (points rather than vectors). Intended originally for temperatures, this class implements operators for absolute units so that addition of a relative unit to an absolute unit results in another absolute unit : `absolute<T> +/- T -> absolute<T>` and subtraction of one absolute unit from another results in a relative unit : `absolute<T> - absolute<T> -> T`.

#### absolute public construct/copy/destroy

1. `absolute();`
2. `absolute(const value_type & val);`
3. `absolute(const this_type & source);`
4. `absolute& operator=(const this_type & source);`

#### absolute public member functions

1. `const value_type & value() const;`
2. `const this_type & operator+=(const value_type & val);`

3. `const this_type & operator--=(const value_type & val);`

## Macro BOOST\_UNITS\_DEFINE\_CONVERSION\_OFFSET

BOOST\_UNITS\_DEFINE\_CONVERSION\_OFFSET

## Synopsis

```
// In header: <boost/units/absolute.hpp>

BOOST_UNITS_DEFINE_CONVERSION_OFFSET(From, To, type_, value_)
```

### Description

Macro to define the offset between two absolute units. Requires the value to be in the destination units e.g

```
BOOST_UNITS_DEFINE_CONVERSION_OFFSET(celsius_base_unit, fahrenheit_base_unit, double, 32.0);
```

BOOST\_UNITS\_DEFINE\_CONVERSION\_FACTOR is also necessary to specify the conversion factor. Like BOOST\_UNITS\_DEFINE\_CONVERSION\_FACTOR this macro defines both forward and reverse conversions so defining, e.g., the conversion from celsius to fahrenheit as above will also define the inverse conversion from fahrenheit to celsius.

## Header <boost/units/base\_dimension.hpp>

base dimensions (mass, length, time...).

base dimension definition registration.

```
namespace boost {
    namespace units {
        template<typename Derived, long N> class base_dimension;
    }
}
```

## Class template `base_dimension`

`boost::units::base_dimension`

## Synopsis

```
// In header: <boost/units/base_dimension.hpp>

template<typename Derived, long N>
class base_dimension {
public:
    // types
    typedef unspecified dimension_type; // A convenience typedef. Equivalent to boost::units::derived_dimension<Derived,1>::type.
    typedef Derived type;               // Provided for mpl compatability.
};
```

## Description

Defines a base dimension. To define a dimension you need to provide the derived class (CRTP) and a unique integer.

```
struct my_dimension : boost::units::base_dimension<my_dimension, 1> {};
```

It is designed so that you will get an error message if you try to use the same value in multiple definitions.

## Header `<boost/units/base_unit.hpp>`

base unit (meter, kg, sec...).

base unit definition registration.

```
namespace boost {
    namespace units {
        template<typename Derived, typename Dim, long N> class base_unit;
    }
}
```

## Class template `base_unit`

`boost::units::base_unit`

## Synopsis

```
// In header: <boost/units/base_unit.hpp>

template<typename Derived, typename Dim, long N>
class base_unit {
public:
    // types
    typedef Dim          dimension_type; // The dimensions of this base unit.
    typedef Derived      type;          // Provided for mpl compatability.
    typedef unspecified unit_type;      // The unit corresponding to this base unit.
};
```

## Description

Defines a base unit. To define a unit you need to provide the derived class (CRTP), a dimension list and a unique integer.

```
struct my_unit : boost::units::base_unit<my_unit, length_dimension, 1> {};
```

It is designed so that you will get an error message if you try to use the same value in multiple definitions.

## Header `<boost/units/cmath.hpp>`

Overloads of functions in `<cmath>` for quantities.

Only functions for which a dimensionally-correct result type can be determined are overloaded. All functions work with dimensionless quantities.

```

namespace boost {
namespace units {
template<typename Unit, typename Y>
    bool isinfinite(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    bool isinf(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    bool isnan(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    bool isnormal(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    bool isgreater(const quantity< Unit, Y > & q1,
                   const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    bool isgreaterequal(const quantity< Unit, Y > & q1,
                        const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    bool isless(const quantity< Unit, Y > & q1,
                const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    bool islessequal(const quantity< Unit, Y > & q1,
                     const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    bool islessgreater(const quantity< Unit, Y > & q1,
                       const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    bool isunordered(const quantity< Unit, Y > & q1,
                     const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    quantity< Unit, Y > abs(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    quantity< Unit, Y > ceil(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    quantity< Unit, Y >
        copysign(const quantity< Unit, Y > & q1, const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    quantity< Unit, Y > fabs(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    quantity< Unit, Y > floor(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    quantity< Unit, Y >
        fdim(const quantity< Unit, Y > & q1, const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    quantity< Unit, Y >
        fmax(const quantity< Unit, Y > & q1, const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    quantity< Unit, Y >
        fmin(const quantity< Unit, Y > & q1, const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    int fpclassify(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    root_typeof_helper< typename add_typeof_helper< typename power_typeof_helper< quant.L
ity< Unit, Y >, static_rational< 2 > >::type, typename power_typeof_helper< quant.L
ity< Unit, Y >, static_rational< 2 > >::type >::type, static_rational< 2 > >::type
        hypot(const quantity< Unit, Y > & q1, const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    quantity< Unit, Y >
        nextafter(const quantity< Unit, Y > & q1,
                  const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    quantity< Unit, Y >
        nexttoward(const quantity< Unit, Y > & q1,
                   const quantity< Unit, Y > & q2);

```

```

template<typename Unit, typename Y>
    quantity< Unit, Y > round(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    int signbit(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    quantity< Unit, Y > trunc(const quantity< Unit, Y > & q);
template<typename Unit, typename Y>
    quantity< Unit, Y >
        fmod(const quantity< Unit, Y > & q1, const quantity< Unit, Y > & q2);
template<typename Unit, typename Y>
    quantity< Unit, Y >
        modf(const quantity< Unit, Y > & q1, quantity< Unit, Y > * q2);
template<typename Unit, typename Y, typename Int>
    quantity< Unit, Y > frexp(const quantity< Unit, Y > & q, Int * ex);
template<typename S, typename Y>
    quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y >
        pow(const quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y > &,
            const quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y > &);
template<typename S, typename Y>
    quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y >
        exp(const quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y > & q);
template<typename Unit, typename Y, typename Int>
    quantity< Unit, Y > ldexp(const quantity< Unit, Y > & q, const Int & ex);
template<typename S, typename Y>
    quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y >
        log(const quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y > & q);
template<typename S, typename Y>
    quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y >
        log10(const quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y > & q);
template<typename Unit, typename Y>
    root_typeof_helper< quantity< Unit, Y >, static_rational< 2 > >::type
        sqrt(const quantity< Unit, Y > & q);

// cos of theta in radians
template<typename Y>
    dimensionless_quantity< si::system, Y >::type
        cos(const quantity< si::plane_angle, Y > & theta);

// sin of theta in radians
template<typename Y>
    dimensionless_quantity< si::system, Y >::type
        sin(const quantity< si::plane_angle, Y > & theta);

// tan of theta in radians
template<typename Y>
    dimensionless_quantity< si::system, Y >::type
        tan(const quantity< si::plane_angle, Y > & theta);

// cos of theta in other angular units
template<typename System, typename Y>
    dimensionless_quantity< System, Y >::type
        cos(const quantity< unit< plane_angle_dimension, System >, Y > & theta);

// sin of theta in other angular units
template<typename System, typename Y>
    dimensionless_quantity< System, Y >::type
        sin(const quantity< unit< plane_angle_dimension, System >, Y > & theta);

// tan of theta in other angular units
template<typename System, typename Y>
    dimensionless_quantity< System, Y >::type
        tan(const quantity< unit< plane_angle_dimension, System >, Y > & theta);

```



```

// acos of dimensionless quantity returning angle in same system
template<typename Y, typename System>
    quantity< unit< plane_angle_dimension, homogeneous_system< System > >, Y >
    acos(const quantity< unit< dimensionless_type, homogeneous_system< System > >, Y > & val);

// acos of dimensionless quantity returning angle in radians
template<typename Y>
    quantity< angle::radian_base_unit::unit_type, Y >
    acos(const quantity< unit< dimensionless_type, heterogeneous_dimensionless_sys-
tem >, Y > & val);

// asin of dimensionless quantity returning angle in same system
template<typename Y, typename System>
    quantity< unit< plane_angle_dimension, homogeneous_system< System > >, Y >
    asin(const quantity< unit< dimensionless_type, homogeneous_system< System > >, Y > & val);

// asin of dimensionless quantity returning angle in radians
template<typename Y>
    quantity< angle::radian_base_unit::unit_type, Y >
    asin(const quantity< unit< dimensionless_type, heterogeneous_dimensionless_sys-
tem >, Y > & val);

// atan of dimensionless quantity returning angle in same system
template<typename Y, typename System>
    quantity< unit< plane_angle_dimension, homogeneous_system< System > >, Y >
    atan(const quantity< unit< dimensionless_type, homogeneous_system< System > >, Y > & val);

// atan of dimensionless quantity returning angle in radians
template<typename Y>
    quantity< angle::radian_base_unit::unit_type, Y >
    atan(const quantity< unit< dimensionless_type, heterogeneous_dimensionless_sys-
tem >, Y > & val);

// atan2 of value_type returning angle in radians
template<typename Y, typename Dimension, typename System>
    quantity< unit< plane_angle_dimension, homogeneous_system< System > >, Y >
    atan2(const quantity< unit< Dimension, homogeneous_system< System > >, Y > & y,
          const quantity< unit< Dimension, homogeneous_system< System > >, Y > & x);

// atan2 of value_type returning angle in radians
template<typename Y, typename Dimension, typename System>
    quantity< angle::radian_base_unit::unit_type, Y >
    atan2(const quantity< unit< Dimension, heterogeneous_system< System > >, Y > & y,
          const quantity< unit< Dimension, heterogeneous_system< System > >, Y > & x);
}

```

## Function template pow

boost::units::pow

## Synopsis

```
// In header: <boost/units/cmath.hpp>

template<typename S, typename Y>
    quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y >
    pow(const quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y > & q1,
        const quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(S), Y > & q2);
```

## Description

For non-dimensionless quantities, integral and rational powers and roots can be computed by `pow<Ex>` and `root<Rt>` respectively.

## Header <boost/units/config.hpp>

```
BOOST_UNITS_REQUIRE_LAYOUT_COMPATIBILITY
BOOST_UNITS_NO_COMPILER_CHECK
BOOST_UNITS_CHECK_HOMOGENEOUS_UNITS
```

**Macro BOOST\_UNITS\_REQUIRE\_LAYOUT\_COMPATIBILITY**

BOOST\_UNITS\_REQUIRE\_LAYOUT\_COMPATIBILITY

**Synopsis**

```
// In header: <boost/units/config.hpp>

BOOST_UNITS_REQUIRE_LAYOUT_COMPATIBILITY
```

**Description**

If defined will trigger a static assertion if `quantity<Unit, T>` is not layout compatible with `T`

## Macro BOOST\_UNITS\_NO\_COMPILER\_CHECK

BOOST\_UNITS\_NO\_COMPILER\_CHECK

## Synopsis

```
// In header: <boost/units/config.hpp>

BOOST_UNITS_NO_COMPILER_CHECK
```

## Description

If defined will disable a preprocessor check that the compiler is able to handle the library.

## Macro `BOOST_UNITS_CHECK_HOMOGENEOUS_UNITS`

`BOOST_UNITS_CHECK_HOMOGENEOUS_UNITS`

## Synopsis

```
// In header: <boost/units/config.hpp>

BOOST_UNITS_CHECK_HOMOGENEOUS_UNITS
```

### Description

Enable checking to verify that a homogeneous system is actually capable of representing all the dimensions that it is used with. Off by default.

## Header `<boost/units/conversion.hpp>`

Template for defining conversions between quantities.

```
BOOST_UNITS_DEFINE_CONVERSION_FACTOR(Source, Destination, type_, value_)
BOOST_UNITS_DEFINE_CONVERSION_FACTOR_TEMPLATE(Params, Source, Destination, type_, value_)
BOOST_UNITS_DEFAULT_CONVERSION(Source, Dest)
BOOST_UNITS_DEFAULT_CONVERSION_TEMPLATE(Params, Source, Dest)
```

```
namespace boost {
  namespace units {
    template<typename From, typename To> struct conversion_helper;

    // Find the conversion factor between two units.
    template<typename FromUnit, typename ToUnit>
      unspecified conversion_factor(const FromUnit &, const ToUnit &);
  }
}
```

## Struct template conversion\_helper

boost::units::conversion\_helper

## Synopsis

```
// In header: <boost/units/conversion.hpp>

template<typename From, typename To>
struct conversion_helper {

    // public static functions
    static To convert(const From &);
};
```

## Description

Template for defining conversions between quantities. This template should be specialized for every quantity that allows conversions. For example, if you have a two units called pair and dozen you would write

```
namespace boost {
namespace units {
template<class T0, class T1>
struct conversion_helper<quantity<dozen, T0>, quantity<pair, T1> >
{
    static quantity<pair, T1> convert(const quantity<dozen, T0>& source)
    {
        return(quantity<pair, T1>::from_value(6 * source.value()));
    }
};
}
}
```

In most cases, the predefined specializations for unit and absolute should be sufficient, so users should rarely need to use this.

### conversion\_helper public static functions

1. 

```
static To convert(const From &);
```

## Macro `BOOST_UNITS_DEFINE_CONVERSION_FACTOR`

`BOOST_UNITS_DEFINE_CONVERSION_FACTOR`

## Synopsis

```
// In header: <boost/units/conversion.hpp>

BOOST_UNITS_DEFINE_CONVERSION_FACTOR(Source, Destination, type_, value_)
```

## Description

Defines the conversion factor from a base unit to any unit or to another base unit with the correct dimensions. Uses of this macro must appear at global scope. If the destination unit is a base unit or a unit that contains only one base unit which is raised to the first power (e.g. feet->meters) the reverse (meters->feet in this example) need not be defined explicitly.

## Macro `BOOST_UNITS_DEFINE_CONVERSION_FACTOR_TEMPLATE`

`BOOST_UNITS_DEFINE_CONVERSION_FACTOR_TEMPLATE`

## Synopsis

```
// In header: <boost/units/conversion.hpp>

BOOST_UNITS_DEFINE_CONVERSION_FACTOR_TEMPLATE(Params, Source, Destination, type_, value_)
```

## Description

Defines the conversion factor from a base unit to any other base unit with the same dimensions. Params should be a Boost.Preprocessor Seq of template parameters, such as (class T1)(class T2) All uses of must appear at global scope. The reverse conversion will be defined automatically. This macro is a little dangerous, because, unlike the non-template form, it will silently fail if either base unit is scaled. This is probably not an issue if both the source and destination types depend on the template parameters, but be aware that a generic conversion to kilograms is not going to work.



## Macro BOOST\_UNITS\_DEFAULT\_CONVERSION

BOOST\_UNITS\_DEFAULT\_CONVERSION

## Synopsis

```
// In header: <boost/units/conversion.hpp>

BOOST_UNITS_DEFAULT_CONVERSION(Source, Dest)
```

## Description

Specifies the default conversion to be applied when no direct conversion is available. Source is a base unit. Dest is any unit with the same dimensions.

## Macro BOOST\_UNITS\_DEFAULT\_CONVERSION\_TEMPLATE

BOOST\_UNITS\_DEFAULT\_CONVERSION\_TEMPLATE

## Synopsis

```
// In header: <boost/units/conversion.hpp>

BOOST_UNITS_DEFAULT_CONVERSION_TEMPLATE(Params, Source, Dest)
```

### Description

Specifies the default conversion to be applied when no direct conversion is available. Params is a PP Sequence of template arguments. Source is a base unit. Dest is any unit with the same dimensions. The source must not be a scaled base unit.

### Header <boost/units/derived\_dimension.hpp>

```
namespace boost {
  namespace units {
    template<typename DT1 = dimensionless_type, long E1 = 0,
            typename DT2 = dimensionless_type, long E2 = 0,
            typename DT3 = dimensionless_type, long E3 = 0,
            typename DT4 = dimensionless_type, long E4 = 0,
            typename DT5 = dimensionless_type, long E5 = 0,
            typename DT6 = dimensionless_type, long E6 = 0,
            typename DT7 = dimensionless_type, long E7 = 0,
            typename DT8 = dimensionless_type, long E8 = 0>
    struct derived_dimension;
  }
}
```

## Struct template `derived_dimension`

`boost::units::derived_dimension` — A utility class for defining composite dimensions with integer powers.

## Synopsis

```
// In header: <boost/units/derived_dimension.hpp>

template<typename DT1 = dimensionless_type, long E1 = 0,
        typename DT2 = dimensionless_type, long E2 = 0,
        typename DT3 = dimensionless_type, long E3 = 0,
        typename DT4 = dimensionless_type, long E4 = 0,
        typename DT5 = dimensionless_type, long E5 = 0,
        typename DT6 = dimensionless_type, long E6 = 0,
        typename DT7 = dimensionless_type, long E7 = 0,
        typename DT8 = dimensionless_type, long E8 = 0>
struct derived_dimension {
    // types
    typedef unspecified type;
};
```

## Header `<boost/units/dim.hpp>`

Handling of fundamental dimension/exponent pairs.

```
namespace boost {
    namespace units {
        template<typename T, typename V> struct dim;
    }
}
```

## Struct template dim

boost::units::dim — Dimension tag/exponent pair for a single fundamental dimension.

## Synopsis

```
// In header: <boost/units/dim.hpp>

template<typename T, typename V>
struct dim {
    // types
    typedef dim      type;
    typedef unspecified tag;
    typedef T        tag_type;
    typedef V        value_type;
};
```

### Description

The dim class represents a single dimension tag/dimension exponent pair. That is, `dim<tag_type, value_type>` is a pair where `tag_type` represents the fundamental dimension being represented and `value_type` represents the exponent of that fundamental dimension as a `static_rational`. `tag_type` must be derived from a specialization of `base_dimension`. Specialization of the following Boost.MPL metafunctions are provided

- `mpl::plus` for two dims
- `mpl::minus` for two dims
- `mpl::negate` for a dim

These metafunctions all operate on the exponent, and require that the dim operands have the same base dimension tag. In addition, multiplication and division by `static_rational` is supported.

- `mpl::times` for a `static_rational` and a dim in either order
- `mpl::divides` for a `static_rational` and a dim in either order

These metafunctions likewise operate on the exponent only.

## Header **<boost/units/dimension.hpp>**

Core metaprogramming utilities for compile-time dimensional analysis.

```
namespace boost {
    namespace units {
        template<typename Seq> struct make_dimension_list;
        template<typename DL, typename Ex> struct static_power;
        template<typename DL, typename Rt> struct static_root;
    }
}
```

## Struct template `make_dimension_list`

`boost::units::make_dimension_list`

## Synopsis

```
// In header: <boost/units/dimension.hpp>

template<typename Seq>
struct make_dimension_list {
    // types
    typedef unspecified type;
};
```

## Description

Reduce dimension list to cardinal form. This algorithm collapses duplicate base dimension tags and sorts the resulting list by the tag ordinal value. Dimension lists that resolve to the same dimension are guaranteed to be represented by an identical type.

The argument should be an MPL forward sequence containing instances of the `dim` template.

The result is also an MPL forward sequence. It also supports the following metafunctions to allow use as a dimension.

- `mpl::plus` is defined only on two equal dimensions and returns the argument unchanged.
- `mpl::minus` is defined only for two equal dimensions and returns the argument unchanged.
- `mpl::negate` will return its argument unchanged.
- `mpl::times` is defined for any dimensions and adds corresponding exponents.
- `mpl::divides` is defined for any dimensions and subtracts the exponents of the right hand argument from the corresponding exponents of the left hand argument. Missing base dimension tags are assumed to have an exponent of zero.
- `static_power` takes a dimension and a `static_rational` and multiplies all the exponents of the dimension by the `static_rational`.
- `static_root` takes a dimension and a `static_rational` and divides all the exponents of the dimension by the `static_rational`.

## Struct template `static_power`

`boost::units::static_power` — Raise a dimension list to a scalar power.

## Synopsis

```
// In header: <boost/units/dimension.hpp>

template<typename DL, typename Ex>
struct static_power {
    // types
    typedef unspecified type;
};
```

## Struct template `static_root`

`boost::units::static_root` — Take a scalar root of a dimension list.

## Synopsis

```
// In header: <boost/units/dimension.hpp>

template<typename DL, typename Rt>
struct static_root {
    // types
    typedef unspecified type;
};
```

## Header <[boost/units/dimensionless\\_quantity.hpp](#)>

Utility class to simplify construction of dimensionless quantities.

```
namespace boost {
    namespace units {
        template<typename System, typename Y> struct dimensionless_quantity;
    }
}
```

## Struct template `dimensionless_quantity`

`boost::units::dimensionless_quantity` — Utility class to simplify construction of dimensionless quantities.

## Synopsis

```
// In header: <boost/units/dimensionless_quantity.hpp>

template<typename System, typename Y>
struct dimensionless_quantity {
    // types
    typedef quantity< typename dimensionless_unit< System >::type, Y > type;
};
```

## Header `<boost/units/dimensionless_type.hpp>`

Dimension lists in which all exponents resolve to zero reduce to `dimensionless_type`.

```
namespace boost {
    namespace units {
        struct dimensionless_type;
    }
}
```



## Struct `dimensionless_type`

`boost::units::dimensionless_type` — Dimension lists in which all exponents resolve to zero reduce to `dimensionless_type`.

## Synopsis

```
// In header: <boost/units/dimensionless_type.hpp>

struct dimensionless_type {
    // types
    typedef dimensionless_type type;
    typedef unspecified        tag;
    typedef mpl::long_< 0 >    size;
};
```

## Header `<boost/units/dimensionless_unit.hpp>`

Utility class to simplify construction of dimensionless units in a system.

```
namespace boost {
    namespace units {
        template<typename System> struct dimensionless_unit;
    }
}
```

## Struct template `dimensionless_unit`

`boost::units::dimensionless_unit` — Utility class to simplify construction of dimensionless units in a system.

## Synopsis

```
// In header: <boost/units/dimensionless_unit.hpp>

template<typename System>
struct dimensionless_unit {
    // types
    typedef unit< dimensionless_type, System > type;
};
```

## Header <boost/units/get\_dimension.hpp>

Get the dimension of a unit, absolute unit and quantity.

```
namespace boost {
    namespace units {
        template<typename T> struct get_dimension;

        template<typename Dim, typename System>
            struct get_dimension<unit< Dim, System >>;
        template<typename Unit> struct get_dimension<absolute< Unit >>;
        template<typename Unit, typename Y>
            struct get_dimension<quantity< Unit, Y >>;
    }
}
```

## Struct template `get_dimension`

`boost::units::get_dimension`

## Synopsis

```
// In header: <boost/units/get_dimension.hpp>

template<typename T>
struct get_dimension {
};
```

## Struct template `get_dimension<unit< Dim, System >>`

`boost::units::get_dimension<unit< Dim, System >>` — Get the dimension of a unit.

## Synopsis

```
// In header: <boost/units/get_dimension.hpp>

template<typename Dim, typename System>
struct get_dimension<unit< Dim, System >> {
    // types
    typedef Dim type;
};
```

## Struct template `get_dimension<absolute< Unit >>`

`boost::units::get_dimension<absolute< Unit >>` — Get the dimension of an absolute unit.

## Synopsis

```
// In header: <boost/units/get_dimension.hpp>

template<typename Unit>
struct get_dimension<absolute< Unit >> {
    // types
    typedef get_dimension< Unit >::type type;
};
```

## Struct template `get_dimension<quantity< Unit, Y >>`

`boost::units::get_dimension<quantity< Unit, Y >>` — Get the dimension of a quantity.

## Synopsis

```
// In header: <boost/units/get_dimension.hpp>

template<typename Unit, typename Y>
struct get_dimension<quantity< Unit, Y >> {
    // types
    typedef get_dimension< Unit >::type type;
};
```

## Header `<boost/units/get_system.hpp>`

Get the system of a unit, absolute unit or quantity.

```
namespace boost {
    namespace units {
        template<typename T> struct get_system;

        template<typename Dim, typename System>
            struct get_system<unit< Dim, System >>;
        template<typename Unit> struct get_system<absolute< Unit >>;
        template<typename Unit, typename Y> struct get_system<quantity< Unit, Y >>;
    }
}
```

## Struct template `get_system`

`boost::units::get_system`

## Synopsis

```
// In header: <boost/units/get_system.hpp>

template<typename T>
struct get_system {
};
```

## Struct template `get_system<unit< Dim, System >>`

`boost::units::get_system<unit< Dim, System >>` — Get the system of a unit.

## Synopsis

```
// In header: <boost/units/get_system.hpp>

template<typename Dim, typename System>
struct get_system<unit< Dim, System >> {
    // types
    typedef System type;
};
```



## Struct template `get_system<absolute< Unit >>`

`boost::units::get_system<absolute< Unit >>` — Get the system of an absolute unit.

## Synopsis

```
// In header: <boost/units/get_system.hpp>

template<typename Unit>
struct get_system<absolute< Unit >> {
    // types
    typedef get_system< Unit >::type type;
};
```

## Struct template `get_system<quantity< Unit, Y >>`

`boost::units::get_system<quantity< Unit, Y >>` — Get the system of a quantity.

## Synopsis

```
// In header: <boost/units/get_system.hpp>

template<typename Unit, typename Y>
struct get_system<quantity< Unit, Y >> {
    // types
    typedef get_system< Unit >::type type;
};
```

## Header `<boost/units/heterogeneous_system.hpp>`

A heterogeneous system is a sorted list of base unit/exponent pairs.

```
namespace boost {
    namespace mpl {
    }
    namespace units {
        template<typename T> struct heterogeneous_system;
    }
}
```

## Struct template heterogeneous\_system

boost::units::heterogeneous\_system

## Synopsis

```
// In header: <boost/units/heterogeneous_system.hpp>

template<typename T>
struct heterogeneous_system {
};
```

## Description

A system that can represent any possible combination of units at the expense of not preserving information about how it was created. Do not create specializations of this template directly. Instead use `reduce_unit` and `base_unit<...>unit_type`.

## Header <boost/units/homogeneous\_system.hpp>

```
namespace boost {
  namespace units {
    template<typename L> struct homogeneous_system;
  }
}
```

## Struct template homogeneous\_system

boost::units::homogeneous\_system

## Synopsis

```
// In header: <boost/units/homogeneous_system.hpp>

template<typename L>
struct homogeneous_system {
};
```

### Description

A system that can uniquely represent any unit which can be composed from a linearly independent set of base units. It is safe to rebind a unit with such a system to different dimensions.

Do not construct this template directly. Use `make_system` instead.

### Header <boost/units/io.hpp>

Stream input and output for rationals, units and quantities.

Functions and manipulators for output and input of units and quantities. symbol and name format, and engineering and binary autoprefix. Serialization output is also supported.

```

namespace boost {
    namespace serialization {

        // Boost Serialization library support for units.
        template<typename Archive, typename System, typename Dim>
        void serialize(Archive & ar, boost::units::unit< Dim, System > &,
            const unsigned int);

        // Boost Serialization library support for quantities.
        template<typename Archive, typename Unit, typename Y>
        void serialize(Archive & ar, boost::units::quantity< Unit, Y > & q,
            const unsigned int);
    }
    namespace units {
        template<typename BaseUnit> struct base_unit_info;

        // format of output of units, for example "m" or "meter".
        enum format_mode { symbol_fmt = 0, name_fmt = 1, raw_fmt = 2,
            typename_fmt = 3, fmt_mask = 3 };

        // automatic scaling and prefix (controlled by value of quantity) a, if any,
        enum autoprefix_mode { autoprefix_none = 0, autoprefix_engineering = 4,
            autoprefix_binary = 8, autoprefix_mask = 12 };
        template<typename T> std::string to_string(const T & t);

        // get string representation of integral-valued static_rational.
        template<integer_type N>
        std::string to_string(const static_rational< N > &);

        // get string representation of static_rational.
        template<integer_type N, integer_type D>
        std::string to_string(const static_rational< N, D > &);

        // Write static_rational to std::basic_ostream.
        template<typename Char, typename Traits, integer_type N, integer_type D>
        std::basic_ostream< Char, Traits > &
        operator<<(std::basic_ostream< Char, Traits > & os,
            const static_rational< N, D > & r);

        // returns flags controlling output.
        long get_flags(std::ios_base & ios, long mask);

        // Set new flags controlling output format.
        void set_flags(std::ios_base & ios, long new_flags, long mask);

        // returns flags controlling output format.
        format_mode get_format(std::ios_base & ios);

        // Set new flags controlling output format.
        void set_format(std::ios_base & ios, format_mode new_mode);

        // Set new flags for type_name output format.
        std::ios_base & typename_format(std::ios_base & ios);

        // set new flag for raw format output, for example "m".
        std::ios_base & raw_format(std::ios_base & ios);

        // set new format flag for symbol output, for example "m".
        std::ios_base & symbol_format(std::ios_base & ios);

        // set new format for name output, for example "meter".
        std::ios_base & name_format(std::ios_base & ios);
    }
}

```

```

// get autoprefix flags for output.
autoprefix_mode get_autoprefix(std::ios_base & ios);

// Get format for output.
void set_autoprefix(std::ios_base & ios, autoprefix_mode new_mode);

// Clear autoprefix flags.
std::ios_base & no_prefix(std::ios_base & ios);

// Set flag for engineering prefix, so 1234.5 m displays as "1.2345 km".
std::ios_base & engineering_prefix(std::ios_base & ios);

// Set flag for binary prefix, so 1024 byte displays as "1 Kib".
std::ios_base & binary_prefix(std::ios_base & ios);
template<typename Dimension, typename System>
    std::string typename_string(const unit< Dimension, System > &);
template<typename Dimension, typename System>
    std::string symbol_string(const unit< Dimension, System > &);
template<typename Dimension, typename System>
    std::string name_string(const unit< Dimension, System > &);
template<typename Char, typename Traits, typename Dimension,
        typename System>
    std::basic_ostream< Char, Traits > &
    operator<<(std::basic_ostream< Char, Traits > &,
        const unit< Dimension, System > &);
template<typename Char, typename Traits, typename Unit, typename T>
    std::basic_ostream< Char, Traits > &
    operator<<(std::basic_ostream< Char, Traits > &,
        const quantity< Unit, T > &);
}
}

```

## Struct template `base_unit_info`

`boost::units::base_unit_info` — traits template for unit names.

## Synopsis

```
// In header: <boost/units/io.hpp>

template<typename BaseUnit>
struct base_unit_info {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

### Description

#### `base_unit_info` public static functions

1. `static std::string name();`

The full name of the unit (returns `BaseUnit::name()` by default).

2. `static std::string symbol();`

The symbol for the base unit (Returns `BaseUnit::symbol()` by default).

Returns:      `BaseUnit::symbol()`, for example "m"

## Function template operator<<

boost::units::operator<<

## Synopsis

```
// In header: <boost/units/io.hpp>

template<typename Char, typename Traits, typename Dimension, typename System>
std::basic_ostream< Char, Traits > &
operator<<(std::basic_ostream< Char, Traits > & os,
          const unit< Dimension, System > & u);
```

## Description

Print a unit as a list of base units and their exponents.

for `symbol_format` outputs e.g. "m s<sup>-1</sup>" or "J". for `name_format` outputs e.g. "meter second<sup>-1</sup>" or "joule". for `raw_format` outputs e.g. "m s<sup>-1</sup>" or "meter kilogram<sup>2</sup> second<sup>-2</sup>". for `typename_format` outputs the typename itself (currently demangled only on GCC).



## Function template operator<<

boost::units::operator<< — Print a quantity.

## Synopsis

```
// In header: <boost/units/io.hpp>

template<typename Char, typename Traits, typename Unit, typename T>
std::basic_ostream< Char, Traits > &
operator<<(std::basic_ostream< Char, Traits > & os,
          const quantity< Unit, T > & q);
```

## Description

Prints the value followed by the unit. If the `engineering_prefix`, or `binary_prefix` is set, tries to scale the value appropriately. For example, it might print 12.345 km instead of 12345 m. (Note does **not** attempt to automatically scale scalars like double, float...)

## Header <boost/units/is\_dim.hpp>

Check that a type is a valid dim.

```
namespace boost {
  namespace units {
    template<typename T> struct is_dim;

    template<typename T, typename V> struct is_dim<dim< T, V >>;
  }
}
```

## Struct template is\_dim

boost::units::is\_dim — Check that a type is a valid dim.

## Synopsis

```
// In header: <boost/units/is_dim.hpp>

template<typename T>
struct is_dim {
};
```

## Struct template `is_dim<dim< T, V >>`

`boost::units::is_dim<dim< T, V >>`

## Synopsis

```
// In header: <boost/units/is_dim.hpp>

template<typename T, typename V>
struct is_dim<dim< T, V >> {
};
```

## Header `<boost/units/is_dimension_list.hpp>`

Check that a type is a valid dimension list.

```
namespace boost {
  namespace units {
    template<typename Seq> struct is_dimension_list;

    template<typename Item, typename Next>
      struct is_dimension_list<list< Item, Next >>;
    template<> struct is_dimension_list<dimensionless_type>;
  }
}
```

## Struct template `is_dimension_list`

`boost::units::is_dimension_list` — Check that a type is a valid dimension list.

## Synopsis

```
// In header: <boost/units/is_dimension_list.hpp>

template<typename Seq>
struct is_dimension_list {
};
```

**Struct template `is_dimension_list<list< Item, Next >>`**

`boost::units::is_dimension_list<list< Item, Next >>`

**Synopsis**

```
// In header: <boost/units/is_dimension_list.hpp>

template<typename Item, typename Next>
struct is_dimension_list<list< Item, Next >> {
};
```

## Struct `is_dimension_list<dimensionless_type>`

`boost::units::is_dimension_list<dimensionless_type>`

## Synopsis

```
// In header: <boost/units/is_dimension_list.hpp>

struct is_dimension_list<dimensionless_type> {
};
```

## Header `<boost/units/is_dimensionless.hpp>`

Check if a unit or quantity is dimensionless.

```
namespace boost {
  namespace units {
    template<typename T> struct is_dimensionless;

    template<typename System>
      struct is_dimensionless<unit< dimensionless_type, System >>;
    template<typename Unit, typename Y>
      struct is_dimensionless<quantity< Unit, Y >>;
  }
}
```

## Struct template `is_dimensionless`

`boost::units::is_dimensionless`

## Synopsis

```
// In header: <boost/units/is_dimensionless.hpp>

template<typename T>
struct is_dimensionless {
};
```

**Struct template `is_dimensionless<unit< dimensionless_type, System >>`**

`boost::units::is_dimensionless<unit< dimensionless_type, System >>` — Check if a unit is dimensionless.

## Synopsis

```
// In header: <boost/units/is_dimensionless.hpp>

template<typename System>
struct is_dimensionless<unit< dimensionless_type, System >> {
};
```



## Struct template `is_dimensionless<quantity< Unit, Y >>`

`boost::units::is_dimensionless<quantity< Unit, Y >>` — Check if a quantity is dimensionless.

## Synopsis

```
// In header: <boost/units/is_dimensionless.hpp>

template<typename Unit, typename Y>
struct is_dimensionless<quantity< Unit, Y >> : public boost::units::is_dimensionless< Unit > {
};
```

## Header `<boost/units/is_dimensionless_quantity.hpp>`

check that a type is a dimensionless quantity

```
namespace boost {
  namespace units {
    template<typename T> struct is_dimensionless_quantity;
  }
}
```

## Struct template `is_dimensionless_quantity`

`boost::units::is_dimensionless_quantity` — Check that a type is a dimensionless quantity.

## Synopsis

```
// In header: <boost/units/is_dimensionless_quantity.hpp>

template<typename T>
struct is_dimensionless_quantity :
    public boost::units::is_quantity_of_dimension< T, dimensionless_type >
{
};
```

## Header `<boost/units/is_dimensionless_unit.hpp>`

Check that a type is a dimensionless unit.

```
namespace boost {
    namespace units {
        template<typename T> struct is_dimensionless_unit;
    }
}
```

## Struct template `is_dimensionless_unit`

`boost::units::is_dimensionless_unit` — Check that a type is a dimensionless unit.

## Synopsis

```
// In header: <boost/units/is_dimensionless_unit.hpp>

template<typename T>
struct is_dimensionless_unit :
    public boost::units::is_unit_of_dimension< T, dimensionless_type >
{
};
```

## Header `<boost/units/is_quantity.hpp>`

Check that a type is a quantity.

```
namespace boost {
    namespace units {
        template<typename T> struct is_quantity;

        template<typename Unit, typename Y> struct is_quantity<quantity< Unit, Y >>;
    }
}
```

## Struct template `is_quantity`

`boost::units::is_quantity` — Check that a type is a quantity.

## Synopsis

```
// In header: <boost/units/is_quantity.hpp>

template<typename T>
struct is_quantity {
};
```

## Struct template `is_quantity<quantity< Unit, Y >>`

`boost::units::is_quantity<quantity< Unit, Y >>`

## Synopsis

```
// In header: <boost/units/is_quantity.hpp>

template<typename Unit, typename Y>
struct is_quantity<quantity< Unit, Y >> {
};
```

## Header `<boost/units/is_quantity_of_dimension.hpp>`

Check that a type is a quantity of the specified dimension.

```
namespace boost {
  namespace units {
    template<typename T, typename Dim> struct is_quantity_of_dimension;

    template<typename Unit, typename Y, typename Dim>
      struct is_quantity_of_dimension<quantity< Unit, Y >, Dim>;
  }
}
```

## Struct template `is_quantity_of_dimension`

`boost::units::is_quantity_of_dimension` — Check that a type is a quantity of the specified dimension.

## Synopsis

```
// In header: <boost/units/is_quantity_of_dimension.hpp>

template<typename T, typename Dim>
struct is_quantity_of_dimension {
};
```

## Struct template `is_quantity_of_dimension<quantity< Unit, Y >, Dim>`

`boost::units::is_quantity_of_dimension<quantity< Unit, Y >, Dim>`

## Synopsis

```
// In header: <boost/units/is_quantity_of_dimension.hpp>

template<typename Unit, typename Y, typename Dim>
struct is_quantity_of_dimension<quantity< Unit, Y >, Dim> :
    public boost::units::is_unit_of_dimension< Unit, Dim >
{
};
```

## Header `<boost/units/is_quantity_of_system.hpp>`

Check that a type is a quantity in a specified system.

```
namespace boost {
    namespace units {
        template<typename T, typename System> struct is_quantity_of_system;

        template<typename Unit, typename Y, typename System>
            struct is_quantity_of_system<quantity< Unit, Y >, System>;
    }
}
```

## Struct template `is_quantity_of_system`

`boost::units::is_quantity_of_system` — Check that a type is a quantity in a specified system.

## Synopsis

```
// In header: <boost/units/is_quantity_of_system.hpp>

template<typename T, typename System>
struct is_quantity_of_system {
};
```



## Struct template `is_quantity_of_system<quantity< Unit, Y >, System>`

`boost::units::is_quantity_of_system<quantity< Unit, Y >, System>`

## Synopsis

```
// In header: <boost/units/is_quantity_of_system.hpp>

template<typename Unit, typename Y, typename System>
struct is_quantity_of_system<quantity< Unit, Y >, System> :
    public boost::units::is_unit_of_system< Unit, System >
{
};
```

## Header `<boost/units/is_unit.hpp>`

Check that a type is a unit.

```
namespace boost {
    namespace units {
        template<typename T> struct is_unit;

        template<typename Dim, typename System> struct is_unit<unit< Dim, System >>;
    }
}
```

## Struct template `is_unit`

`boost::units::is_unit` — Check that a type is a unit.

## Synopsis

```
// In header: <boost/units/is_unit.hpp>

template<typename T>
struct is_unit {
};
```

## Struct template `is_unit<unit< Dim, System >>`

`boost::units::is_unit<unit< Dim, System >>`

## Synopsis

```
// In header: <boost/units/is_unit.hpp>

template<typename Dim, typename System>
struct is_unit<unit< Dim, System >> {
};
```

## Header `<boost/units/is_unit_of_dimension.hpp>`

Check that a type is a unit of the specified dimension.

```
namespace boost {
  namespace units {
    template<typename T, typename Dim> struct is_unit_of_dimension;

    template<typename Dim, typename System>
      struct is_unit_of_dimension<unit< Dim, System >, Dim>;
    template<typename Dim, typename System>
      struct is_unit_of_dimension<absolute< unit< Dim, System > >, Dim>;
  }
}
```

## Struct template `is_unit_of_dimension`

`boost::units::is_unit_of_dimension` — Check that a type is a unit of the specified dimension.

## Synopsis

```
// In header: <boost/units/is_unit_of_dimension.hpp>

template<typename T, typename Dim>
struct is_unit_of_dimension {
};
```

**Struct template `is_unit_of_dimension<unit< Dim, System >, Dim>`**

`boost::units::is_unit_of_dimension<unit< Dim, System >, Dim>`

**Synopsis**

```
// In header: <boost/units/is_unit_of_dimension.hpp>

template<typename Dim, typename System>
struct is_unit_of_dimension<unit< Dim, System >, Dim> {
};
```

**Struct template `is_unit_of_dimension<absolute< unit< Dim, System > >, Dim>`**

`boost::units::is_unit_of_dimension<absolute< unit< Dim, System > >, Dim>`

## Synopsis

```
// In header: <boost/units/is_unit_of_dimension.hpp>

template<typename Dim, typename System>
struct is_unit_of_dimension<absolute< unit< Dim, System > >, Dim> {
};
```

**Header `<boost/units/is_unit_of_system.hpp>`**

Check that a type is a unit in a specified system.

```
namespace boost {
  namespace units {
    template<typename T, typename System> struct is_unit_of_system;

    template<typename Dim, typename System>
      struct is_unit_of_system<unit< Dim, System >, System>;
    template<typename Dim, typename System>
      struct is_unit_of_system<absolute< unit< Dim, System > >, System>;
  }
}
```

## Struct template `is_unit_of_system`

`boost::units::is_unit_of_system` — Check that a type is a unit in a specified system.

## Synopsis

```
// In header: <boost/units/is_unit_of_system.hpp>

template<typename T, typename System>
struct is_unit_of_system {
};
```

**Struct template `is_unit_of_system<unit< Dim, System >, System>`**

`boost::units::is_unit_of_system<unit< Dim, System >, System>`

## Synopsis

```
// In header: <boost/units/is_unit_of_system.hpp>

template<typename Dim, typename System>
struct is_unit_of_system<unit< Dim, System >, System> {
};
```



**Struct template `is_unit_of_system<absolute< unit< Dim, System > >, System>`**

`boost::units::is_unit_of_system<absolute< unit< Dim, System > >, System>`

## Synopsis

```
// In header: <boost/units/is_unit_of_system.hpp>

template<typename Dim, typename System>
struct is_unit_of_system<absolute< unit< Dim, System > >, System> {
};
```

**Header `<boost/units/lambda.hpp>`**

Definitions to ease the usage of Boost.Units' quantity, unit, and absolute types in functors created with the Boost.Lambda library.

Torsten Maehne

2008-06-16

Boost.Lambda's return type deduction system is extended to make use of Boost.Units' `typeof_helper` trait classes for Boost.Units' quantity, absolute, and unit template classes.

```

namespace boost {
    namespace lambda {
        template<typename System, typename Dim, typename Y>
            struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::unit< Dim, Sys-
tem >, Y>;
        template<typename System, typename Dim, typename Y>
            struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim, Sys-
tem >, Y>;
        template<typename System, typename Dim, typename Y>
            struct plain_return_type_2<arithmetic_action< multiply_ac-
tion >, Y, boost::units::unit< Dim, System >>;
        template<typename System, typename Dim, typename Y>
            struct plain_return_type_2<arithmetic_action< divide_ac-
tion >, Y, boost::units::unit< Dim, System >>;
        template<typename Unit, typename X>
            struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::quant-
ity< Unit, X >, X>;
        template<typename Unit, typename X>
            struct plain_return_type_2<arithmetic_action< multiply_action >, X, boost::units::quant-
ity< Unit, X >>;
        template<typename Unit, typename X>
            struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::quant-
ity< Unit, X >, X>;
        template<typename Unit, typename X>
            struct plain_return_type_2<arithmetic_action< divide_action >, X, boost::units::quant-
ity< Unit, X >>;
        template<typename System1, typename Dim1, typename Unit2, typename Y>
            struct plain_return_type_2<arithmetic_action< multiply_ac-
tion >, boost::units::unit< Dim1, System1 >, boost::units::quantity< Unit2, Y >>;
        template<typename System1, typename Dim1, typename Unit2, typename Y>
            struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim1, Sys-
tem1 >, boost::units::quantity< Unit2, Y >>;
        template<typename Unit1, typename Y, typename System2, typename Dim2>
            struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::quant-
ity< Unit1, Y >, boost::units::unit< Dim2, System2 >>;
        template<typename Unit1, typename Y, typename System2, typename Dim2>
            struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::quant-
ity< Unit1, Y >, boost::units::unit< Dim2, System2 >>;
        template<typename Unit, typename Y>
            struct plain_return_type_1<unary_arithmetic_action< plus_action >, boost::units::quant-
ity< Unit, Y >>;
        template<typename Unit, typename Y>
            struct plain_return_type_1<unary_arithmetic_action< minus_action >, boost::units::quant-
ity< Unit, Y >>;
        template<typename Unit1, typename X, typename Unit2, typename Y>
            struct plain_return_type_2<arithmetic_action< plus_action >, boost::units::quant-
ity< Unit1, X >, boost::units::quantity< Unit2, Y >>;
        template<typename System, typename X, typename Y>
            struct plain_return_type_2<arithmetic_action< plus_action >, boost::units::quant-
ity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), X >, Y>;
        template<typename System, typename X, typename Y>
            struct plain_return_type_2<arithmetic_action< plus_action >, X, boost::units::quant-
ity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y >>;
        template<typename Unit1, typename X, typename Unit2, typename Y>
            struct plain_return_type_2<arithmetic_action< minus_action >, boost::units::quant-
ity< Unit1, X >, boost::units::quantity< Unit2, Y >>;
        template<typename System, typename X, typename Y>
            struct plain_return_type_2<arithmetic_action< minus_action >, boost::units::quant-
ity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), X >, Y>;
        template<typename System, typename X, typename Y>
            struct plain_return_type_2<arithmetic_action< minus_action >, X, boost::units::quant-
ity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y >>;
        template<typename Unit1, typename X, typename Unit2, typename Y>

```

```

        struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>;
        template<typename Unit1, typename X, typename Unit2, typename Y>
            struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>;
        template<typename Dim, typename System>
            struct plain_return_type_1<unary_arithmetic_action< plus_action >, boost::units::unit< Dim, System >>;
        template<typename Dim, typename System>
            struct plain_return_type_1<unary_arithmetic_action< minus_action >, boost::units::unit< Dim, System >>;
        template<typename Dim1, typename Dim2, typename System1, typename System2>
            struct plain_return_type_2<arithmetic_action< plus_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>;
        template<typename Dim1, typename Dim2, typename System1, typename System2>
            struct plain_return_type_2<arithmetic_action< minus_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>;
        template<typename Dim1, typename Dim2, typename System1, typename System2>
            struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>;
        template<typename Dim1, typename Dim2, typename System1, typename System2>
            struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>;
        template<typename Y>
            struct plain_return_type_2<arithmetic_action< plus_action >, boost::units::absolute< Y >, Y>;
        template<typename Y>
            struct plain_return_type_2<arithmetic_action< plus_action >, Y, boost::units::absolute< Y >>;
        template<typename Y>
            struct plain_return_type_2<arithmetic_action< minus_action >, boost::units::absolute< Y >, Y>;
        template<typename Y>
            struct plain_return_type_2<arithmetic_action< minus_action >, boost::units::absolute< Y >, Y>;
        template<typename D, typename S, typename T>
            struct plain_return_type_2<arithmetic_action< multiply_action >, T, boost::units::absolute< boost::units::unit< D, S >>>;
        template<typename D, typename S, typename T>
            struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::absolute< boost::units::unit< D, S >>, T>;
    }
    namespace units {
        template<typename System, typename Dim, typename Arg>
            struct multiply_typeof_helper<boost::units::unit< Dim, System >, boost::lambda::lambda_functor< Arg >>;
        template<typename System, typename Dim, typename Arg>
            struct divide_typeof_helper<boost::units::unit< Dim, System >, boost::lambda::lambda_functor< Arg >>;
        template<typename System, typename Dim, typename Arg>
            struct multiply_typeof_helper<boost::lambda::lambda_functor< Arg >, boost::units::unit< Dim, System >>;
        template<typename System, typename Dim, typename Arg>
            struct divide_typeof_helper<boost::lambda::lambda_functor< Arg >, boost::units::unit< Dim, System >>;
        template<typename System, typename Dim, typename Arg>
            struct multiply_typeof_helper<boost::lambda::lambda_functor< Arg >, boost::units::absolute< boost::units::unit< Dim, System >>>;
        template<typename System, typename Dim, typename Arg>
            struct divide_typeof_helper<boost::lambda::lambda_functor< Arg >, boost::units::absolute< boost::units::unit< Dim, System >>>;
        const multiply_typeof_helper< boost::units::unit< Dim, System >, boost::lambda::lambda_functor< Arg >>::type
        operator*(const boost::units::unit< Dim, System > &,

```

```

        const boost::lambda::lambda_functor< Arg > &);
template<typename System, typename Dim, typename Arg>
    const divide_typeof_helper< boost::units::unit< Dim, System >, boost::lambda::lambda_functor< Arg > >::type
operator/(const boost::units::unit< Dim, System > &,
        const boost::lambda::lambda_functor< Arg > &);
template<typename System, typename Dim, typename Arg>
    const multiply_typeof_helper< boost::lambda::lambda_functor< Arg >, boost::units::unit< Dim, System > >::type
operator*(const boost::lambda::lambda_functor< Arg > &,
        const boost::units::unit< Dim, System > &);
template<typename System, typename Dim, typename Arg>
    const divide_typeof_helper< boost::lambda::lambda_functor< Arg >, boost::units::unit< Dim, System > >::type
operator/(const boost::lambda::lambda_functor< Arg > &,
        const boost::units::unit< Dim, System > &);
template<typename System, typename Dim, typename Arg>
    const multiply_typeof_helper< boost::lambda::lambda_functor< Arg >, boost::units::absolute< boost::units::unit< Dim, System > > >::type
operator*(const boost::lambda::lambda_functor< Arg > &,
        const boost::units::absolute< boost::units::unit< Dim, System > > &);
template<typename System, typename Dim, typename Arg>
    const multiply_typeof_helper< boost::units::absolute< boost::units::unit< Dim, System > >, boost::lambda::lambda_functor< Arg > >::type
operator*(const boost::units::absolute< boost::units::unit< Dim, System > > &,
        const boost::lambda::lambda_functor< Arg > &);
    }
}

```

## Struct template `plain_return_type_2<arithmetic_action< multiply_action >, boost::units::unit< Dim, System >, Y>`

`boost::lambda::plain_return_type_2<arithmetic_action< multiply_action >, boost::units::unit< Dim, System >, Y>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Y>
struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::unit< Dim, System >, Y> {
    // types
    typedef boost::units::multiply_typeof_helper< boost::units::unit< Dim, System >, Y >::type type;
};
```

## Description

Partial specialization of return type trait for action `unit<Dim, System> * Y`.

**Struct template `plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim, System >, Y>`**

`boost::lambda::plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim, System >, Y>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Y>
struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim, System >, Y> {
    // types
    typedef boost::units::divide_typeof_helper< boost::units::unit< Dim, System >, Y >::type type;
};
```

## Description

Partial specialization of return type trait for action `unit<Dim, System> / Y`.

**Struct template `plain_return_type_2<arithmetic_action< multiply_action >, Y, boost::units::unit< Dim, System >>`**

`boost::lambda::plain_return_type_2<arithmetic_action< multiply_action >, Y, boost::units::unit< Dim, System >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Y>
struct plain_return_type_2<arithmetic_action< multiply_action >, Y, boost::units::unit< Dim, System >> {
    // types
    typedef boost::units::multiply_typeof_helper< Y, boost::units::unit< Dim, System > >::type type;
};
```

## Description

Partial specialization of return type trait for action `Y * unit<Dim, System>`.

**Struct template `plain_return_type_2<arithmetic_action< divide_action >, Y, boost::units::unit< Dim, System >>`**

`boost::lambda::plain_return_type_2<arithmetic_action< divide_action >, Y, boost::units::unit< Dim, System >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Y>
struct plain_return_type_2<arithmetic_action< divide_action >, Y, boost::units::unit< Dim, System >> {
    // types
    typedef boost::units::divide_typeof_helper< Y, boost::units::unit< Dim, System > >::type type;
};
```

## Description

Partial specialization of return type trait for action `Y / unit<Dim, System>`.



**Struct**      **template**      **plain\_return\_type\_2**<arithmetic\_action<      **multiply\_action**      >, boost::units::quantity< Unit, X >, X>

boost::lambda::plain\_return\_type\_2<arithmetic\_action< multiply\_action >, boost::units::quantity< Unit, X >, X>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit, typename X>
struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::quantity< Unit, X >, X> {
    // types
    typedef boost::units::multiply_typeof_helper< boost::units::quantity< Unit, X >, X >::type type;
};
```

## Description

Partial specialization of return type trait for action quantity<Unit, X> \* X.

**Struct**    **template**    **plain\_return\_type\_2**<arithmetic\_action<    **multiply\_action**    >,    **X**,  
**boost::units::quantity**< **Unit**, **X** >>

boost::lambda::plain\_return\_type\_2<arithmetic\_action< multiply\_action >, X, boost::units::quantity< Unit, X >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit, typename X>
struct plain_return_type_2<arithmetic_action< multiply_action >, X, boost::units::quantity< Unit, X >> {
    // types
    typedef boost::units::multiply_typeof_helper< X, boost::units::quantity< Unit, X > >::type type;
};
```

## Description

Partial specialization of return type trait for action  $X * \text{quantity}\langle \text{Unit}, X \rangle$ .

## Struct template `plain_return_type_2<arithmetic_action< divide_action >, boost::units::quantity< Unit, X >, X>`

`boost::lambda::plain_return_type_2<arithmetic_action< divide_action >, boost::units::quantity< Unit, X >, X>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit, typename X>
struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::quantity< Unit, X >, X> {
    // types
    typedef boost::units::divide_typeof_helper< boost::units::quantity< Unit, X >, X >::type type;
};
```

## Description

Partial specialization of return type trait for action `quantity<Unit, X> / X`.

**Struct**    **template**    **plain\_return\_type\_2**<arithmetic\_action<    **divide\_action**    >,    **X**,  
**boost::units::quantity**< **Unit**, **X** >>

boost::lambda::plain\_return\_type\_2<arithmetic\_action< divide\_action >, X, boost::units::quantity< Unit, X >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit, typename X>
struct plain_return_type_2<arithmetic_action< divide_action >, X, boost::units::quantity< Unit, X >> {
    // types
    typedef boost::units::divide_typeof_helper< X, boost::units::quantity< Unit, X > >::type type;
};
```

## Description

Partial specialization of return type trait for action X / quantity<Unit, X>.

**Struct template `plain_return_type_2<arithmetic_action< multiply_action >, boost::units::unit< Dim1, System1 >, boost::units::quantity< Unit2, Y >>`**

`boost::lambda::plain_return_type_2<arithmetic_action< multiply_action >, boost::units::unit< Dim1, System1 >, boost::units::quantity< Unit2, Y >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System1, typename Dim1, typename Unit2, typename Y>
struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::unit< Dim1, System1 >, boost::units::quantity< Unit2, Y >> {
    // types
    typedef boost::units::multiply_typeof_helper< boost::units::unit< Dim1, System1 >, boost::units::quantity< Unit2, Y >>::type type;
};
```

## Description

Partial specialization of return type trait for action `unit<Dim1, System1> * quantity<Unit2, Y>`.

**Struct template `plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim1, System1 >, boost::units::quantity< Unit2, Y >>`**

`boost::lambda::plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim1, System1 >, boost::units::quantity< Unit2, Y >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System1, typename Dim1, typename Unit2, typename Y>
struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim1, System1 >, boost::units::quantity< Unit2, Y >> {
    // types
    typedef boost::units::divide_typeof_helper< boost::units::unit< Dim1, System1 >, boost::units::quantity< Unit2, Y > >::type type;
};
```

## Description

Partial specialization of return type trait for action `unit<Dim1, System1>` / `quantity<Unit2, Y>`.

**Struct      template      plain\_return\_type\_2<arithmetic\_action<      multiply\_action      >, boost::units::quantity< Unit1, Y >, boost::units::unit< Dim2, System2 >>**

boost::lambda::plain\_return\_type\_2<arithmetic\_action< multiply\_action >, boost::units::quantity< Unit1, Y >, boost::units::unit< Dim2, System2 >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit1, typename Y, typename System2, typename Dim2>
struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::quantity< Unit1, Y >, boost::units::unit< Dim2, System2 >> {
    // types
    typedef boost::units::multiply_typeof_helper< boost::units::quantity< Unit1, Y >, boost::units::unit< Dim2, System2 >>::type type;
};
```

## Description

Partial specialization of return type trait for action quantity<Unit1, Y> \* unit<Dim2, System2>.

## Struct template `plain_return_type_2<arithmetic_action< divide_action >, boost::units::quantity< Unit1, Y >, boost::units::unit< Dim2, System2 >>`

`boost::lambda::plain_return_type_2<arithmetic_action< divide_action >, boost::units::quantity< Unit1, Y >, boost::units::unit< Dim2, System2 >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit1, typename Y, typename System2, typename Dim2>
struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::quantity< Unit1, Y >, boost::units::unit< Dim2, System2 >> {
    // types
    typedef boost::units::divide_typeof_helper< boost::units::quantity< Unit1, Y >, boost::units::unit< Dim2, System2 >>::type type;
};
```

## Description

Partial specialization of return type trait for action quantity<Unit1, Y> / unit<Dim2, System2>.



**Struct**    **template**    **plain\_return\_type\_1**<unary\_arithmetic\_action<    **plus\_action**    >, boost::units::quantity< Unit, Y >>

boost::lambda::plain\_return\_type\_1<unary\_arithmetic\_action< plus\_action >, boost::units::quantity< Unit, Y >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit, typename Y>
struct plain_return_type_1<unary_arithmetic_action< plus_action >, boost::units::quantity< Unit, Y >> {
    // types
    typedef boost::units::unary_plus_typeof_helper< boost::units::quantity< Unit, Y > >::type type;
};
```

## Description

Partial specialization of return type trait for action +quantity<Unit, Y>.

**Struct**    **template**    **plain\_return\_type\_1**<unary\_arithmetic\_action<    **minus\_action**    >, boost::units::quantity< Unit, Y >>

boost::lambda::plain\_return\_type\_1<unary\_arithmetic\_action< minus\_action >, boost::units::quantity< Unit, Y >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit, typename Y>
struct plain_return_type_1<unary_arithmetic_action< minus_action >, boost::units::quantity< Unit, Y >> {
    // types
    typedef boost::units::unary_minus_typeof_helper< boost::units::quantity< Unit, Y > >::type type;
};
```

## Description

Partial specialization of return type trait for action -quantity<Unit, Y>.

## Struct template `plain_return_type_2<arithmetic_action< plus_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>`

`boost::lambda::plain_return_type_2<arithmetic_action< plus_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit1, typename X, typename Unit2, typename Y>
struct plain_return_type_2<arithmetic_action< plus_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >> {
    // types
    typedef boost::units::add_typeof_helper< boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>::type type;
};
```

## Description

Partial specialization of return type trait for action `quantity<Unit1, X> + quantity<Unit2, Y>`.

## Struct template `plain_return_type_2<arithmetic_action< plus_action >, boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), X >, Y>`

`boost::lambda::plain_return_type_2<arithmetic_action< plus_action >, boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), X >, Y>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename X, typename Y>
struct plain_return_type_2<arithmetic_action< plus_action >, boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), X >, Y> {
    // types
    typedef boost::units::add_typeof_helper< boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), X >, Y >::type type;
};
```

## Description

Partial specialization of return type trait for action quantity<dimensionless, X> + Y.

**Struct**    **template**    **plain\_return\_type\_2**<arithmetic\_action<    **plus\_action**    >,    **X**,  
**boost::units::quantity**< **BOOST\_UNITS\_DIMENSIONLESS\_UNIT**(System), Y >>

boost::lambda::plain\_return\_type\_2<arithmetic\_action< plus\_action >, X, boost::units::quantity< BOOST\_UNITS\_DIMENSIONLESS\_UNIT(System), Y >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename X, typename Y>
struct plain_return_type_2<arithmetic_action< plus_action >, X, boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y >> {
    // types
    typedef boost::units::add_typeof_helper< X, boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y >>::type type;
};
```

## Description

Partial specialization of return type trait for action X + quantity<dimensionless, Y>.

## Struct template `plain_return_type_2<arithmetic_action< minus_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>`

`boost::lambda::plain_return_type_2<arithmetic_action< minus_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit1, typename X, typename Unit2, typename Y>
struct plain_return_type_2<arithmetic_action< minus_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >> {
    // types
    typedef boost::units::subtract_typeof_helper< boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>::type type;
};
```

## Description

Partial specialization of return type trait for action `quantity<Unit1, X> - quantity<Unit2, Y>`.

## Struct template `plain_return_type_2<arithmetic_action< minus_action >, boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), X >, Y>`

`boost::lambda::plain_return_type_2<arithmetic_action< minus_action >, boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), X >, Y>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename X, typename Y>
struct plain_return_type_2<arithmetic_action< minus_action >, boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), X >, Y> {
    // types
    typedef boost::units::subtract_typeof_helper< boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), X >, Y >::type type;
};
```

## Description

Partial specialization of return type trait for action `quantity<dimensionless, X> - Y`.

**Struct    template    plain\_return\_type\_2<arithmetic\_action<    minus\_action    >,    X,  
boost::units::quantity< BOOST\_UNITS\_DIMENSIONLESS\_UNIT(System), Y >>**

boost::lambda::plain\_return\_type\_2<arithmetic\_action< minus\_action >, X, boost::units::quantity< BOOST\_UNITS\_DIMENSIONLESS\_UNIT(System), Y >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename X, typename Y>
struct plain_return_type_2<arithmetic_action< minus_action >, X, boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y >> {
    // types
    typedef boost::units::subtract_typeof_helper< X, boost::units::quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y >>::type type;
};
```

## Description

Partial specialization of return type trait for action X - quantity<dimensionless, Y>.



**Struct**      **template**      **plain\_return\_type\_2**<arithmetic\_action<      **multiply\_action**      >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>

boost::lambda::plain\_return\_type\_2<arithmetic\_action< multiply\_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit1, typename X, typename Unit2, typename Y>
struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >> {
    // types
    typedef boost::units::multiply_typeof_helper< boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>::type type;
};
```

## Description

Partial specialization of return type trait for action quantity<Unit1, X> \* quantity<Unit2, Y>.

## Struct template `plain_return_type_2<arithmetic_action< divide_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>`

`boost::lambda::plain_return_type_2<arithmetic_action< divide_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Unit1, typename X, typename Unit2, typename Y>
struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >> {
    // types
    typedef boost::units::divide_typeof_helper< boost::units::quantity< Unit1, X >, boost::units::quantity< Unit2, Y >>::type type;
};
```

## Description

Partial specialization of return type trait for action `quantity<Unit1, X> / quantity<Unit2, Y>`.

**Struct    template    plain\_return\_type\_1<unary\_arithmetic\_action<    plus\_action    >, boost::units::unit< Dim, System >>**

boost::lambda::plain\_return\_type\_1<unary\_arithmetic\_action< plus\_action >, boost::units::unit< Dim, System >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Dim, typename System>
struct plain_return_type_1<unary_arithmetic_action< plus_action >, boost::units::unit< Dim, System >> {
    // types
    typedef boost::units::unary_plus_typeof_helper< boost::units::unit< Dim, System > >::type type;
};
```

## Description

Partial specialization of return type trait for action +unit<Dim, System>.

**Struct**    **template**    **plain\_return\_type\_1**<unary\_arithmetic\_action<    **minus\_action**    >, boost::units::unit< Dim, System >>

boost::lambda::plain\_return\_type\_1<unary\_arithmetic\_action< minus\_action >, boost::units::unit< Dim, System >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Dim, typename System>
struct plain_return_type_1<unary_arithmetic_action< minus_action >, boost::units::unit< Dim, System >> {
    // types
    typedef boost::units::unary_minus_typeof_helper< boost::units::unit< Dim, System > >::type type;
};
```

## Description

Partial specialization of return type trait for action -unit<Dim, System>.

**Struct template `plain_return_type_2<arithmetic_action< plus_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>`**

`boost::lambda::plain_return_type_2<arithmetic_action< plus_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Dim1, typename Dim2, typename System1, typename System2>
struct plain_return_type_2<arithmetic_action< plus_action >, boost::units::unit< Dim1, Sys-
tem1 >, boost::units::unit< Dim2, System2 >> {
    // types
    typedef boost::units::add_typeof_helper< boost::units::unit< Dim1, Sys-
tem1 >, boost::units::unit< Dim2, System2 > >::type type;
};
```

## Description

Partial specialization of return type trait for action `unit<Dim1, System1> + unit<Dim2, System2>`.

## Struct template `plain_return_type_2<arithmetic_action< minus_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>`

`boost::lambda::plain_return_type_2<arithmetic_action< minus_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Dim1, typename Dim2, typename System1, typename System2>
struct plain_return_type_2<arithmetic_action< minus_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >> {
    // types
    typedef boost::units::subtract_typeof_helper< boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 > >::type type;
};
```

## Description

Partial specialization of return type trait for action `unit<Dim1, System1> - unit<Dim2, System2>`.

**Struct template plain\_return\_type\_2<arithmetic\_action< multiply\_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>**

boost::lambda::plain\_return\_type\_2<arithmetic\_action< multiply\_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Dim1, typename Dim2, typename System1, typename System2>
struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >> {
    // types
    typedef boost::units::multiply_typeof_helper< boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 > >::type type;
};
```

## Description

Partial specialization of return type trait for action unit<Dim1, System1> \* unit<Dim2, System2>.

**Struct template `plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>`**

`boost::lambda::plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Dim1, typename Dim2, typename System1, typename System2>
struct plain_return_type_2<arithmetic_action< divide_action >, boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 >> {
    // types
    typedef boost::units::divide_typeof_helper< boost::units::unit< Dim1, System1 >, boost::units::unit< Dim2, System2 > >::type type;
};
```

## Description

Partial specialization of return type trait for action `unit<Dim1, System1> / unit<Dim2, System2>`.



**Struct template `plain_return_type_2<arithmetic_action< plus_action >, boost::units::absolute< Y >, Y>`**

`boost::lambda::plain_return_type_2<arithmetic_action< plus_action >, boost::units::absolute< Y >, Y>`

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Y>
struct plain_return_type_2<arithmetic_action< plus_action >, boost::units::absolute< Y >, Y> {
    // types
    typedef boost::units::absolute< Y > type;
};
```

## Description

Partial specialization of return type trait for action `absolute<Y> + Y`.

**Struct**    **template**    **plain\_return\_type\_2**<arithmetic\_action<    **plus\_action**    >,    **Y**,  
**boost::units::absolute**<Y >>

boost::lambda::plain\_return\_type\_2<arithmetic\_action< plus\_action >, Y, boost::units::absolute< Y >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Y>
struct plain_return_type_2<arithmetic_action< plus_action >, Y, boost::units::absolute< Y >> {
    // types
    typedef boost::units::absolute< Y > type;
};
```

## Description

Partial specialization of return type trait for action Y + absolute<Y>.

**Struct**      **template**      **plain\_return\_type\_2**<arithmetic\_action<      **minus\_action**      >, **boost::units::absolute**< Y >, Y>

boost::lambda::plain\_return\_type\_2<arithmetic\_action< minus\_action >, boost::units::absolute< Y >, Y>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Y>
struct plain_return_type_2<arithmetic_action< minus_action >, boost::units::absolute< Y >, Y> {
    // types
    typedef boost::units::absolute< Y > type;
};
```

## Description

Partial specialization of return type trait for action absolute<Y> - Y.

**Struct**      **template**      **plain\_return\_type\_2**<arithmetic\_action<      **minus\_action**      >, boost::units::absolute< Y >, boost::units::absolute< Y >>

boost::lambda::plain\_return\_type\_2<arithmetic\_action< minus\_action >, boost::units::absolute< Y >, boost::units::absolute< Y >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename Y>
struct plain_return_type_2<arithmetic_action< minus_action >, boost::units::absolute< Y >, boost::units::absolute< Y >> {
    // types
    typedef Y type;
};
```

## Description

Partial specialization of return type trait for action absolute<Y> - absolute<Y>.

**Struct**    **template**    **plain\_return\_type\_2**<arithmetic\_action<    **multiply\_action**    >,    **T**,  
**boost::units::absolute**< **boost::units::unit**< **D**, **S** > >>

boost::lambda::plain\_return\_type\_2<arithmetic\_action< multiply\_action >, T, boost::units::absolute< boost::units::unit< D, S > >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename D, typename S, typename T>
struct plain_return_type_2<arithmetic_action< multiply_action >, T, boost::units::abso↓
lute< boost::units::unit< D, S > >> {
    // types
    typedef boost::units::quantity< boost::units::absolute< boost::units::unit< D, S > >, T > type;
};
```

## Description

Partial specialization of return type trait for action  $T * \text{absolute}\langle \text{unit}\langle D, S \rangle \rangle$ .

**Struct**     **template**     **plain\_return\_type\_2**<arithmetic\_action<     **multiply\_action**     >, boost::units::absolute< boost::units::unit< D, S > >, T>

boost::lambda::plain\_return\_type\_2<arithmetic\_action< multiply\_action >, boost::units::absolute< boost::units::unit< D, S > >, T>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename D, typename S, typename T>
struct plain_return_type_2<arithmetic_action< multiply_action >, boost::units::absolute< boost::units::unit< D, S > >, T> {
    // types
    typedef boost::units::quantity< boost::units::absolute< boost::units::unit< D, S > >, T > type;
};
```

## Description

Partial specialization of return type trait for action absolute<unit<D, S> > \* T.

**Struct    template    multiply\_typeof\_helper<boost::units::unit<    Dim,    System    >, boost::lambda::lambda\_functor< Arg >>**

boost::units::multiply\_typeof\_helper<boost::units::unit< Dim, System >, boost::lambda::lambda\_functor< Arg >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
struct multiply_typeof_helper<boost::units::unit< Dim, System >, boost::lambda::lambda_functor< Arg >> {
    // types
    typedef boost::lambda::lambda_functor< boost::lambda::lambda_functor_base< boost::lambda::arithmetic_action< boost::lambda::multiply_action >, tuple< typename boost::lambda::const_copy_argument< const boost::units::unit< Dim, System > >::type, boost::lambda::lambda_functor< Arg > > > type;
};
```

**Struct      template      divide\_typeof\_helper<boost::units::unit<      Dim,      System      >,      boost::lambda::lambda\_functor< Arg >>**

boost::units::divide\_typeof\_helper<boost::units::unit< Dim, System >, boost::lambda::lambda\_functor< Arg >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
struct divide_typeof_helper<boost::units::unit< Dim, System >, boost::lambda::lambda_functor< Arg >> {
    // types
    typedef boost::lambda::lambda_functor< boost::lambda::lambda_functor_base< boost::lambda::arithmetic_action< boost::lambda::divide_action >, tuple< typename boost::lambda::const_copy_argument< const boost::units::unit< Dim, System > >::type, boost::lambda::lambda_functor< Arg > > > type;
};
```



**Struct    template    multiply\_typeof\_helper<boost::lambda::lambda\_functor<    Arg    >, boost::units::unit< Dim, System >>**

boost::units::multiply\_typeof\_helper<boost::lambda::lambda\_functor< Arg >, boost::units::unit< Dim, System >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
struct multiply_typeof_helper<boost::lambda::lambda_functor< Arg >, boost::units::unit< Dim, System >> {
    // types
    typedef boost::lambda::lambda_functor< boost::lambda::lambda_functor_base< boost::lambda::arithmetic_action< boost::lambda::multiply_action >, tuple< boost::lambda::lambda_functor< Arg >, typename boost::lambda::const_copy_argument< const boost::units::unit< Dim, System >>::type >> > type;
};
```

**Struct      template      divide\_typeof\_helper<boost::lambda::lambda\_functor<      Arg      >, boost::units::unit< Dim, System >>**

boost::units::divide\_typeof\_helper<boost::lambda::lambda\_functor< Arg >, boost::units::unit< Dim, System >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
struct divide_typeof_helper<boost::lambda::lambda_functor< Arg >, boost::units::unit< Dim, System >> {
    // types
    typedef boost::lambda::lambda_functor< boost::lambda::lambda_functor_base< boost::lambda::arithmetic_action< boost::lambda::divide_action >, tuple< boost::lambda::lambda_functor< Arg >, typename boost::lambda::const_copy_argument< const boost::units::unit< Dim, System >>::type >> > type;
};
```

**Struct    template    multiply\_typeof\_helper<boost::lambda::lambda\_functor<    Arg    >, boost::units::absolute< boost::units::unit< Dim, System > >>**

boost::units::multiply\_typeof\_helper<boost::lambda::lambda\_functor<Arg>, boost::units::absolute< boost::units::unit< Dim, System > >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
struct multiply_typeof_helper<boost::lambda::lambda_functor< Arg >, boost::units::absolute< boost::units::unit< Dim, System > >> {
    // types
    typedef boost::lambda::lambda_functor< boost::lambda::lambda_functor_base< boost::lambda::arithmetic_action< boost::lambda::multiply_action >, tuple< boost::lambda::lambda_functor< Arg >, typename boost::lambda::const_copy_argument< const boost::units::absolute< boost::units::unit< Dim, System > >>::type > > type;
};
```

**Struct template multiply\_typeof\_helper<boost::units::absolute< boost::units::unit< Dim, System > >, boost::lambda::lambda\_functor< Arg >>**

boost::units::multiply\_typeof\_helper<boost::units::absolute< boost::units::unit< Dim, System > >, boost::lambda::lambda\_functor< Arg >>

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
struct multiply_typeof_helper<boost::units::absolute< boost::units::unit< Dim, Sys↓
tem > >, boost::lambda::lambda_functor< Arg >> {
    // types
    typedef boost::lambda::lambda_functor< boost::lambda::lambda_functor_base< boost::lambda::arith↓
metic_action< boost::lambda::multiply_action >, tuple< typename boost::lambda::const_copy_argu↓
ment< const boost::units::absolute< boost::units::unit< Dim, Sys↓
tem > > >::type, boost::lambda::lambda_functor< Arg > > > > type;
};
```

## Function template operator\*

boost::units::operator\*

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
    const multiply_typeof_helper< boost::units::unit< Dim, System >, boost::lambda::lambda_functor< Arg > >::type
    operator*(const boost::units::unit< Dim, System > & a,
              const boost::lambda::lambda_functor< Arg > & b);
```

## Description

Disambiguating overload for action `unit<Dim, System> * lambda_functor<Arg>` based on `<boost/lambda/detail/operators.hpp>`.

## Function template operator/

boost::units::operator/

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
    const divide_typeof_helper< boost::units::unit< Dim, System >, boost::lambda::lambda_functor< Arg > >::type
    operator/(const boost::units::unit< Dim, System > & a,
              const boost::lambda::lambda_functor< Arg > & b);
```

## Description

Disambiguating overload for action unit<Dim, System> / lambda\_functor<Arg> based on <boost/lambda/detail/operators.hpp>.

## Function template operator\*

boost::units::operator\*

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
    const multiply_typeof_helper< boost::lambda::lambda_functor< Arg >, boost::units::unit< Dim, System > >::type
    operator*(const boost::lambda::lambda_functor< Arg > & a,
              const boost::units::unit< Dim, System > & b);
```

## Description

Disambiguating overload for action `lambda_functor<Arg> * unit<Dim, System>` based on `<boost/lambda/detail/operators.hpp>`.

## Function template operator/

boost::units::operator/

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
    const divide_typeof_helper< boost::lambda::lambda_functor< Arg >, boost::units::unit< Dim, System > >::type
    operator/(const boost::lambda::lambda_functor< Arg > & a,
              const boost::units::unit< Dim, System > & b);
```

## Description

Disambiguating overload for action `lambda_functor<Arg> / unit<Dim, System>` based on `<boost/lambda/detail/operators.hpp>`.



## Function template operator\*

boost::units::operator\*

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
    const multiply_typeof_helper< boost::lambda::lambda_functor< Arg >, boost::units::absolute< boost::units::unit< Dim, System > > >::type
    operator*(const boost::lambda::lambda_functor< Arg > & a,
              const boost::units::absolute< boost::units::unit< Dim, System > > & b);
```

## Description

Disambiguating overload for action `lambda_functor<Arg> * absolute<unit<Dim, System> >` based on `<boost/lambda/detail/operators.hpp>`.

## Function template operator\*

boost::units::operator\*

## Synopsis

```
// In header: <boost/units/lambda.hpp>

template<typename System, typename Dim, typename Arg>
    const multiply_typeof_helper< boost::units::absolute< boost::units::unit< Dim, System > >, boost::lambda::lambda_functor< Arg > >::type
operator*(const boost::units::absolute< boost::units::unit< Dim, System > > & a,
          const boost::lambda::lambda_functor< Arg > & b);
```

## Description

Disambiguating overload for action `absolute<unit<Dim, System> > * lambda_functor<Arg>` based on `<boost/lambda/detail/operators.hpp>`.

## Header `<boost/units/limits.hpp>`

specialize `std::numeric_limits` for units.

```
namespace std {
    template<typename Unit, typename T>
        class numeric_limits<::boost::units::quantity< Unit, T >>;
}
```

## Class template `numeric_limits<::boost::units::quantity< Unit, T >>`

`std::numeric_limits<::boost::units::quantity< Unit, T >>`

## Synopsis

```
// In header: <boost/units/limits.hpp>

template<typename Unit, typename T>
class numeric_limits<::boost::units::quantity< Unit, T >> {
public:
    // types
    typedef ::boost::units::quantity< Unit, T > quantity_type;

    // public static functions
    static quantity_type() min();
    static quantity_type() max();
    static quantity_type epsilon();
    static quantity_type round_error();
    static quantity_type infinity();
    static quantity_type quiet_NaN();
    static quantity_type signaling_NaN();
    static quantity_type denorm_min();
    static const bool is_specialized;
    static const int digits;
    static const int digits10;
    static const bool is_signed;
    static const bool is_integer;
    static const bool is_exact;
    static const int radix;
    static const int min_exponent;
    static const int min_exponent10;
    static const int max_exponent;
    static const int max_exponent10;
    static const bool has_infinity;
    static const bool has_quiet_NaN;
    static const bool has_signaling_NaN;
    static const bool has_denorm_loss;
    static const bool is_iec559;
    static const bool is_bounded;
    static const bool is_modulo;
    static const bool traps;
    static const bool tinyness_before;
    static const float_denorm_style has_denorm;
    static const float_round_style round_style;
};
```

## Description

### `numeric_limits` public static functions

1. `static quantity_type() min();`
2. `static quantity_type() max();`
3. `static quantity_type epsilon();`

4. `static quantity_type round_error();`

5. `static quantity_type infinity();`

6. `static quantity_type quiet_NaN();`

7. `static quantity_type signaling_NaN();`

8. `static quantity_type denorm_min();`

## Header `<boost/units/make_scaled_unit.hpp>`

```
namespace boost {
    namespace units {
        template<typename Unit, typename Scale> struct make_scaled_unit;

        template<typename Dimension, typename UnitList, typename OldScale,
                typename Scale>
            struct make_scaled_unit<unit< Dimension, heterogeneous_system< heterogeneous_system_im_
pl< UnitList, Dimension, OldScale > > >, Scale>;
        template<typename Dimension, typename UnitList, typename OldScale,
                long Base>
            struct make_scaled_unit<unit< Dimension, heterogeneous_system< heterogeneous_system_im_
pl< UnitList, Dimension, OldScale > > >, scale< Base, static_rational< 0 > >>;
    }
}
```

## Struct template make\_scaled\_unit

boost::units::make\_scaled\_unit

## Synopsis

```
// In header: <boost/units/make_scaled_unit.hpp>

template<typename Unit, typename Scale>
struct make_scaled_unit {
    // types
    typedef make_scaled_unit< typename reduce_unit< Unit >::type, Scale >::type type;
};
```

**Struct    template    make\_scaled\_unit<unit<    Dimension,    heterogeneous\_system<  
heterogeneous\_system\_impl< UnitList, Dimension, OldScale > > >, Scale>**

boost::units::make\_scaled\_unit<unit< Dimension, heterogeneous\_system< heterogeneous\_system\_impl< UnitList, Dimension,  
OldScale > > >, Scale>

## Synopsis

```
// In header: <boost/units/make_scaled_unit.hpp>

template<typename Dimension, typename UnitList, typename OldScale,
        typename Scale>
struct make_scaled_unit<unit< Dimension, heterogeneous_system< heterogeneous_system_impl< UnitList,
ist, Dimension, OldScale > > >, Scale> {
    // types
    typedef unit< Dimension, heterogeneous_system< heterogeneous_system_impl< UnitList, Dimen-
sion, typename mpl::times< OldScale, list< scale_list_dim< Scale >, dimension-
less_type > >::type > > > type;
};
```

```
Struct    template    make_scaled_unit<unit<    Dimension,    heterogeneous_system<  
heterogeneous_system_impl< UnitList, Dimension, OldScale > > >, scale< Base, static_rational<  
0 > >>
```

```
boost::units::make_scaled_unit<unit< Dimension, heterogeneous_system< heterogeneous_system_impl< UnitList, Dimension,  
OldScale > > >, scale< Base, static_rational< 0 > >>
```

## Synopsis

```
// In header: <boost/units/make_scaled_unit.hpp>

template<typename Dimension, typename UnitList, typename OldScale, long Base>
struct make_scaled_unit<unit< Dimension, heterogeneous_system< heterogeneous_system_impl< UnitList, Dimension, OldScale > > >, scale< Base, static_rational< 0 > >> {
    // types
    typedef unit< Dimension, heterogeneous_system< heterogeneous_system_impl< UnitList, Dimension, OldScale > > > type;
};
```

## Header <boost/units/make\_system.hpp>

Metafunction returning a homogeneous system that can represent any combination of the base units.

Metafunction `make_system` returning a homogeneous system that can represent any combination of the base units. There must be no way to represent any of the base units in terms of the others. `make_system<foot_base_unit, meter_base_unit>::type` is not allowed, for example.

```
namespace boost {
    namespace units {
        template<typename BaseUnit0, typename BaseUnit1, typename BaseUnit2, ... ,
                typename BaseUnitN>
            struct make_system;
    }
}
```

## Struct template make\_system

boost::units::make\_system

## Synopsis

```
// In header: <boost/units/make_system.hpp>

template<typename BaseUnit0, typename BaseUnit1, typename BaseUnit2, ... ,
        typename BaseUnitN>
struct make_system {
    // types
    typedef unspecified type;
};
```

### Description

Metafunction returning a homogeneous system that can represent any combination of the base units. There must be no way to represent any of the base units in terms of the others. `make_system<foot_base_unit, meter_base_unit>::type` is not allowed, for example.

### Header <boost/units/operators.hpp>

Compile time operators and typeof helper classes.

These operators declare the compile-time operators needed to support dimensional analysis algebra. They require the use of Boost.Typeof, emulation or native. Typeof helper classes define result type for heterogeneous operators on value types. These must be defined through specialization for powers and roots.

```
namespace boost {
    namespace units {
        template<typename X> struct unary_plus_typeof_helper;
        template<typename X> struct unary_minus_typeof_helper;
        template<typename X, typename Y> struct add_typeof_helper;
        template<typename X, typename Y> struct subtract_typeof_helper;
        template<typename X, typename Y> struct multiply_typeof_helper;
        template<typename X, typename Y> struct divide_typeof_helper;
        template<typename BaseType, typename Exponent> struct power_typeof_helper;
        template<typename Radicand, typename Index> struct root_typeof_helper;
    }
}
```



## Struct template unary\_plus\_typeof\_helper

boost::units::unary\_plus\_typeof\_helper

## Synopsis

```
// In header: <boost/units/operators.hpp>

template<typename X>
struct unary_plus_typeof_helper {

    // public member functions
    typedef typeof((+typeof_::make< X >())) ;
};
```

## Description

**unary\_plus\_typeof\_helper public member functions**

1. `typedef typeof((+typeof_::make< X >())) ;`

## Struct template unary\_minus\_typeof\_helper

boost::units::unary\_minus\_typeof\_helper

## Synopsis

```
// In header: <boost/units/operators.hpp>

template<typename X>
struct unary_minus_typeof_helper {

    // public member functions
    typedef typeof((-typeof_::make< X >())) ;
};
```

## Description

**unary\_minus\_typeof\_helper public member functions**

1. `typedef typeof((-typeof_::make< X >())) ;`

## Struct template `add_typeof_helper`

`boost::units::add_typeof_helper`

## Synopsis

```
// In header: <boost/units/operators.hpp>

template<typename X, typename Y>
struct add_typeof_helper {

    // public member functions
    typedef typeof((typeof_::make< X >()+typeof_::make< Y >())) ;
};
```

## Description

`add_typeof_helper` **public member functions**

1. `typedef` `typeof((typeof_::make< X >()+typeof_::make< Y >())) ;`

## Struct template `subtract_typeof_helper`

`boost::units::subtract_typeof_helper`

## Synopsis

```
// In header: <boost/units/operators.hpp>

template<typename X, typename Y>
struct subtract_typeof_helper {

    // public member functions
    typedef typeof((typeof_::make< X >()-typeof_::make< Y >())) ;
};
```

## Description

`subtract_typeof_helper` public member functions

1. `typedef` `typeof((typeof_::make< X >()-typeof_::make< Y >())) ;`

## Struct template multiply\_typeof\_helper

boost::units::multiply\_typeof\_helper

## Synopsis

```
// In header: <boost/units/operators.hpp>

template<typename X, typename Y>
struct multiply_typeof_helper {

    // public member functions
    typedef typeof((typeof_::make< X >()*typeof_::make< Y >())) ;
};
```

## Description

**multiply\_typeof\_helper public member functions**

1. `typedef` `typeof((typeof_::make< X >()*typeof_::make< Y >())) ;`

## Struct template `divide_typeof_helper`

`boost::units::divide_typeof_helper`

## Synopsis

```
// In header: <boost/units/operators.hpp>

template<typename X, typename Y>
struct divide_typeof_helper {

    // public member functions
    typedef typeof((typeof_::make< X >()/typeof_::make< Y >())) ;
};
```

## Description

`divide_typeof_helper` public member functions

1. `typedef` `typeof((typeof_::make< X >()/typeof_::make< Y >())) ;`

## Struct template `power_typeof_helper`

`boost::units::power_typeof_helper`

## Synopsis

```
// In header: <boost/units/operators.hpp>

template<typename BaseType, typename Exponent>
struct power_typeof_helper {
    // types
    typedef unspecified type; // specifies the result type

    // public static functions
    static type value(const BaseType &);
};
```

### Description

A helper used by `pow` to raise a runtime object to a compile time known exponent. This template is intended to be specialized. All specializations must conform to the interface shown here. `Exponent` will be either the exponent passed to `pow` or `static_rational<N>` for and integer argument, `N`.

#### `power_typeof_helper` public static functions

1. 

```
static type value(const BaseType & base);
```

Carries out the runtime calculation.

## Struct template `root_typeof_helper`

`boost::units::root_typeof_helper`

## Synopsis

```
// In header: <boost/units/operators.hpp>

template<typename Radicand, typename Index>
struct root_typeof_helper {
    // types
    typedef unspecified type; // specifies the result type

    // public static functions
    static type value(const Radicand &);
};
```

### Description

A helper used by `root` to take a root of a runtime object using a compile time known index. This template is intended to be specialized. All specializations must conform to the interface shown here. `Index` will be either the type passed to `pow` or `static_rational<N>` for and integer argument, `N`.

#### `root_typeof_helper` public static functions

1. 

```
static type value(const Radicand & base);
```

Carries out the runtime calculation.

## Header `<boost/units/pow.hpp>`

Raise values to exponents known at compile-time.

```
namespace boost {
    namespace units {
        template<typename Rat, typename Y>
        power_typeof_helper< Y, Rat >::type pow(const Y &);
        template<typename Rat, typename Y>
        root_typeof_helper< Y, Rat >::type root(const Y &);
    }
}
```



## Function template pow

boost::units::pow — raise a value to a static\_rational power.

## Synopsis

```
// In header: <boost/units/pow.hpp>

template<typename Rat, typename Y>
    power_typeof_helper< Y, Rat >::type pow(const Y & x);
```

## Description

raise a value to an integer power.

## Function template root

boost::units::root — take the static\_rational root of a value.

## Synopsis

```
// In header: <boost/units/pow.hpp>

template<typename Rat, typename Y>
    root_typeof_helper< Y, Rat >::type root(const Y & x);
```

## Description

take the integer root of a value.

## Header <boost/units/quantity.hpp>

```
namespace boost {
    namespace units {
        template<typename Unit, typename Y = double> class quantity;

        template<typename System, typename Y>
            class quantity<BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y>;

        template<typename Dim1, typename System1, typename Dim2, typename System2,
                 typename X, typename Y>
            struct add_typeof_helper<quantity< unit< Dim1, System1 >, X >, quantity< unit< Dim2, System2 >, Y >>;
        template<typename Dim, typename System, typename X, typename Y>
            struct add_typeof_helper<quantity< unit< Dim, System >, X >, quantity< unit< Dim, System >, Y >>;
        template<typename Dim1, typename System1, typename Dim2, typename System2,
                 typename X, typename Y>
            struct subtract_typeof_helper<quantity< unit< Dim1, System1 >, X >, quantity< unit< Dim2, System2 >, Y >>;
        template<typename Dim, typename System, typename X, typename Y>
            struct subtract_typeof_helper<quantity< unit< Dim, System >, X >, quantity< unit< Dim, System >, Y >>;

        // quantity_cast provides mutating access to underlying quantity value_type
        template<typename X, typename Y> X quantity_cast(Y & source);
        template<typename X, typename Y> X quantity_cast(const Y & source);

        // swap quantities
        template<typename Unit, typename Y>
            void swap(quantity< Unit, Y > & lhs, quantity< Unit, Y > & rhs);

        // runtime unit divided by scalar
        template<typename System, typename Dim, typename Y>
            divide_typeof_helper< unit< Dim, System >, Y >::type
            operator/(const unit< Dim, System > &, const Y & rhs);

        // runtime scalar times unit
        template<typename System, typename Dim, typename Y>
            multiply_typeof_helper< Y, unit< Dim, System > >::type
            operator*(const Y & lhs, const unit< Dim, System > &);

        // runtime scalar divided by unit
        template<typename System, typename Dim, typename Y>
```

```

    divide_typeof_helper< Y, unit< Dim, System > >::type
    operator/(const Y & lhs, const unit< Dim, System > &);

// runtime quantity times scalar
template<typename Unit, typename X>
    multiply_typeof_helper< quantity< Unit, X >, X >::type
    operator*(const quantity< Unit, X > & lhs, const X & rhs);

// runtime scalar times quantity
template<typename Unit, typename X>
    multiply_typeof_helper< X, quantity< Unit, X > >::type
    operator*(const X & lhs, const quantity< Unit, X > & rhs);

// runtime quantity divided by scalar
template<typename Unit, typename X>
    divide_typeof_helper< quantity< Unit, X >, X >::type
    operator/(const quantity< Unit, X > & lhs, const X & rhs);

// runtime scalar divided by quantity
template<typename Unit, typename X>
    divide_typeof_helper< X, quantity< Unit, X > >::type
    operator/(const X & lhs, const quantity< Unit, X > & rhs);

// runtime unit times quantity
template<typename System1, typename Dim1, typename Unit2, typename Y>
    multiply_typeof_helper< unit< Dim1, System1 >, quantity< Unit2, Y > >::type
    operator*(const unit< Dim1, System1 > &,
              const quantity< Unit2, Y > & rhs);

// runtime unit divided by quantity
template<typename System1, typename Dim1, typename Unit2, typename Y>
    divide_typeof_helper< unit< Dim1, System1 >, quantity< Unit2, Y > >::type
    operator/(const unit< Dim1, System1 > &,
              const quantity< Unit2, Y > & rhs);

// runtime quantity times unit
template<typename Unit1, typename System2, typename Dim2, typename Y>
    multiply_typeof_helper< quantity< Unit1, Y >, unit< Dim2, System2 > >::type
    operator*(const quantity< Unit1, Y > & lhs,
              const unit< Dim2, System2 > &);

// runtime quantity divided by unit
template<typename Unit1, typename System2, typename Dim2, typename Y>
    divide_typeof_helper< quantity< Unit1, Y >, unit< Dim2, System2 > >::type
    operator/(const quantity< Unit1, Y > & lhs,
              const unit< Dim2, System2 > &);

// runtime unary plus quantity
template<typename Unit, typename Y>
    unary_plus_typeof_helper< quantity< Unit, Y > >::type
    operator+(const quantity< Unit, Y > & val);

// runtime unary minus quantity
template<typename Unit, typename Y>
    unary_minus_typeof_helper< quantity< Unit, Y > >::type
    operator-(const quantity< Unit, Y > & val);

// runtime quantity plus quantity
template<typename Unit1, typename Unit2, typename X, typename Y>
    add_typeof_helper< quantity< Unit1, X >, quantity< Unit2, Y > >::type
    operator+(const quantity< Unit1, X > & lhs,
              const quantity< Unit2, Y > & rhs);

```

```

// runtime quantity minus quantity
template<typename Unit1, typename Unit2, typename X, typename Y>
    subtract_typeof_helper< quantity< Unit1, X >, quantity< Unit2, Y > >::type
    operator-(const quantity< Unit1, X > & lhs,
              const quantity< Unit2, Y > & rhs);

// runtime quantity times quantity
template<typename Unit1, typename Unit2, typename X, typename Y>
    multiply_typeof_helper< quantity< Unit1, X >, quantity< Unit2, Y > >::type
    operator*(const quantity< Unit1, X > & lhs,
              const quantity< Unit2, Y > & rhs);

// runtime quantity divided by quantity
template<typename Unit1, typename Unit2, typename X, typename Y>
    divide_typeof_helper< quantity< Unit1, X >, quantity< Unit2, Y > >::type
    operator/(const quantity< Unit1, X > & lhs,
              const quantity< Unit2, Y > & rhs);

// runtime operator==
template<typename Unit, typename X, typename Y>
    bool operator==(const quantity< Unit, X > & val1,
                    const quantity< Unit, Y > & val2);

// runtime operator!=
template<typename Unit, typename X, typename Y>
    bool operator!=(const quantity< Unit, X > & val1,
                    const quantity< Unit, Y > & val2);

// runtime operator<
template<typename Unit, typename X, typename Y>
    bool operator<(const quantity< Unit, X > & val1,
                  const quantity< Unit, Y > & val2);

// runtime operator<=
template<typename Unit, typename X, typename Y>
    bool operator<=(const quantity< Unit, X > & val1,
                    const quantity< Unit, Y > & val2);

// runtime operator>
template<typename Unit, typename X, typename Y>
    bool operator>(const quantity< Unit, X > & val1,
                  const quantity< Unit, Y > & val2);

// runtime operator>=
template<typename Unit, typename X, typename Y>
    bool operator>=(const quantity< Unit, X > & val1,
                    const quantity< Unit, Y > & val2);
}

```

## Class template quantity

boost::units::quantity — class declaration

## Synopsis

```
// In header: <boost/units/quantity.hpp>

template<typename Unit, typename Y = double>
class quantity {
public:
    // types
    typedef quantity< Unit, Y > this_type;
    typedef Y value_type;
    typedef Unit unit_type;

    // construct/copy/destruct
    quantity();
    quantity(unspecified_null_pointer_constant_type);
    quantity(const this_type &);
    template<typename YY>
        quantity(const quantity< Unit, YY > &, unspecified = 0);
    template<typename YY>
        quantity(const quantity< Unit, YY > &, unspecified = 0);
    template<typename Unit2, typename YY>
        quantity(const quantity< Unit2, YY > &, unspecified = 0);
    template<typename Unit2, typename YY>
        quantity(const quantity< Unit2, YY > &, unspecified = 0);
    quantity(const value_type &, int);
    quantity& operator=(const this_type &);
    template<typename YY> quantity& operator=(const quantity< Unit, YY > &);
    template<typename Unit2, typename YY>
        quantity& operator=(const quantity< Unit2, YY > &);

    // private member functions
    BOOST_MPL_ASSERT_NOT(unspecified);

    // public member functions
    const value_type & value() const;
    template<typename Unit2, typename YY>
        this_type & operator+=(const quantity< Unit2, YY > &);
    template<typename Unit2, typename YY>
        this_type & operator-=(const quantity< Unit2, YY > &);
    template<typename Unit2, typename YY>
        this_type & operator*=(const quantity< Unit2, YY > &);
    template<typename Unit2, typename YY>
        this_type & operator/=(const quantity< Unit2, YY > &);
    this_type & operator*=(const value_type &);
    this_type & operator/=(const value_type &);

    // public static functions
    static this_type from_value(const value_type &);
};
```

## Description

### quantity public construct/copy/destruct

1. `quantity();`

2. `quantity(unspecified_null_pointer_constant_type);`

3. `quantity(const this_type & source);`

4. `template<typename YY>  
quantity(const quantity< Unit, YY > & source, unspecified = 0);`

implicit conversion between value types is allowed if allowed for value types themselves

5. `template<typename YY>  
quantity(const quantity< Unit, YY > & source, unspecified = 0);`

implicit conversion between value types is not allowed if not allowed for value types themselves

6. `template<typename Unit2, typename YY>  
quantity(const quantity< Unit2, YY > & source, unspecified = 0);`

explicit conversion between different unit systems is allowed if implicit conversion is disallowed

7. `template<typename Unit2, typename YY>  
quantity(const quantity< Unit2, YY > & source, unspecified = 0);`

implicit conversion between different unit systems is allowed if each fundamental dimension is implicitly convertible

8. `quantity(const value_type & val, int);`

9. `quantity& operator=(const this_type & source);`

10. `template<typename YY> quantity& operator=(const quantity< Unit, YY > & source);`

implicit assignment between value types is allowed if allowed for value types themselves

11. `template<typename Unit2, typename YY>  
quantity& operator=(const quantity< Unit2, YY > & source);`

implicit assignment between different unit systems is allowed if each fundamental dimension is implicitly convertible

#### **quantity private member functions**

1. `BOOST_MPL_ASSERT_NOT(unspecified);`

#### **quantity public member functions**

1. `const value_type & value() const;`

constant accessor to value

can add a quantity of the same type if `add_typeof_helper<value_type,value_type>::type` is convertible to `value_type`

```
2. template<typename Unit2, typename YY>
    this_type & operator+=(const quantity< Unit2, YY > & source);
```

can subtract a quantity of the same type if `subtract_typeof_helper<value_type,value_type>::type` is convertible to `value_type`

```
3. template<typename Unit2, typename YY>
    this_type & operator-=(const quantity< Unit2, YY > & source);
```

```
4. template<typename Unit2, typename YY>
    this_type & operator*=(const quantity< Unit2, YY > & source);
```

```
5. template<typename Unit2, typename YY>
    this_type & operator/=(const quantity< Unit2, YY > & source);
```

can multiply a quantity by a scalar `value_type` if `multiply_typeof_helper<value_type,value_type>::type` is convertible to `value_type`

```
6. this_type & operator*=(const value_type & source);
```

can divide a quantity by a scalar `value_type` if `divide_typeof_helper<value_type,value_type>::type` is convertible to `value_type`

```
7. this_type & operator/=(const value_type & source);
```

#### **quantity public static functions**

```
1. static this_type from_value(const value_type & val);
```

Construct quantity directly from `value_type` (potentially dangerous).

#### **Specializations**

- Class template `quantity<BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y>`

**Class template quantity<BOOST\_UNITS\_DIMENSIONLESS\_UNIT(System), Y>**

boost::units::quantity&lt;BOOST\_UNITS\_DIMENSIONLESS\_UNIT(System), Y&gt;

**Synopsis**

```
// In header: <boost/units/quantity.hpp>

template<typename System, typename Y>
class quantity<BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y> {
public:
    // types
    typedef quantity< unit< dimensionless_type, System >, Y > this_type;
    typedef Y value_type;
    typedef System system_type;
    typedef dimensionless_type dimension_type;
    typedef unit< dimension_type, system_type > unit_type;

    // construct/copy/destruct
    quantity& operator=(const this_type &);
    template<typename YY>
        quantity& operator=(const quantity< unit< dimension_type, system_type >, YY > &);
    template<typename System2>
        quantity& operator=(const quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System2), Y > &);

    // public member functions
    quantity();
    quantity(value_type);
    quantity(const this_type &);
    template<typename YY>
        quantity(const quantity< unit< dimension_type, system_type >, YY > &,
            unspecified = 0);
    template<typename YY>
        quantity(const quantity< unit< dimension_type, system_type >, YY > &,
            unspecified = 0);
    template<typename System2, typename Y2>
        quantity(const quantity< unit< dimensionless_type, System2 >, Y2 > &,
            unspecified = 0, unspecified = 0, unspecified = 0);
    template<typename System2, typename Y2>
        quantity(const quantity< unit< dimensionless_type, System2 >, Y2 > &,
            unspecified = 0, unspecified = 0, unspecified = 0);
    template<typename System2, typename Y2>
        quantity(const quantity< unit< dimensionless_type, System2 >, Y2 > &,
            unspecified = 0);
    operator value_type() const;
    const value_type & value() const;
    this_type & operator+=(const this_type &);
    this_type & operator-=(const this_type &);
    this_type & operator*=(const value_type &);
    this_type & operator/=(const value_type &);

    // public static functions
    static this_type from_value(const value_type &);
};
```

**Description**

Specialization for dimensionless quantities. Implicit conversions between unit systems are allowed because all dimensionless quantities are equivalent. Implicit construction and assignment from and conversion to `value_type` is also allowed.



**quantity public construct/copy/destruct**

1. 

```
quantity& operator=(const this_type & source);
```
2. 

```
template<typename YY>
    quantity& operator=(const quantity< unit< dimension_type, system_type >, YY > & source);
```

implicit assignment between value types is allowed if allowed for value types themselves

3. 

```
template<typename System2>
    quantity& operator=(const quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System2), Y > & source);
```

implicit assignment between different unit systems is allowed

**quantity public member functions**

1. 

```
quantity();
```
2. 

```
quantity(value_type val);
```
3. 

```
quantity(const this_type & source);
```
4. 

```
template<typename YY>
    quantity(const quantity< unit< dimension_type, system_type >, YY > & source,
              unspecified = 0);
```

implicit conversion between value types is allowed if allowed for value types themselves

5. 

```
template<typename YY>
    quantity(const quantity< unit< dimension_type, system_type >, YY > & source,
              unspecified = 0);
```

implicit conversion between value types is not allowed if not allowed for value types themselves

6. 

```
template<typename System2, typename Y2>
    quantity(const quantity< unit< dimensionless_type, System2 >, Y2 > & source,
              unspecified = 0, unspecified = 0, unspecified = 0);
```

implicit conversion between different unit systems is allowed

7. 

```
template<typename System2, typename Y2>
    quantity(const quantity< unit< dimensionless_type, System2 >, Y2 > & source,
              unspecified = 0, unspecified = 0, unspecified = 0);
```

implicit conversion between different unit systems is allowed

8. 

```
template<typename System2, typename Y2>
    quantity(const quantity< unit< dimensionless_type, System2 >, Y2 > & source,
              unspecified = 0);
```

conversion between different unit systems is explicit when the units are not equivalent.

9. 

```
operator value_type() const;
```

implicit conversion to value\_type is allowed

10. 

```
const value_type & value() const;
```

constant accessor to value

can add a quantity of the same type if add\_typeof\_helper<value\_type,value\_type>::type is convertible to value\_type

11. 

```
this_type & operator+=(const this_type & source);
```

can subtract a quantity of the same type if subtract\_typeof\_helper<value\_type,value\_type>::type is convertible to value\_type

12. 

```
this_type & operator-=(const this_type & source);
```

can multiply a quantity by a scalar value\_type if multiply\_typeof\_helper<value\_type,value\_type>::type is convertible to value\_type

13. 

```
this_type & operator*=(const value_type & val);
```

can divide a quantity by a scalar value\_type if divide\_typeof\_helper<value\_type,value\_type>::type is convertible to value\_type

14. 

```
this_type & operator/=(const value_type & val);
```

#### **quantity public static functions**

1. 

```
static this_type from_value(const value_type & val);
```

Construct quantity directly from value\_type.

**Struct template `add_typeof_helper<quantity< unit< Dim1, System1 >, X >, quantity< unit< Dim2, System2 >, Y >>`**

`boost::units::add_typeof_helper<quantity< unit< Dim1, System1 >, X >, quantity< unit< Dim2, System2 >, Y >>`

## Synopsis

```
// In header: <boost/units/quantity.hpp>

template<typename Dim1, typename System1, typename Dim2, typename System2,
        typename X, typename Y>
struct add_typeof_helper<quantity< unit< Dim1, System1 >, X >, quantity< unit< Dim2, System2 >, Y >> {
};
```

## Description

for sun CC we need to invoke SFINAE at the top level, otherwise it will silently return int.

**Struct template `add_typeof_helper<quantity< unit< Dim, System >, X >, quantity< unit< Dim, System >, Y >>`**

`boost::units::add_typeof_helper<quantity< unit< Dim, System >, X >, quantity< unit< Dim, System >, Y >>`

## Synopsis

```
// In header: <boost/units/quantity.hpp>

template<typename Dim, typename System, typename X, typename Y>
struct add_typeof_helper<quantity< unit< Dim, System >, X >, quantity< unit< Dim, System >, Y >> {
    // types
    typedef add_typeof_helper< X, Y >::type    value_type;
    typedef unit< Dim, System >                unit_type;
    typedef quantity< unit_type, value_type > type;
};
```

**Struct template `subtract_typeof_helper<quantity< unit< Dim1, System1 >, X >, quantity< unit< Dim2, System2 >, Y >>`**

`boost::units::subtract_typeof_helper<quantity< unit< Dim1, System1 >, X >, quantity< unit< Dim2, System2 >, Y >>`

## Synopsis

```
// In header: <boost/units/quantity.hpp>

template<typename Dim1, typename System1, typename Dim2, typename System2,
        typename X, typename Y>
struct subtract_typeof_helper<quantity< unit< Dim1, System1 >, X >, quantity< unit< Dim2, System2 >, Y >> {
};
```

## Struct template `subtract_typeof_helper<quantity< unit< Dim, System >, X >, quantity< unit< Dim, System >, Y >>`

`boost::units::subtract_typeof_helper<quantity< unit< Dim, System >, X >, quantity< unit< Dim, System >, Y >>`

## Synopsis

```
// In header: <boost/units/quantity.hpp>

template<typename Dim, typename System, typename X, typename Y>
struct subtract_typeof_helper<quantity< unit< Dim, System >, X >, quantity< unit< Dim, System >, Y >> {
    // types
    typedef subtract_typeof_helper< X, Y >::type value_type;
    typedef unit< Dim, System > unit_type;
    typedef quantity< unit_type, value_type > type;
};
```

## Header <boost/units/reduce\_unit.hpp>

Returns a unique type for every unit.

```
namespace boost {
    namespace units {
        template<typename Unit> struct reduce_unit;
    }
}
```

## Struct template `reduce_unit`

`boost::units::reduce_unit` — Returns a unique type for every unit.

## Synopsis

```
// In header: <boost/units/reduce_unit.hpp>

template<typename Unit>
struct reduce_unit {
    // types
    typedef unspecified type;
};
```

## Header `<boost/units/scale.hpp>`

10<sup>3</sup> Engineering & 2<sup>10</sup> binary scaling factors for autoprefixing.

```
namespace boost {
    namespace units {
        template<long Base, typename Exponent> struct scale;
        template<long Base, typename Exponent>
            std::string symbol_string(const scale< Base, Exponent > &);
        template<long Base, typename Exponent>
            std::string name_string(const scale< Base, Exponent > &);
    }
}
```

## Struct template scale

boost::units::scale

## Synopsis

```
// In header: <boost/units/scale.hpp>

template<long Base, typename Exponent>
struct scale {
    // types
    typedef Exponent exponent;
    typedef double    value_type;

    // public static functions
    static value_type value();
    static const long base;
};
```

## Description

class representing a scaling factor such as  $10^3$  The exponent must be a static rational.

### scale public static functions

1. 

```
static value_type value();
```

## Header <boost/units/scaled\_base\_unit.hpp>

```
namespace boost {
    namespace units {
        template<typename S, typename Scale> struct scaled_base_unit;
    }
}
```



## Struct template `scaled_base_unit`

`boost::units::scaled_base_unit`

## Synopsis

```
// In header: <boost/units/scaled_base_unit.hpp>

template<typename S, typename Scale>
struct scaled_base_unit {
    // types
    typedef scaled_base_unit      type;
    typedef scaled_base_unit_tag  tag;
    typedef S                    system_type;
    typedef Scale                scale_type;
    typedef S::dimension_type    dimension_type;
    typedef unspecified          unit_type;

    // public static functions
    static std::string symbol();
    static std::string name();
};
```

## Description

`scaled_base_unit` public static functions

1. `static std::string symbol();`
2. `static std::string name();`

## Header `<boost/units/static_constant.hpp>`

```
BOOST_UNITS_STATIC_CONSTANT(name, type)
```

## Macro BOOST\_UNITS\_STATIC\_CONSTANT

BOOST\_UNITS\_STATIC\_CONSTANT

## Synopsis

```
// In header: <boost/units/static_constant.hpp>

BOOST_UNITS_STATIC_CONSTANT(name, type)
```

### Description

A convenience macro that allows definition of static constants in headers in an ODR-safe way.

## Header <boost/units/static\_rational.hpp>

Compile-time rational numbers and operators.

```
namespace boost {
  namespace units {
    template<integer_type Value> struct static_abs;

    template<integer_type N, integer_type D = 1> class static_rational;

    typedef long integer_type;

    // get decimal value of static_rational
    template<typename T, integer_type N, integer_type D>
      divide_typeof_helper< T, T >::type
      value(const static_rational< N, D > &);
  }
}
```

## Struct template static\_abs

boost::units::static\_abs — Compile time absolute value.

## Synopsis

```
// In header: <boost/units/static_rational.hpp>

template<integer_type Value>
struct static_abs {

    // public member functions
    BOOST_STATIC_CONSTANT(integer_type);

};
```

## Description

### static\_abs public member functions

1. BOOST\_STATIC\_CONSTANT(integer\_type);

## Class template static\_rational

boost::units::static\_rational

## Synopsis

```
// In header: <boost/units/static_rational.hpp>

template<integer_type N, integer_type D = 1>
class static_rational {
public:
    // types
    typedef unspecified tag;
    typedef static_rational< Numerator, Denominator > type; // static_rational<N,D> reduced by GCD

    // construct/copy/destroy
    static_rational();

    // public static functions
    static integer_type numerator();
    static integer_type denominator();
    static const integer_type Numerator;
    static const integer_type Denominator;
};
```

### Description

This is an implementation of a compile time rational number, where `static_rational<N,D>` represents a rational number with numerator `N` and denominator `D`. Because of the potential for ambiguity arising from multiple equivalent values of `static_rational` (e.g. `static_rational<6,2>==static_rational<3>`), static rationals should always be accessed through `static_rational<N,D>::type`. Template specialization prevents instantiation of zero denominators (i.e. `static_rational<N,0>`). The following compile-time arithmetic operators are provided for `static_rational` variables only (no operators are defined between long and `static_rational`):

- `mpl::negate`
- `mpl::plus`
- `mpl::minus`
- `mpl::times`
- `mpl::divides`

Neither `static_power` nor `static_root` are defined for `static_rational`. This is because template types may not be floating point values, while powers and roots of rational numbers can produce floating point values.

### static\_rational public construct/copy/destroy

1. `static_rational();`

### static\_rational public static functions

1. `static integer_type numerator();`
2. `static integer_type denominator();`

Header **<boost/units/unit.hpp>**

```

namespace boost {
  namespace units {
    template<typename Dim, typename System, typename Enable> class unit;

    template<typename Dim, typename System>
      struct reduce_unit<unit< Dim, System >>;
    template<typename Dim, typename System, long N, long D>
      struct power_typeof_helper<unit< Dim, System >, static_rational< N, D >>;
    template<typename Dim, typename System, long N, long D>
      struct root_typeof_helper<unit< Dim, System >, static_rational< N, D >>;

    // unit runtime unary plus
    template<typename Dim, typename System>
      unary_plus_typeof_helper< unit< Dim, System > >::type
      operator+(const unit< Dim, System > &);

    // unit runtime unary minus
    template<typename Dim, typename System>
      unary_minus_typeof_helper< unit< Dim, System > >::type
      operator-(const unit< Dim, System > &);

    // runtime add two units
    template<typename Dim1, typename Dim2, typename System1, typename System2>
      add_typeof_helper< unit< Dim1, System1 >, unit< Dim2, System2 > >::type
      operator+(const unit< Dim1, System1 > &, const unit< Dim2, System2 > &);

    // runtime subtract two units
    template<typename Dim1, typename Dim2, typename System1, typename System2>
      subtract_typeof_helper< unit< Dim1, System1 >, unit< Dim2, System2 > >::type
      operator-(const unit< Dim1, System1 > &, const unit< Dim2, System2 > &);

    // runtime multiply two units
    template<typename Dim1, typename Dim2, typename System1, typename System2>
      multiply_typeof_helper< unit< Dim1, System1 >, unit< Dim2, System2 > >::type
      operator*(const unit< Dim1, System1 > &, const unit< Dim2, System2 > &);

    // runtime divide two units
    template<typename Dim1, typename Dim2, typename System1, typename System2>
      divide_typeof_helper< unit< Dim1, System1 >, unit< Dim2, System2 > >::type
      operator/(const unit< Dim1, System1 > &, const unit< Dim2, System2 > &);

    // unit runtime operator==
    template<typename Dim1, typename Dim2, typename System1, typename System2>
      bool operator==(const unit< Dim1, System1 > &,
                     const unit< Dim2, System2 > &);

    // unit runtime operator!=
    template<typename Dim1, typename Dim2, typename System1, typename System2>
      bool operator!=(const unit< Dim1, System1 > &,
                     const unit< Dim2, System2 > &);
  }
}

```

## Class template unit

boost::units::unit — class representing a model-dependent unit with no associated value

## Synopsis

```
// In header: <boost/units/unit.hpp>

template<typename Dim, typename System, typename Enable>
class unit {
public:
    // types
    typedef unit< Dim, System > unit_type;
    typedef unit< Dim, System > this_type;
    typedef Dim                dimension_type;
    typedef System              system_type;

    // construct/copy/destroy
    unit();
    unit(const this_type &);
    unit& operator=(const this_type &);
};
```

### Description

(e.g. meters, Kelvin, feet, etc...)

#### unit public construct/copy/destroy

1. `unit();`
2. `unit(const this_type &);`
3. `unit& operator=(const this_type &);`

## Struct template `reduce_unit<unit< Dim, System >>`

`boost::units::reduce_unit<unit< Dim, System >>` — Returns a unique type for every unit.

## Synopsis

```
// In header: <boost/units/unit.hpp>

template<typename Dim, typename System>
struct reduce_unit<unit< Dim, System >> {
    // types
    typedef unspecified type;
};
```

## Struct template `power_typeof_helper<unit< Dim, System >, static_rational< N, D >>`

`boost::units::power_typeof_helper<unit< Dim, System >, static_rational< N, D >>` — raise unit to a `static_rational` power

## Synopsis

```
// In header: <boost/units/unit.hpp>

template<typename Dim, typename System, long N, long D>
struct power_typeof_helper<unit< Dim, System >, static_rational< N, D >> {
    // types
    typedef unit< typename static_power< Dim, static_rational< N, D >>::type, typename static_rational< N, D >>::type > type;

    // public static functions
    static type value(const unit< Dim, System > &);
};
```

## Description

`power_typeof_helper` public static functions

1. `static type value(const unit< Dim, System > &);`



## Struct template `root_typeof_helper<unit< Dim, System >, static_rational< N, D >>`

`boost::units::root_typeof_helper<unit< Dim, System >, static_rational< N, D >>` — take the `static_rational` root of a unit

## Synopsis

```
// In header: <boost/units/unit.hpp>

template<typename Dim, typename System, long N, long D>
struct root_typeof_helper<unit< Dim, System >, static_rational< N, D >> {
    // types
    typedef unit< typename static_root< Dim, static_rational< N, D > >::type, typename static_rational< N, D > >::type> type;

    // public static functions
    static type value(const unit< Dim, System > &);
};
```

## Description

`root_typeof_helper` public static functions

1. `static type value(const unit< Dim, System > &);`

## Header `<boost/units/units_fwd.hpp>`

Forward declarations of library components.

Forward declarations of units library - dimensions, systems, quantity and string components.

## Dimensions Reference

### Header `<boost/units/physical_dimensions.hpp>`

Physical dimensions according to the SI system.

This header includes all physical dimension headers for both base and derived dimensions.

### Header `<boost/units/physical_dimensions/absorbed_dose.hpp>`

```
namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 2, time_base_dimension, -2 >::type absorbed_dose_dimension; // derived dimension for absorbed dose : L^2 T^-2
    }
}
```

### Header `<boost/units/physical_dimensions/acceleration.hpp>`

```
namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 1, time_base_dimension, -2 >::type acceleration_dimension; // derived dimension for acceleration : L T^-2
    }
}
```

**Header <boost/units/physical\_dimensions/action.hpp>**

```
namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -1 >::type action_dimension; // derived dimension for action : L^2 M T^-1
    }
}
```

**Header <boost/units/physical\_dimensions/activity.hpp>**

```
namespace boost {
    namespace units {
        typedef derived_dimension< time_base_dimension, -1 >::type activity_dimension; // derived dimension for activity : T^-1
    }
}
```

**Header <boost/units/physical\_dimensions/amount.hpp>**

```
namespace boost {
    namespace units {
        struct amount_base_dimension;

        typedef amount_base_dimension::dimension_type amount_dimension; // dimension of amount of substance (N)
    }
}
```

## Struct `amount_base_dimension`

`boost::units::amount_base_dimension` — base dimension of amount

## Synopsis

```
// In header: <boost/units/physical_dimensions/amount.hpp>

struct amount_base_dimension {
};
```

## Header `<boost/units/physical_dimensions/angular_acceleration.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< time_base_dimension, -2, plane_angle_base_dimension, 1 >::type angular_acceleration_dimension; // derived dimension for angular acceleration : T^-2 QP
  }
}
```

## Header `<boost/units/physical_dimensions/angular_momentum.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -1, plane_angle_base_dimension, -1 >::type angular_momentum_dimension; // derived dimension for angular momentum : L^2 M T^-1 QP^-1
  }
}
```

## Header `<boost/units/physical_dimensions/angular_velocity.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< time_base_dimension, -1, plane_angle_base_dimension, 1 >::type angular_velocity_dimension; // derived dimension for angular velocity : T^-1 QP
  }
}
```

## Header `<boost/units/physical_dimensions/area.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2 >::type area_dimension; // derived dimension for area : L^2
  }
}
```

**Header <boost/units/physical\_dimensions/capacitance.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension,-2, mass_base_dimension,-1, time_base_dimension, 4, current_base_dimension, 2 >::type capacitance_dimension; // derived dimension for capacitance : L^-2 M^-1 T^4 I^2
    }
}

```

**Header <boost/units/physical\_dimensions/conductance.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension,-2, mass_base_dimension,-1, time_base_dimension, 3, current_base_dimension, 2 >::type conductance_dimension; // derived dimension for conductance : L^-2 M^-1 T^3 I^2
    }
}

```

**Header <boost/units/physical\_dimensions/conductivity.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension,-3, mass_base_dimension,-1, time_base_dimension, 3, current_base_dimension, 2 >::type conductivity_dimension; // derived dimension for conductivity : L^-3 M^-1 T^3 I^2
    }
}

```

**Header <boost/units/physical\_dimensions/current.hpp>**

```

namespace boost {
    namespace units {
        struct current_base_dimension;

        typedef current_base_dimension::dimension_type current_dimension; // dimension of electric current (I)
    }
}

```

## Struct `current_base_dimension`

`boost::units::current_base_dimension` — base dimension of current

## Synopsis

```
// In header: <boost/units/physical_dimensions/current.hpp>

struct current_base_dimension {
};
```

## Header `<boost/units/physical_dimensions/dose_equivalent.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, time_base_dimension,-2 >::type dose_equivalent_dimension; // derived dimension for dose equivalent : L^2 T^-2
  }
}
```

## Header `<boost/units/physical_dimensions/dynamic_viscosity.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< mass_base_dimension, 1, length_base_dimension,-1, time_base_dimension,-1 >::type dynamic_viscosity_dimension; // derived dimension for dynamic viscosity : M L^-1 T^-1
  }
}
```

## Header `<boost/units/physical_dimensions/electric_charge.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< time_base_dimension, 1, current_base_dimension, 1 >::type electric_charge_dimension; // derived dimension for electric charge : T^1 I^1
  }
}
```

## Header `<boost/units/physical_dimensions/electric_potential.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension,-3, current_base_dimension,-1 >::type electric_potential_dimension; // derived dimension for electric potential : L^2 M T^-3 I^-1
  }
}
```

**Header <boost/units/physical\_dimensions/energy.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -2 >::type energy_dimension; // derived dimension for energy : L^2 M T^-2
    }
}

```

**Header <boost/units/physical\_dimensions/energy\_density.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, -1, mass_base_dimension, 1, time_base_dimension, -2 >::type energy_density_dimension; // derived dimension for energy density : L^-1 M^1 T^-2
    }
}

```

**Header <boost/units/physical\_dimensions/force.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 1, mass_base_dimension, 1, time_base_dimension, -2 >::type force_dimension; // derived dimension for force : L M T^-2
    }
}

```

**Header <boost/units/physical\_dimensions/frequency.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< time_base_dimension, -1 >::type frequency_dimension; // derived dimension for frequency : T^-1
    }
}

```

**Header <boost/units/physical\_dimensions/heat\_capacity.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -2, temperature_base_dimension, -1 >::type heat_capacity_dimension; // derived dimension for heat capacity : L^2 M T^-2 Theta^-1
    }
}

```

**Header <boost/units/physical\_dimensions/illuminance.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, -2, luminous_intensity_base_dimension, 1, solid_angle_base_dimension, 1 >::type illuminance_dimension; // derived dimension for illuminance : L^-2 I QS
    }
}

```

**Header <boost/units/physical\_dimensions/impedance.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -3, current_base_dimension, -2 >::type impedance_dimension; // derived dimension for impedance : L^2 M T^-3 I^-2
    }
}

```

**Header <boost/units/physical\_dimensions/inductance.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -2, current_base_dimension, -2 >::type inductance_dimension; // derived dimension for inductance : L^2 M T^-2 I^-2
    }
}

```

**Header <boost/units/physical\_dimensions/kinematic\_viscosity.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 2, time_base_dimension, -1 >::type kinematic_viscosity_dimension; // derived dimension for kinematic viscosity : L^2 T^-1
    }
}

```

**Header <boost/units/physical\_dimensions/length.hpp>**

```

namespace boost {
    namespace units {
        struct length_base_dimension;

        typedef length_base_dimension::dimension_type length_dimension; // dimension of length (L)
    }
}

```

## Struct `length_base_dimension`

`boost::units::length_base_dimension` — base dimension of length

## Synopsis

```
// In header: <boost/units/physical_dimensions/length.hpp>

struct length_base_dimension {
};
```

## Header `<boost/units/physical_dimensions/luminance.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, -2, luminous_intensity_base_dimension, 1 >::type luminance_dimension; // derived dimension for luminance : L^-2 I
  }
}
```

## Header `<boost/units/physical_dimensions/luminous_flux.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< luminous_intensity_base_dimension, 1, solid_angle_base_dimension, 1 >::type luminous_flux_dimension; // derived dimension for luminous flux : I QS
  }
}
```

## Header `<boost/units/physical_dimensions/luminous_intensity.hpp>`

```
namespace boost {
  namespace units {
    struct luminous_intensity_base_dimension;

    typedef luminous_intensity_base_dimension::dimension_type luminous_intensity_dimension; // dimension of luminous intensity (J)
  }
}
```



## Struct `luminous_intensity_base_dimension`

`boost::units::luminous_intensity_base_dimension` — base dimension of luminous intensity

## Synopsis

```
// In header: <boost/units/physical_dimensions/luminous_intensity.hpp>

struct luminous_intensity_base_dimension {
};
```

## Header `<boost/units/physical_dimensions/magnetic_field_intensity.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, -1, current_base_dimension, 1 >::type magnetic_field_intensity_dimension; // derived dimension for magnetic field intensity : L^-1 I
  }
}
```

## Header `<boost/units/physical_dimensions/magnetic_flux.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -2, current_base_dimension, -1 >::type magnetic_flux_dimension; // derived dimension for magnetic flux : L^2 M T^-2 I^-1
  }
}
```

## Header `<boost/units/physical_dimensions/magnetic_flux_density.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< mass_base_dimension, 1, time_base_dimension, -2, current_base_dimension, -1 >::type magnetic_flux_density_dimension; // derived dimension for magnetic flux density : M T^-2 I^-1
  }
}
```

## Header `<boost/units/physical_dimensions/mass.hpp>`

```
namespace boost {
  namespace units {
    struct mass_base_dimension;

    typedef mass_base_dimension::dimension_type mass_dimension; // dimension of mass (M)
  }
}
```

## Struct `mass_base_dimension`

`boost::units::mass_base_dimension` — base dimension of mass

## Synopsis

```
// In header: <boost/units/physical_dimensions/mass.hpp>

struct mass_base_dimension {
};
```

## Header `<boost/units/physical_dimensions/mass_density.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, -3, mass_base_dimension, 1 >::type mass_density_dimension; // derived dimension for mass density : L^-3 M
  }
}
```

## Header `<boost/units/physical_dimensions/molar_energy.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -2, amount_base_dimension, -1 >::type molar_energy_dimension; // derived dimension for molar energy : L^2 M T^-2 N^-1
  }
}
```

## Header `<boost/units/physical_dimensions/molar_heat_capacity.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -2, temperature_base_dimension, -1, amount_base_dimension, -1 >::type molar_heat_capacity_dimension; // derived dimension for molar heat capacity : L^2 M T^-2 Theta^-1 N^-1
  }
}
```

## Header `<boost/units/physical_dimensions/moment_of_inertia.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, plane_angle_base_dimension, -2 >::type moment_of_inertia_dimension; // derived dimension for moment of inertia : L^2 M QP^-2
  }
}
```

**Header <boost/units/physical\_dimensions/momentum.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 1, mass_base_dimension, 1, time_base_dimension, -1 >::type momentum_dimension; // derived dimension for linear momentum : L M T^-1
    }
}

```

**Header <boost/units/physical\_dimensions/permeability.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, 1, mass_base_dimension, 1, time_base_dimension, -2, current_base_dimension, -2 >::type permeability_dimension; // derived dimension for permeability : L M T^-2 I^-2
    }
}

```

**Header <boost/units/physical\_dimensions/permittivity.hpp>**

```

namespace boost {
    namespace units {
        typedef derived_dimension< length_base_dimension, -3, mass_base_dimension, -1, time_base_dimension, 4, current_base_dimension, 2 >::type permittivity_dimension; // derived dimension for permittivity : L^-3 M^-1 T^4 I^2
    }
}

```

**Header <boost/units/physical\_dimensions/plane\_angle.hpp>**

```

namespace boost {
    namespace units {
        struct plane_angle_base_dimension;

        typedef plane_angle_base_dimension::dimension_type plane_angle_dimension; // base dimension of plane angle (QP)
    }
}

```

## Struct plane\_angle\_base\_dimension

boost::units::plane\_angle\_base\_dimension — base dimension of plane angle

## Synopsis

```
// In header: <boost/units/physical_dimensions/plane_angle.hpp>

struct plane_angle_base_dimension {
};
```

## Header <boost/units/physical\_dimensions/power.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -3 >::type power_dimension; // derived dimension for power : L^2 M T^-3
  }
}
```

## Header <boost/units/physical\_dimensions/pressure.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, -1, mass_base_dimension, 1, time_base_dimension, -2 >::type pressure_dimension; // derived dimension for pressure : L^-1 M T^-2
  }
}
```

## Header <boost/units/physical\_dimensions/reluctance.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, -2, mass_base_dimension, -1, time_base_dimension, 2, current_base_dimension, 2 >::type reluctance_dimension; // derived dimension for reluctance : L^-2 M^-1 T^2 I^2
  }
}
```

## Header <boost/units/physical\_dimensions/resistance.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -3, current_base_dimension, -2 >::type resistance_dimension; // derived dimension for resistance : L^2 M T^-3 I^-2
  }
}
```

**Header <boost/units/physical\_dimensions/resistivity.hpp>**

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 3, mass_base_dimension, 1, time_base_dimension, -3, current_base_dimension, -2 >::type resistivity_dimension; // derived dimension for resistivity : L^3 M T^-3 I^-2
  }
}
```

**Header <boost/units/physical\_dimensions/solid\_angle.hpp>**

```
namespace boost {
  namespace units {
    struct solid_angle_base_dimension;

    typedef solid_angle_base_dimension::dimension_type solid_angle_dimension; // base dimension of solid angle (QS)
  }
}
```

## Struct `solid_angle_base_dimension`

`boost::units::solid_angle_base_dimension` — base dimension of solid angle

## Synopsis

```
// In header: <boost/units/physical_dimensions/solid_angle.hpp>

struct solid_angle_base_dimension {
};
```

## Header `<boost/units/physical_dimensions/specific_energy.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, time_base_dimension,-2 >::type specific_energy_dimension; // derived dimension for specific energy : L^2 T^-2
  }
}
```

## Header `<boost/units/physical_dimensions/specific_heat_capacity.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, time_base_dimension,-2, temperature_base_dimension,-1 >::type specific_heat_capacity_dimension; // derived dimension for specific heat capacity : L^2 T^-2 Theta^-1
  }
}
```

## Header `<boost/units/physical_dimensions/specific_volume.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 3, mass_base_dimension,-1 >::type specific_volume_dimension; // derived dimension for specific volume : L^3 M^-1
  }
}
```

## Header `<boost/units/physical_dimensions/stress.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-1, mass_base_dimension, 1, time_base_dimension,-2 >::type stress_dimension; // derived dimension for stress : L^-1 M T^-2
  }
}
```

**Header <boost/units/physical\_dimensions/surface\_density.hpp>**

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, -2, mass_base_dimension, 1 >::type surface_density_dimension; // derived dimension for surface density : L^-2 M
  }
}
```

**Header <boost/units/physical\_dimensions/surface\_tension.hpp>**

```
namespace boost {
  namespace units {
    typedef derived_dimension< mass_base_dimension, 1, time_base_dimension, -2 >::type surface_tension_dimension; // derived dimension for surface tension : M T^-2
  }
}
```

**Header <boost/units/physical\_dimensions/temperature.hpp>**

```
namespace boost {
  namespace units {
    struct temperature_base_dimension;

    typedef temperature_base_dimension::dimension_type temperature_dimension; // dimension of temperature (Theta)
  }
}
```

## Struct `temperature_base_dimension`

`boost::units::temperature_base_dimension` — base dimension of temperature

## Synopsis

```
// In header: <boost/units/physical_dimensions/temperature.hpp>

struct temperature_base_dimension {
};
```

## Header `<boost/units/physical_dimensions/thermal_conductivity.hpp>`

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 1, mass_base_dimension, 1, time_base_dimension, -3, temperature_base_dimension, -1 >::type thermal_conductivity_dimension; // derived dimension for thermal conductivity : L^1 M^1 T^-3 Theta^-1
  }
}
```

## Header `<boost/units/physical_dimensions/time.hpp>`

```
namespace boost {
  namespace units {
    struct time_base_dimension;

    typedef time_base_dimension::dimension_type time_dimension; // dimension of time (T)
  }
}
```



## Struct time\_base\_dimension

boost::units::time\_base\_dimension — base dimension of time

## Synopsis

```
// In header: <boost/units/physical_dimensions/time.hpp>

struct time_base_dimension {
};
```

## Header <boost/units/physical\_dimensions/torque.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_dimension, -2, plane_angle_base_dimension, -1 >::type torque_dimension; // derived dimension for torque : L^2 M T^-2 QP^-1
  }
}
```

## Header <boost/units/physical\_dimensions/velocity.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 1, time_base_dimension, -1 >::type velocity_dimension; // derived dimension for velocity : L T^-1
  }
}
```

## Header <boost/units/physical\_dimensions/volume.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 3 >::type volume_dimension; // derived dimension for volume : L^3
  }
}
```

## Header <boost/units/physical\_dimensions/wavenumber.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, -1 >::type wavenumber_dimension; // derived dimension for wavenumber : L^-1
  }
}
```

## SI System Reference

### Header <boost/units/systems/si.hpp>

Includes all the si unit headers

**Header <boost/units/systems/si/absorbed\_dose.hpp>**

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< absorbed_dose_dimension, si::system > absorbed_dose;

      static const absorbed_dose gray;
      static const absorbed_dose grays;
    }
  }
}
```

## Global gray

boost::units::si::gray

## Synopsis

```
// In header: <boost/units/systems/si/absorbed_dose.hpp>

static const absorbed_dose gray;
```

## Global grays

boost::units::si::grays

## Synopsis

```
// In header: <boost/units/systems/si/absorbed_dose.hpp>

static const absorbed_dose grays;
```

## Header <boost/units/systems/si/acceleration.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< acceleration_dimension, si::system > acceleration;

      static const acceleration meter_per_second_squared;
      static const acceleration meters_per_second_squared;
      static const acceleration metre_per_second_squared;
      static const acceleration metres_per_second_squared;
    }
  }
}
```

## Global meter\_per\_second\_squared

boost::units::si::meter\_per\_second\_squared

## Synopsis

```
// In header: <boost/units/systems/si/acceleration.hpp>

static const acceleration meter_per_second_squared;
```

## Global meters\_per\_second\_squared

boost::units::si::meters\_per\_second\_squared

## Synopsis

```
// In header: <boost/units/systems/si/acceleration.hpp>

static const acceleration meters_per_second_squared;
```

## Global metre\_per\_second\_squared

boost::units::si::metre\_per\_second\_squared

## Synopsis

```
// In header: <boost/units/systems/si/acceleration.hpp>

static const acceleration metre_per_second_squared;
```

## Global metres\_per\_second\_squared

boost::units::si::metres\_per\_second\_squared

## Synopsis

```
// In header: <boost/units/systems/si/acceleration.hpp>

static const acceleration metres_per_second_squared;
```

## Header <boost/units/systems/si/action.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< action_dimension, si::system > action;
    }
  }
}
```

## Header <boost/units/systems/si/activity.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< activity_dimension, si::system > activity;

      static const activity becquerel;
      static const activity becquerels;
    }
  }
}
```



## Global becquerel

boost::units::si::becquerel

## Synopsis

```
// In header: <boost/units/systems/si/activity.hpp>

static const activity becquerel;
```

## Global becquerels

boost::units::si::becquerels

## Synopsis

```
// In header: <boost/units/systems/si/activity.hpp>

static const activity becquerels;
```

## Header <boost/units/systems/si/amount.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< amount_dimension, si::system > amount;

      static const amount mole;
      static const amount moles;
    }
  }
}
```

## Global mole

boost::units::si::mole

## Synopsis

```
// In header: <boost/units/systems/si/amount.hpp>

static const amount mole;
```

## Global moles

boost::units::si::moles

## Synopsis

```
// In header: <boost/units/systems/si/amount.hpp>

static const amount moles;
```

## Header <boost/units/systems/si/angular\_acceleration.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< angular_acceleration_dimension, si::system > angular_acceleration;
    }
  }
}
```

## Header <boost/units/systems/si/angular\_momentum.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< angular_momentum_dimension, si::system > angular_momentum;
    }
  }
}
```

## Header <boost/units/systems/si/angular\_velocity.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< angular_velocity_dimension, si::system > angular_velocity;

      static const angular_velocity radian_per_second;
      static const angular_velocity radians_per_second;
    }
  }
}
```

## Global radian\_per\_second

boost::units::si::radian\_per\_second

## Synopsis

```
// In header: <boost/units/systems/si/angular_velocity.hpp>

static const angular_velocity radian_per_second;
```

## Global radians\_per\_second

boost::units::si::radians\_per\_second

## Synopsis

```
// In header: <boost/units/systems/si/angular_velocity.hpp>

static const angular_velocity radians_per_second;
```

## Header <boost/units/systems/si/area.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< area_dimension, si::system > area;

      static const area square_meter;
      static const area square_meters;
      static const area square_metre;
      static const area square_metres;
    }
  }
}
```

## Global square\_meter

boost::units::si::square\_meter

## Synopsis

```
// In header: <boost/units/systems/si/area.hpp>

static const area square_meter;
```

## Global square\_meters

boost::units::si::square\_meters

## Synopsis

```
// In header: <boost/units/systems/si/area.hpp>

static const area square_meters;
```



## Global square\_metre

boost::units::si::square\_metre

## Synopsis

```
// In header: <boost/units/systems/si/area.hpp>

static const area square_metre;
```

## Global square\_metres

boost::units::si::square\_metres

## Synopsis

```
// In header: <boost/units/systems/si/area.hpp>

static const area square_metres;
```

## Header <boost/units/systems/si/base.hpp>

```
namespace boost {
    namespace units {
        namespace si {
            typedef make_system< meter_base_unit, kilo_
gram_base_unit, second_base_unit, ampere_base_unit, kelvin_base_unit, mole_base_unit, candela_base_unit, angle::radian_base_unit, angle::steradian_base_unit >::type system; // place
holder class defining si unit system
            typedef unit< dimensionless_type, system > dimensionless; // dimensionless si unit
        }
    }
}
```

## Header <boost/units/systems/si/capacitance.hpp>

```
namespace boost {
    namespace units {
        namespace si {
            typedef unit< capacitance_dimension, si::system > capacitance;

            static const capacitance farad;
            static const capacitance farads;
        }
    }
}
```

## Global farad

boost::units::si::farad

## Synopsis

```
// In header: <boost/units/systems/si/capacitance.hpp>

static const capacitance farad;
```

## Global farads

boost::units::si::farads

## Synopsis

```
// In header: <boost/units/systems/si/capacitance.hpp>

static const capacitance farads;
```

## Header <boost/units/systems/si/catalytic\_activity.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef derived_dimension< time_base_dimension, -1, amount_base_dimension, 1 >::type catalytic_activity_dim; // catalytic activity : T^-1 A^1
      typedef unit< si::catalytic_activity_dim, si::system > catalytic_activity;

      static const catalytic_activity katal;
      static const catalytic_activity katals;
    }
  }
}
```

## Global katal

boost::units::si::katal

## Synopsis

```
// In header: <boost/units/systems/si/catalytic_activity.hpp>

static const catalytic_activity katal;
```

## Global katals

boost::units::si::katals

## Synopsis

```
// In header: <boost/units/systems/si/catalytic_activity.hpp>

static const catalytic_activity katals;
```

## Header <boost/units/systems/si/codata/alpha\_constants.hpp>

CODATA recommended values of fundamental atomic and nuclear constants CODATA 2006 values as of 2007/03/30

```
namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(m_alpha, quantity< mass >,
                                         6.64465620e-27 *, 3.3e-34 *);

          // alpha-electron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_alpha_over_m_e,
                                         quantity< dimensionless >,
                                         7294.2995365 * dimensionless,
                                         3.1e-6 * dimensionless);

          // alpha-proton mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_alpha_over_m_p,
                                         quantity< dimensionless >,
                                         3.97259968951 * dimensionless,
                                         4.1e-10 * dimensionless);

          // alpha molar mass
          BOOST_UNITS_PHYSICAL_CONSTANT(M_alpha,
                                         quantity< mass_over_amount >,
                                         4.001506179127e-3 *kilograms/ mole,
                                         6.2e-14 *kilograms/ mole);
        }
      }
    }
  }
}
```

## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/alpha_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(m_alpha, quantity< mass >,
                               6.64465620e-27 * kilograms,
                               3.3e-34 * kilograms);
```

## Description

alpha particle mass

## Header <boost/units/systems/si/codata/atomic-nuclear\_constants.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(alpha, quantity< dimensionless >,
                                         7.2973525376e-3 *, 5.0e-12 *);

          // Rydberg constant.
          BOOST_UNITS_PHYSICAL_CONSTANT(R_infinity, quantity< wavenumber >,
                                         10973731.568527/ meter,
                                         7.3e-5/ meter);

          // Bohr radius.
          BOOST_UNITS_PHYSICAL_CONSTANT(a_0, quantity< length >,
                                         0.52917720859e-10 * meters,
                                         3.6e-20 * meters);

          // Hartree energy.
          BOOST_UNITS_PHYSICAL_CONSTANT(E_h, quantity< energy >,
                                         4.35974394e-18 * joules,
                                         2.2e-25 * joules);
        }
      }
    }
  }
}
```

## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/atomic-nuclear_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(alpha, quantity< dimensionless >,
                               7.2973525376e-3 * dimensionless,
                               5.0e-12 * dimensionless);
```

## Description

fine structure constant

## Header <[boost/units/systems/si/codata/deuteron\\_constants.hpp](#)>

CODATA recommended values of fundamental atomic and nuclear constants CODATA 2006 values as of 2007/03/30



```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(m_d, quantity< mass >,
                                          3.34358320e-27 *, 1.7e-34 *);

          // deuteron-electron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_d_over_m_e,
                                          quantity< dimensionless >,
                                          3670.4829654 * dimensionless,
                                          1.6e-6 * dimensionless);

          // deuteron-proton mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_d_over_m_p,
                                          quantity< dimensionless >,
                                          1.99900750108 * dimensionless,
                                          2.2e-10 * dimensionless);

          // deuteron molar mass
          BOOST_UNITS_PHYSICAL_CONSTANT(M_d, quantity< mass_over_amount >,
                                          2.013553212724e-3 *kilograms/ mole,
                                          7.8e-14 *kilograms/ mole);

          // deuteron rms charge radius
          BOOST_UNITS_PHYSICAL_CONSTANT(R_d, quantity< length >,
                                          2.1402e-15 * meters,
                                          2.8e-18 * meters);

          // deuteron magnetic moment
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_d,
                                          quantity< energy_over_magnetic_flux_density >,
                                          0.433073465e-26 *joules/ tesla,
                                          1.1e-34 *joules/ tesla);

          // deuteron-Bohr magneton ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_d_over_mu_B,
                                          quantity< dimensionless >,
                                          0.4669754556e-3 * dimensionless,
                                          3.9e-12 * dimensionless);

          // deuteron-nuclear magneton ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_d_over_mu_N,
                                          quantity< dimensionless >,
                                          0.8574382308 * dimensionless,
                                          7.2e-9 * dimensionless);

          // deuteron g-factor
          BOOST_UNITS_PHYSICAL_CONSTANT(g_d, quantity< dimensionless >,
                                          0.8574382308 * dimensionless,
                                          7.2e-9 * dimensionless);

          // deuteron-electron magnetic moment ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_d_over_mu_e,
                                          quantity< dimensionless >,
                                          -4.664345537e-4 * dimensionless,
                                          3.9e-12 * dimensionless);

          // deuteron-proton magnetic moment ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_d_over_mu_p,
                                          quantity< dimensionless >,
                                          0.3070122070 * dimensionless,

```

```
                2.4e-9 * dimensionless);  
  
// deuteron-neutron magnetic moment ratio  
BOOST_UNITS_PHYSICAL_CONSTANT(mu_d_over_mu_n,  
                                quantity< dimensionless >,  
                                -0.44820652 * dimensionless,  
                                1.1e-7 * dimensionless);  
        }  
    }  
}
```

## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/deuteron_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(m_d, quantity< mass >,
                               3.34358320e-27 * kilograms,
                               1.7e-34 * kilograms);
```

### Description

deuteron mass

### Header <[boost/units/systems/si/codata/electromagnetic\\_constants.hpp](#)>

CODATA recommended values of fundamental electromagnetic constants.

CODATA recommended values of the fundamental physical constants: NIST SP 961 CODATA 2006 values as of 2007/03/30

```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {

          // elementary charge
          BOOST_UNITS_PHYSICAL_CONSTANT(e, quantity< electric_charge >,
            1.602176487e-19 * coulombs,
            4.0e-27 * coulombs);

          // elementary charge to Planck constant ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(e_over_h,
            quantity< current_over_energy >,
            2.417989454e14 *amperes/ joule,
            6.0e6 *amperes/ joule);

          // magnetic flux quantum
          BOOST_UNITS_PHYSICAL_CONSTANT(Phi_0, quantity< magnetic_flux >,
            2.067833667e-15 * webers,
            5.2e-23 * webers);

          // conductance quantum
          BOOST_UNITS_PHYSICAL_CONSTANT(G_0, quantity< conductance >,
            7.7480917004e-5 * siemens,
            5.3e-14 * siemens);

          // Josephson constant.
          BOOST_UNITS_PHYSICAL_CONSTANT(K_J,
            quantity< frequency_over_electric_potential >,
            483597.891e9 *hertz/ volt,
            1.2e7 *hertz/ volt);

          // von Klitzing constant
          BOOST_UNITS_PHYSICAL_CONSTANT(R_K, quantity< resistance >,
            25812.807557 * ohms, 1.77e-5 * ohms);

          // Bohr magneton.
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_B,
            quantity< energy_over_magnetic_flux_density >,
            927.400915e-26 *joules/ tesla,
            2.3e-31 *joules/ tesla);

          // nuclear magneton
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_N,
            quantity< energy_over_magnetic_flux_density >,
            5.05078324e-27 *joules/ tesla,
            1.3e-34 *joules/ tesla);

        }
      }
    }
  }
}

```

## Header `<boost/units/systems/si/codata/electron_constants.hpp>`

CODATA recommended values of fundamental atomic and nuclear constants CODATA 2006 values as of 2007/03/30

```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(m_e, quantity< mass >,
                                         9.10938215e-31 *, 4.5e-38 *);

          // electron-muon mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_e_over_m_mu,
                                         quantity< dimensionless >,
                                         4.83633171e-3 * dimensionless,
                                         1.2e-10 * dimensionless);

          // electron-tau mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_e_over_m_tau,
                                         quantity< dimensionless >,
                                         2.87564e-4 * dimensionless,
                                         4.7e-8 * dimensionless);

          // electron-proton mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_e_over_m_p,
                                         quantity< dimensionless >,
                                         5.4461702177e-4 * dimensionless,
                                         2.4e-13 * dimensionless);

          // electron-neutron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_e_over_m_n,
                                         quantity< dimensionless >,
                                         5.4386734459e-4 * dimensionless,
                                         3.3e-13 * dimensionless);

          // electron-deuteron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_e_over_m_d,
                                         quantity< dimensionless >,
                                         2.7244371093e-4 * dimensionless,
                                         1.2e-13 * dimensionless);

          // electron-alpha particle mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_e_over_m_alpha,
                                         quantity< dimensionless >,
                                         1.37093355570e-4 * dimensionless,
                                         5.8e-14 * dimensionless);

          // electron charge to mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(e_over_m_e,
                                         quantity< electric_charge_over_mass >,
                                         1.758820150e11 *coulombs/ kilogram,
                                         4.4e3 *coulombs/ kilogram);

          // electron molar mass
          BOOST_UNITS_PHYSICAL_CONSTANT(M_e, quantity< mass_over_amount >,
                                         5.4857990943e-7 *kilograms/ mole,
                                         2.3e-16 *kilograms/ mole);

          // Compton wavelength.
          BOOST_UNITS_PHYSICAL_CONSTANT(lambda_C, quantity< length >,
                                         2.4263102175e-12 * meters,
                                         3.3e-21 * meters);

          // classical electron radius
          BOOST_UNITS_PHYSICAL_CONSTANT(r_e, quantity< length >,
                                         2.8179402894e-15 * meters,

```

```

        5.8e-24 * meters);

// Thompson cross section.
BOOST_UNITS_PHYSICAL_CONSTANT(sigma_e, quantity< area >,
    0.6652458558e-28 * square_meters,
    2.7e-37 * square_meters);

// electron magnetic moment
BOOST_UNITS_PHYSICAL_CONSTANT(mu_e,
    quantity< energy_over_magnetic_flux_density >,
    -928.476377e-26 * joules/ tesla,
    2.3e-31 * joules/ tesla);

// electron-Bohr magneton moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_e_over_mu_B,
    quantity< dimensionless >,
    -1.00115965218111 * dimensionless,
    7.4e-13 * dimensionless);

// electron-nuclear magneton moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_e_over_mu_N,
    quantity< dimensionless >,
    -183.28197092 * dimensionless,
    8.0e-7 * dimensionless);

// electron magnetic moment anomaly
BOOST_UNITS_PHYSICAL_CONSTANT(a_e, quantity< dimensionless >,
    1.15965218111e-3 * dimensionless,
    7.4e-13 * dimensionless);

// electron g-factor
BOOST_UNITS_PHYSICAL_CONSTANT(g_e, quantity< dimensionless >,
    -2.0023193043622 * dimensionless,
    1.5e-12 * dimensionless);

// electron-muon magnetic moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_e_over_mu_mu,
    quantity< dimensionless >,
    206.7669877 * dimensionless,
    5.2e-6 * dimensionless);

// electron-proton magnetic moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_e_over_mu_p,
    quantity< dimensionless >,
    -658.2106848 * dimensionless,
    5.4e-6 * dimensionless);

// electron-shielded proton magnetic moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_e_over_mu_p_prime,
    quantity< dimensionless >,
    -658.2275971 * dimensionless,
    7.2e-6 * dimensionless);

// electron-neutron magnetic moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_e_over_mu_n,
    quantity< dimensionless >,
    960.92050 * dimensionless,
    2.3e-4 * dimensionless);

// electron-deuteron magnetic moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_e_over_mu_d,
    quantity< dimensionless >,
    -2143.923498 * dimensionless,

```

```
        1.8e-5 * dimensionless);

// electron-shielded helion magnetic moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_e_over_mu_h_prime,
    quantity< dimensionless >,
    864.058257 * dimensionless,
    1.0e-5 * dimensionless);

// electron gyromagnetic ratio
BOOST_UNITS_PHYSICAL_CONSTANT(gamma_e,
    quantity< frequency_over_magnetic_flux_density >,
    1.760859770e11/second/ tesla,
    4.4e3/second/ tesla);
}
}
}
}
```

## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/electron_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(m_e, quantity< mass >,
                               9.10938215e-31 * kilograms,
                               4.5e-38 * kilograms);
```

## Description

electron mass

## Header <[boost/units/systems/si/codata/helion\\_constants.hpp](#)>

CODATA recommended values of fundamental atomic and nuclear constants CODATA 2006 values as of 2007/03/30



```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(m_h, quantity< mass >,
                                         5.00641192e-27 *, 2.5e-34 *);

          // helion-electron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_h_over_m_e,
                                         quantity< dimensionless >,
                                         5495.8852765 * dimensionless,
                                         5.2e-6 * dimensionless);

          // helion-proton mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_h_over_m_p,
                                         quantity< dimensionless >,
                                         2.9931526713 * dimensionless,
                                         2.6e-9 * dimensionless);

          // helion molar mass
          BOOST_UNITS_PHYSICAL_CONSTANT(M_h, quantity< mass_over_amount >,
                                         3.0149322473e-3 *kilograms/ mole,
                                         2.6e-12 *kilograms/ mole);

          // helion shielded magnetic moment
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_h_prime,
                                         quantity< energy_over_magnetic_flux_density >,
                                         -1.074552982e-26 *joules/ tesla,
                                         3.0e-34 *joules/ tesla);

          // shielded helion-Bohr magneton ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_h_prime_over_mu_B,
                                         quantity< dimensionless >,
                                         -1.158671471e-3 * dimensionless,
                                         1.4e-11 * dimensionless);

          // shielded helion-nuclear magneton ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_h_prime_over_mu_N,
                                         quantity< dimensionless >,
                                         -2.127497718 * dimensionless,
                                         2.5e-8 * dimensionless);

          // shielded helion-proton magnetic moment ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_h_prime_over_mu_p,
                                         quantity< dimensionless >,
                                         -0.761766558 * dimensionless,
                                         1.1e-8 * dimensionless);

          // shielded helion-shielded proton magnetic moment ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_h_prime_over_mu_p_prime,
                                         quantity< dimensionless >,
                                         -0.7617861313 * dimensionless,
                                         3.3e-8 * dimensionless);

          // shielded helion gyromagnetic ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(gamma_h_prime,
                                         quantity< frequency_over_magnetic_flux_density >,
                                         2.037894730e8/second/ tesla,

```

```

    }
  }
}

5.6e-0/second/ tesla);

```

## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/helion_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(m_h, quantity< mass >,
                               5.00641192e-27 * kilograms,
                               2.5e-34 * kilograms);
```

## Description

helion mass

## Header <[boost/units/systems/si/codata/muon\\_constants.hpp](#)>

CODATA recommended values of fundamental atomic and nuclear constants CODATA 2006 values as of 2007/03/30

```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(m_mu, quantity< mass >,
                                         1.88353130e-28 *, 1.1e-35 *);

          // muon-electron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_mu_over_m_e,
                                         quantity< dimensionless >,
                                         206.7682823 * dimensionless,
                                         5.2e-6 * dimensionless);

          // muon-tau mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_mu_over_m_tau,
                                         quantity< dimensionless >,
                                         5.94592e-2 * dimensionless,
                                         9.7e-6 * dimensionless);

          // muon-proton mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_mu_over_m_p,
                                         quantity< dimensionless >,
                                         0.1126095261 * dimensionless,
                                         2.9e-9 * dimensionless);

          // muon-neutron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_mu_over_m_n,
                                         quantity< dimensionless >,
                                         0.1124545167 * dimensionless,
                                         2.9e-9 * dimensionless);

          // muon molar mass
          BOOST_UNITS_PHYSICAL_CONSTANT(M_mu, quantity< mass_over_amount >,
                                         0.1134289256e-3 *kilograms/ mole,
                                         2.9e-12 *kilograms/ mole);

          // muon Compton wavelength
          BOOST_UNITS_PHYSICAL_CONSTANT(lambda_C_mu, quantity< length >,
                                         11.73444104e-15 * meters,
                                         3.0e-22 * meters);

          // muon magnetic moment
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_mu,
                                         quantity< energy_over_magnetic_flux_density >,
                                         -4.49044786e-26 *joules/ tesla,
                                         1.6e-33 *joules/ tesla);

          // muon-Bohr magneton ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_mu_over_mu_B,
                                         quantity< dimensionless >,
                                         -4.84197049e-3 * dimensionless,
                                         1.2e-10 * dimensionless);

          // muon-nuclear magneton ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_mu_over_mu_N,
                                         quantity< dimensionless >,
                                         -8.89059705 * dimensionless,
                                         2.3e-7 * dimensionless);

          // muon magnetic moment anomaly
          BOOST_UNITS_PHYSICAL_CONSTANT(a_mu, quantity< dimensionless >,
                                         1.16592069e-3 * dimensionless,

```

```
        6.0e-10 * dimensionless);

// muon g-factor
BOOST_UNITS_PHYSICAL_CONSTANT(g_mu, quantity< dimensionless >,
    -2.0023318414 * dimensionless,
    1.2e-9 * dimensionless);

// muon-proton magnetic moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_mu_over_mu_p,
    quantity< dimensionless >,
    -3.183345137 * dimensionless,
    8.5e-8 * dimensionless);
}
}
}
}
```

## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/muon_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(m_mu, quantity< mass >,
                               1.88353130e-28 * kilograms,
                               1.1e-35 * kilograms);
```

## Description

muon mass

## Header <[boost/units/systems/si/codata/neutron\\_constants.hpp](#)>

CODATA recommended values of fundamental atomic and nuclear constants CODATA 2006 values as of 2007/03/30

```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(m_n, quantity< mass >,
                                         1.674927211e-27 *, 8.4e-35 *);

          // neutron-electron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_n_over_m_e,
                                         quantity< dimensionless >,
                                         1838.6836605 * dimensionless,
                                         1.1e-6 * dimensionless);

          // neutron-muon mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_n_over_m_mu,
                                         quantity< dimensionless >,
                                         8.89248409 * dimensionless,
                                         2.3e-7 * dimensionless);

          // neutron-tau mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_n_over_m_tau,
                                         quantity< dimensionless >,
                                         0.528740 * dimensionless,
                                         8.6e-5 * dimensionless);

          // neutron-proton mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_n_over_m_p,
                                         quantity< dimensionless >,
                                         1.00137841918 * dimensionless,
                                         4.6e-10 * dimensionless);

          // neutron molar mass
          BOOST_UNITS_PHYSICAL_CONSTANT(M_n, quantity< mass_over_amount >,
                                         1.00866491597e-3 *kilograms/ mole,
                                         4.3e-13 *kilograms/ mole);

          // neutron Compton wavelength
          BOOST_UNITS_PHYSICAL_CONSTANT(lambda_C_n, quantity< length >,
                                         1.3195908951e-15 * meters,
                                         2.0e-24 * meters);

          // neutron magnetic moment
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_n,
                                         quantity< energy_over_magnetic_flux_density >,
                                         -0.96623641e-26 *joules/ tesla,
                                         2.3e-33 *joules/ tesla);

          // neutron g-factor
          BOOST_UNITS_PHYSICAL_CONSTANT(g_n, quantity< dimensionless >,
                                         -3.82608545 * dimensionless,
                                         9.0e-7 * dimensionless);

          // neutron-electron magnetic moment ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_n_over_mu_e,
                                         quantity< dimensionless >,
                                         1.04066882e-3 * dimensionless,
                                         2.5e-10 * dimensionless);

          // neutron-proton magnetic moment ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_n_over_mu_p,
                                         quantity< dimensionless >,
                                         -0.68497934 * dimensionless,

```

```
        1.6e-7 * dimensionless);

// neutron-shielded proton magnetic moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_n_over_mu_p_prime,
                               quantity< dimensionless >,
                               -0.68499694 * dimensionless,
                               1.6e-7 * dimensionless);

// neutron gyromagnetic ratio
BOOST_UNITS_PHYSICAL_CONSTANT(gamma_n,
                               quantity< frequency_over_magnetic_flux_density >,
                               1.83247185e8/second/ tesla,
                               4.3e1/second/ tesla);
    }
}
}
```



## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/neutron_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(m_n, quantity< mass >,
                               1.674927211e-27 * kilograms,
                               8.4e-35 * kilograms);
```

## Description

neutron mass

## Header <[boost/units/systems/si/codata/physico-chemical\\_constants.hpp](#)>

CODATA recommended values of fundamental physico-chemical constants CODATA 2006 values as of 2007/03/30

```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {

          // Avogadro constant.
          BOOST_UNITS_PHYSICAL_CONSTANT(N_A, quantity< inverse_amount >,
                                         6.02214179e23/ mole, 3.0e16/ mole);

          // atomic mass constant
          BOOST_UNITS_PHYSICAL_CONSTANT(m_u, quantity< mass >,
                                         1.660538782e-27 * kilograms,
                                         8.3e-35 * kilograms);

          // Faraday constant.
          BOOST_UNITS_PHYSICAL_CONSTANT(F,
                                         quantity< electric_charge_over_amount >,
                                         96485.3399 *coulombs/ mole,
                                         2.4e-3 *coulombs/ mole);

          // molar gas constant
          BOOST_UNITS_PHYSICAL_CONSTANT(R,
                                         quantity< energy_over_temperature_amount >,
                                         8.314472 *joules/kelvin/ mole,
                                         1.5e-5 *joules/kelvin/ mole);

          // Boltzmann constant.
          BOOST_UNITS_PHYSICAL_CONSTANT(k_B,
                                         quantity< energy_over_temperature >,
                                         1.3806504e-23 *joules/ kelvin,
                                         2.4e-29 *joules/ kelvin);

          // Stefan-Boltzmann constant.
          BOOST_UNITS_PHYSICAL_CONSTANT(sigma_SB,
                                         quantity< power_over_area_temperature_4 >,
                                         5.670400e-8 *watts/square_meter/pow< 4 >,
                                         4.0e-13 *watts/square_meter/pow< 4 >);

          // first radiation constant
          BOOST_UNITS_PHYSICAL_CONSTANT(c_1, quantity< power_area >,
                                         3.74177118e-16 *watt * square_meters,
                                         1.9e-23 *watt * square_meters);

          // first radiation constant for spectral radiance
          BOOST_UNITS_PHYSICAL_CONSTANT(c_1L,
                                         quantity< power_area_over_solid_angle >,
                                         1.191042759e-16 *watt *square_meters/ steradian,
                                         5.9e-24 *watt *square_meters/ steradian);

          // second radiation constant
          BOOST_UNITS_PHYSICAL_CONSTANT(c_2, quantity< length_temperature >,
                                         1.4387752e-2 *meter * kelvin,
                                         2.5e-8 *meter * kelvin);

          // Wien displacement law constant : lambda_max T.
          BOOST_UNITS_PHYSICAL_CONSTANT(b, quantity< length_temperature >,
                                         2.8977685e-3 *meter * kelvin,
                                         5.1e-9 *meter * kelvin);

          // Wien displacement law constant : nu_max/T.
          BOOST_UNITS_PHYSICAL_CONSTANT(b_prime,
                                         quantity< frequency_over_temperature >,

```

Header [boost/units/systems/si/codata/proton\\_constants.hpp](#)

```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(m_p, quantity< mass >,
                                         1.672621637e-27 *, 8.3e-35 *);

          // proton-electron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_p_over_m_e,
                                         quantity< dimensionless >,
                                         1836.15267247 * dimensionless,
                                         8.0e-7 * dimensionless);

          // proton-muon mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_p_over_m_mu,
                                         quantity< dimensionless >,
                                         8.88024339 * dimensionless,
                                         2.3e-7 * dimensionless);

          // proton-tau mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_p_over_m_tau,
                                         quantity< dimensionless >,
                                         0.528012 * dimensionless,
                                         8.6e-5 * dimensionless);

          // proton-neutron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_p_over_m_n,
                                         quantity< dimensionless >,
                                         0.99862347824 * dimensionless,
                                         4.6e-10 * dimensionless);

          // proton charge to mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(e_over_m_p,
                                         quantity< electric_charge_over_mass >,
                                         9.57883392e7 *coulombs/ kilogram,
                                         2.4e0 *coulombs/ kilogram);

          // proton molar mass
          BOOST_UNITS_PHYSICAL_CONSTANT(M_p, quantity< mass_over_amount >,
                                         1.00727646677e-3 *kilograms/ mole,
                                         1.0e-13 *kilograms/ mole);

          // proton Compton wavelength
          BOOST_UNITS_PHYSICAL_CONSTANT(lambda_C_p, quantity< length >,
                                         1.3214098446e-15 * meters,
                                         1.9e-24 * meters);

          // proton rms charge radius

```

```

BOOST_UNITS_PHYSICAL_CONSTANT(R_p, quantity< length >,
                                0.8768e-15 * meters,
                                6.9e-18 * meters);

// proton magnetic moment
BOOST_UNITS_PHYSICAL_CONSTANT(mu_p,
                                quantity< energy_over_magnetic_flux_density >,
                                1.410606662e-26 *joules/ tesla,
                                3.7e-34 *joules/ tesla);

// proton-Bohr magneton ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_p_over_mu_B,
                                quantity< dimensionless >,
                                1.521032209e-3 * dimensionless,
                                1.2e-11 * dimensionless);

// proton-nuclear magneton ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_p_over_mu_N,
                                quantity< dimensionless >,
                                2.792847356 * dimensionless,
                                2.3e-8 * dimensionless);

// proton g-factor
BOOST_UNITS_PHYSICAL_CONSTANT(g_p, quantity< dimensionless >,
                                5.585694713 * dimensionless,
                                4.6e-8 * dimensionless);

// proton-neutron magnetic moment ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_p_over_mu_n,
                                quantity< dimensionless >,
                                -1.45989806 * dimensionless,
                                3.4e-7 * dimensionless);

// shielded proton magnetic moment
BOOST_UNITS_PHYSICAL_CONSTANT(mu_p_prime,
                                quantity< energy_over_magnetic_flux_density >,
                                1.410570419e-26 *joules/ tesla,
                                3.8e-34 *joules/ tesla);

// shielded proton-Bohr magneton ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_p_prime_over_mu_B,
                                quantity< dimensionless >,
                                1.520993128e-3 * dimensionless,
                                1.7e-11 * dimensionless);

// shielded proton-nuclear magneton ratio
BOOST_UNITS_PHYSICAL_CONSTANT(mu_p_prime_over_mu_N,
                                quantity< dimensionless >,
                                2.792775598 * dimensionless,
                                3.0e-8 * dimensionless);

// proton magnetic shielding correction
BOOST_UNITS_PHYSICAL_CONSTANT(sigma_p_prime,
                                quantity< dimensionless >,
                                25.694e-6 * dimensionless,
                                1.4e-8 * dimensionless);

// proton gyromagnetic ratio
BOOST_UNITS_PHYSICAL_CONSTANT(gamma_p,
                                quantity< frequency_over_magnetic_flux_density >,
                                2.675222099e8/second/ tesla,
                                7.0e0/second/ tesla);

```

```
// shielded proton gyromagnetic ratio
BOOST_UNITS_PHYSICAL_CONSTANT(gamma_p_prime,
                               quantity< frequency_over_magnetic_flux_density >,
                               2.675153362e8/second/ tesla,
                               7.3e0/second/ tesla);
    }
}
}
```

## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/proton_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(m_p, quantity< mass >,
                               1.672621637e-27 * kilograms,
                               8.3e-35 * kilograms);
```

## Description

proton mass

## Header <[boost/units/systems/si/codata/tau\\_constants.hpp](#)>

CODATA recommended values of fundamental atomic and nuclear constants CODATA 2006 values as of 2007/03/30

```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(m_tau, quantity< mass >,
                                         3.16777e-27 *, 5.2e-31 *);

          // tau-electron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_tau_over_m_e,
                                         quantity< dimensionless >,
                                         3477.48 * dimensionless,
                                         5.7e-1 * dimensionless);

          // tau-muon mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_tau_over_m_mu,
                                         quantity< dimensionless >,
                                         16.8183 * dimensionless,
                                         2.7e-3 * dimensionless);

          // tau-proton mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_tau_over_m_p,
                                         quantity< dimensionless >,
                                         1.89390 * dimensionless,
                                         3.1e-4 * dimensionless);

          // tau-neutron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_tau_over_m_n,
                                         quantity< dimensionless >,
                                         1.89129 * dimensionless,
                                         3.1e-4 * dimensionless);

          // tau molar mass
          BOOST_UNITS_PHYSICAL_CONSTANT(M_tau, quantity< mass_over_amount >,
                                         1.90768e-3 *kilograms/ mole,
                                         3.1e-7 *kilograms/ mole);

          // tau Compton wavelength
          BOOST_UNITS_PHYSICAL_CONSTANT(lambda_C_tau, quantity< length >,
                                         0.69772e-15 * meters,
                                         1.1e-19 * meters);
        }
      }
    }
  }
}

```

## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/tau_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(m_tau, quantity< mass >,
                               3.16777e-27 * kilograms, 5.2e-31 * kilograms);
```

## Description

tau mass

## Header <[boost/units/systems/si/codata/triton\\_constants.hpp](#)>

CODATA recommended values of fundamental atomic and nuclear constants CODATA 2006 values as of 2007/03/30



```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(m_t, quantity< mass >,
                                         5.00735588e-27 *, 2.5e-34 *);

          // triton-electron mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_t_over_m_e,
                                         quantity< dimensionless >,
                                         5496.9215269 * dimensionless,
                                         5.1e-6 * dimensionless);

          // triton-proton mass ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(m_t_over_m_p,
                                         quantity< dimensionless >,
                                         2.9937170309 * dimensionless,
                                         2.5e-9 * dimensionless);

          // triton molar mass
          BOOST_UNITS_PHYSICAL_CONSTANT(M_t, quantity< mass_over_amount >,
                                         3.0155007134e-3 *kilograms/ mole,
                                         2.5e-12 *kilograms/ mole);

          // triton magnetic moment
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_t,
                                         quantity< energy_over_magnetic_flux_density >,
                                         1.504609361e-26 *joules/ tesla,
                                         4.2e-34 *joules/ tesla);

          // triton-Bohr magneton ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_t_over_mu_B,
                                         quantity< dimensionless >,
                                         1.622393657e-3 * dimensionless,
                                         2.1e-11 * dimensionless);

          // triton-nuclear magneton ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_t_over_mu_N,
                                         quantity< dimensionless >,
                                         2.978962448 * dimensionless,
                                         3.8e-8 * dimensionless);

          // triton g-factor
          BOOST_UNITS_PHYSICAL_CONSTANT(g_t, quantity< dimensionless >,
                                         5.957924896 * dimensionless,
                                         7.6e-8 * dimensionless);

          // triton-electron magnetic moment ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_t_over_mu_e,
                                         quantity< dimensionless >,
                                         -1.620514423e-3 * dimensionless,
                                         2.1e-11 * dimensionless);

          // triton-proton magnetic moment ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_t_over_mu_p,
                                         quantity< dimensionless >,
                                         1.066639908 * dimensionless,
                                         1.0e-8 * dimensionless);

          // triton-neutron magnetic moment ratio
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_t_over_mu_n,
                                         quantity< dimensionless >,

```

```
        -1.55718553 * dimensionless,  
        3.7e-7 * dimensionless);  
    }  
}  
}
```

## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/triton_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(m_t, quantity< mass >,
                               5.0073588e-27 * kilograms,
                               2.5e-34 * kilograms);
```

## Description

triton mass

## Header <boost/units/systems/si/codata/typedefs.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          typedef divide_typeof_helper< frequency, electric_potential >::type frequency_over_elec-
tric_potential;
          typedef divide_typeof_helper< electric_charge, mass >::type electric_charge_over_mass;
          typedef divide_typeof_helper< mass, amount >::type mass_over_amount;
          typedef divide_typeof_helper< energy, magnetic_flux_density >::type energy_over_magnet-
ic_flux_density;
          typedef divide_typeof_helper< frequency, magnetic_flux_density >::type frequency_over_mag-
netic_flux_density;
          typedef divide_typeof_helper< current, energy >::type current_over_energy;
          typedef divide_typeof_helper< dimensionless, amount >::type inverse_amount;
          typedef divide_typeof_helper< energy, temperature >::type energy_over_temperature;
          typedef divide_typeof_helper< energy_over_temperature, amount >::type energy_over_tem-
perature_amount;
          typedef divide_typeof_helper< divide_typeof_help-
er< power, area >::type, power_typeof_helper< temperature, static_ration-
al< 4 > >::type >::type power_over_area_temperature_4;
          typedef multiply_typeof_helper< power, area >::type power_area;
          typedef divide_typeof_helper< power_area, solid_angle >::type power_area_over_sol-
id_angle;
          typedef multiply_typeof_helper< length, temperature >::type length_temperature;
          typedef divide_typeof_helper< frequency, temperature >::type frequency_over_temperature;
          typedef divide_typeof_helper< divide_typeof_helper< force, current >::type, cur-
rent >::type force_over_current_squared;
          typedef divide_typeof_helper< capacitance, length >::type capacitance_over_length;
          typedef divide_typeof_helper< divide_typeof_helper< divide_typeof_help-
er< volume, mass >::type, time >::type, time >::type volume_over_mass_time_squared;
          typedef multiply_typeof_helper< energy, time >::type energy_time;
          typedef divide_typeof_helper< electric_charge, amount >::type elec-
tric_charge_over_amount;
        }
      }
    }
  }
}
```

Header **<boost/units/systems/si/codata/universal\_constants.hpp>**

CODATA recommended values of fundamental universal constants using CODATA 2006 values as of 2007/03/30

```

namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {
          BOOST_UNITS_PHYSICAL_CONSTANT(c, quantity< velocity >,
                                         299792458.0 *meters/, 0.0 *meters/);

          // magnetic constant (exactly 4 pi x 10^(-7) - error is due to finite precision of pi)
          BOOST_UNITS_PHYSICAL_CONSTANT(mu_0,
                                         quantity< force_over_current_squared >,
                                         12.56637061435917295385057353311801153679e-7 *new.J
                                         tons/ampere/ ampere,
                                         0.0 *newtons/ampere/ ampere);

          // electric constant
          BOOST_UNITS_PHYSICAL_CONSTANT(epsilon_0,
                                         quantity< capacitance_over_length >,
                                         8.854187817620389850536563031710750260608e-12 *farad/ meter,
                                         0.0 *farad/ meter);

          // characteristic impedance of vacuum
          BOOST_UNITS_PHYSICAL_CONSTANT(Z_0, quantity< resistance >,
                                         376.7303134617706554681984004203193082686 * ohm,
                                         0.0 * ohm);

          // Newtonian constant of gravitation.
          BOOST_UNITS_PHYSICAL_CONSTANT(G,
                                         quantity< volume_over_mass_time_squared >,
                                         6.67428e-11 *cubic_meters/kilogram/second/ second,
                                         6.7e-15 *cubic_meters/kilogram/second/ second);

          // Planck constant.
          BOOST_UNITS_PHYSICAL_CONSTANT(h, quantity< energy_time >,
                                         6.62606896e-34 *joule * seconds,
                                         3.3e-41 *joule * seconds);

          // Dirac constant.
          BOOST_UNITS_PHYSICAL_CONSTANT(hbar, quantity< energy_time >,
                                         1.054571628e-34 *joule * seconds,
                                         5.3e-42 *joule * seconds);

          // Planck mass.
          BOOST_UNITS_PHYSICAL_CONSTANT(m_P, quantity< mass >,
                                         2.17644e-8 * kilograms,
                                         1.1e-12 * kilograms);

          // Planck temperature.
          BOOST_UNITS_PHYSICAL_CONSTANT(T_P, quantity< temperature >,
                                         1.416785e32 * kelvin,
                                         7.1e27 * kelvin);

          // Planck length.
          BOOST_UNITS_PHYSICAL_CONSTANT(l_P, quantity< length >,
                                         1.616252e-35 * meters,
                                         8.1e-40 * meters);
        }
      }
    }
  }
}

```

```
// Planck time.  
BOOST_UNITS_PHYSICAL_CONSTANT(t_P, quantity< time >,  
                               5.39124e-44 * seconds,  
                               2.7e-48 * seconds);  
  
}  
}  
}
```

## Function BOOST\_UNITS\_PHYSICAL\_CONSTANT

boost::units::si::constants::codata::BOOST\_UNITS\_PHYSICAL\_CONSTANT — CODATA recommended values of the fundamental physical constants: NIST SP 961.

## Synopsis

```
// In header: <boost/units/systems/si/codata/universal_constants.hpp>

BOOST_UNITS_PHYSICAL_CONSTANT(c, quantity< velocity >,
                               299792458.0 *meters/ second,
                               0.0 *meters/ second);
```

## Description

speed of light

## Header <boost/units/systems/si/conductance.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< conductance_dimension, si::system > conductance;

      static const conductance siemen;
      static const conductance siemens;
      static const conductance mho;
      static const conductance mhos;
    }
  }
}
```

## Global siemen

boost::units::si::siemen

## Synopsis

```
// In header: <boost/units/systems/si/conductance.hpp>

static const conductance siemen;
```

## Global siemens

boost::units::si::siemens

## Synopsis

```
// In header: <boost/units/systems/si/conductance.hpp>

static const conductance siemens;
```



## Global mho

boost::units::si::mho

## Synopsis

```
// In header: <boost/units/systems/si/conductance.hpp>

static const conductance mho;
```

## Global mhos

boost::units::si::mhos

## Synopsis

```
// In header: <boost/units/systems/si/conductance.hpp>

static const conductance mhos;
```

## Header <boost/units/systems/si/conductivity.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< conductivity_dimension, si::system > conductivity;
    }
  }
}
```

## Header <boost/units/systems/si/current.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< current_dimension, si::system > current;

      static const current ampere;
      static const current amperes;
    }
  }
}
```

## Global ampere

boost::units::si::ampere

## Synopsis

```
// In header: <boost/units/systems/si/current.hpp>

static const current ampere;
```

## Global amperes

boost::units::si::amperes

## Synopsis

```
// In header: <boost/units/systems/si/current.hpp>

static const current amperes;
```

Header <[boost/units/systems/si/dimensionless.hpp](#)>

## Global `si_dimensionless`

`boost::units::si::si_dimensionless`

## Synopsis

```
// In header: <boost/units/systems/si/dimensionless.hpp>

static const dimensionless si_dimensionless;
```

## Header `<boost/units/systems/si/dose_equivalent.hpp>`

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< dose_equivalent_dimension, si::system > dose_equivalent;

      static const dose_equivalent sievert;
      static const dose_equivalent sieverts;
    }
  }
}
```

## Global sievert

boost::units::si::sievert

## Synopsis

```
// In header: <boost/units/systems/si/dose_equivalent.hpp>

static const dose_equivalent sievert;
```

## Global sieverts

boost::units::si::sieverts

## Synopsis

```
// In header: <boost/units/systems/si/dose_equivalent.hpp>

static const dose_equivalent sieverts;
```

## Header <boost/units/systems/si/dynamic\_viscosity.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< dynamic_viscosity_dimension, si::system > dynamic_viscosity;
    }
  }
}
```

## Header <boost/units/systems/si/electric\_charge.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< electric_charge_dimension, si::system > electric_charge;

      static const electric_charge coulomb;
      static const electric_charge coulombs;
    }
  }
}
```

## Global coulomb

boost::units::si::coulomb

## Synopsis

```
// In header: <boost/units/systems/si/electric_charge.hpp>

static const electric_charge coulomb;
```



## Global coulombs

boost::units::si::coulombs

## Synopsis

```
// In header: <boost/units/systems/si/electric_charge.hpp>

static const electric_charge coulombs;
```

## Header <boost/units/systems/si/electric\_potential.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< electric_potential_dimension, si::system > electric_potential;

      static const electric_potential volt;
      static const electric_potential volts;
    }
  }
}
```

## Global volt

boost::units::si::volt

## Synopsis

```
// In header: <boost/units/systems/si/electric_potential.hpp>

static const electric_potential volt;
```

## Global volts

boost::units::si::volts

## Synopsis

```
// In header: <boost/units/systems/si/electric_potential.hpp>

static const electric_potential volts;
```

## Header <boost/units/systems/si/energy.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< energy_dimension, si::system > energy;

      static const energy joule;
      static const energy joules;
    }
  }
}
```

## Global joule

boost::units::si::joule

## Synopsis

```
// In header: <boost/units/systems/si/energy.hpp>

static const energy joule;
```

## Global joules

boost::units::si::joules

## Synopsis

```
// In header: <boost/units/systems/si/energy.hpp>

static const energy joules;
```

## Header <boost/units/systems/si/force.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< force_dimension, si::system > force;

      static const force newton;
      static const force newtons;
    }
  }
}
```

## Global newton

boost::units::si::newton

## Synopsis

```
// In header: <boost/units/systems/si/force.hpp>

static const force newton;
```

## Global newtons

boost::units::si::newtons

## Synopsis

```
// In header: <boost/units/systems/si/force.hpp>

static const force newtons;
```

## Header <boost/units/systems/si/frequency.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< frequency_dimension, si::system > frequency;

      static const frequency hertz;
    }
  }
}
```

## Global hertz

boost::units::si::hertz

## Synopsis

```
// In header: <boost/units/systems/si/frequency.hpp>

static const frequency hertz;
```

## Header <boost/units/systems/si/illuminance.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< illuminance_dimension, si::system > illuminance;

      static const illuminance lux;
    }
  }
}
```



## Global lux

boost::units::si::lux

## Synopsis

```
// In header: <boost/units/systems/si/illuminance.hpp>

static const illuminance lux;
```

## Header <boost/units/systems/si/impedance.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< impedance_dimension, si::system > impedance;
    }
  }
}
```

## Header <boost/units/systems/si/inductance.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< inductance_dimension, si::system > inductance;

      static const inductance henry;
      static const inductance henrys;
    }
  }
}
```

## Global henry

boost::units::si::henry

## Synopsis

```
// In header: <boost/units/systems/si/inductance.hpp>

static const inductance henry;
```

## Global henrys

boost::units::si::henrys

## Synopsis

```
// In header: <boost/units/systems/si/inductance.hpp>

static const inductance henrys;
```

## Header <boost/units/systems/si/io.hpp>

```
namespace boost {
  namespace units {
    std::string name_string(const reduce_unit< si::absorbed_dose >::type &);
    std::string symbol_string(const reduce_unit< si::absorbed_dose >::type &);
    std::string name_string(const reduce_unit< si::capacitance >::type &);
    std::string symbol_string(const reduce_unit< si::capacitance >::type &);
    std::string name_string(const reduce_unit< si::catalytic_activity >::type &);
    std::string symbol_string(const reduce_unit< si::catalytic_activity >::type &);
    std::string name_string(const reduce_unit< si::conductance >::type &);
    std::string symbol_string(const reduce_unit< si::conductance >::type &);
    std::string name_string(const reduce_unit< si::electric_charge >::type &);
    std::string symbol_string(const reduce_unit< si::electric_charge >::type &);
    std::string name_string(const reduce_unit< si::electric_potential >::type &);
    std::string symbol_string(const reduce_unit< si::electric_potential >::type &);
    std::string name_string(const reduce_unit< si::energy >::type &);
    std::string symbol_string(const reduce_unit< si::energy >::type &);
    std::string name_string(const reduce_unit< si::force >::type &);
    std::string symbol_string(const reduce_unit< si::force >::type &);
    std::string name_string(const reduce_unit< si::frequency >::type &);
    std::string symbol_string(const reduce_unit< si::frequency >::type &);
    std::string name_string(const reduce_unit< si::illuminance >::type &);
    std::string symbol_string(const reduce_unit< si::illuminance >::type &);
    std::string name_string(const reduce_unit< si::inductance >::type &);
    std::string symbol_string(const reduce_unit< si::inductance >::type &);
    std::string name_string(const reduce_unit< si::luminous_flux >::type &);
    std::string symbol_string(const reduce_unit< si::luminous_flux >::type &);
    std::string name_string(const reduce_unit< si::magnetic_flux >::type &);
    std::string symbol_string(const reduce_unit< si::magnetic_flux >::type &);
    std::string name_string(const reduce_unit< si::magnetic_flux_density >::type &);
    std::string symbol_string(const reduce_unit< si::magnetic_flux_density >::type &);
    std::string name_string(const reduce_unit< si::power >::type &);
    std::string symbol_string(const reduce_unit< si::power >::type &);
    std::string name_string(const reduce_unit< si::pressure >::type &);
    std::string symbol_string(const reduce_unit< si::pressure >::type &);
    std::string name_string(const reduce_unit< si::resistance >::type &);
    std::string symbol_string(const reduce_unit< si::resistance >::type &);
  }
}
```

**Header <boost/units/systems/si/kinematic\_viscosity.hpp>**

```
namespace boost {  
  namespace units {  
    namespace si {  
      typedef unit< kinematic_viscosity_dimension, si::system > kinematic_viscosity;  
    }  
  }  
}
```

**Header <boost/units/systems/si/length.hpp>**

```
namespace boost {  
  namespace units {  
    namespace si {  
      typedef unit< length_dimension, si::system > length;  
  
      static const length meter;  
      static const length meters;  
      static const length metre;  
      static const length metres;  
    }  
  }  
}
```

## Global meter

boost::units::si::meter

## Synopsis

```
// In header: <boost/units/systems/si/length.hpp>

static const length meter;
```

## Global meters

boost::units::si::meters

## Synopsis

```
// In header: <boost/units/systems/si/length.hpp>

static const length meters;
```

## Global metre

boost::units::si::metre

## Synopsis

```
// In header: <boost/units/systems/si/length.hpp>

static const length metre;
```

## Global metres

boost::units::si::metres

## Synopsis

```
// In header: <boost/units/systems/si/length.hpp>

static const length metres;
```

## Header <boost/units/systems/si/luminous\_flux.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< luminous_flux_dimension, si::system > luminous_flux;

      static const luminous_flux lumen;
      static const luminous_flux lumens;
    }
  }
}
```



## Global lumen

boost::units::si::lumen

## Synopsis

```
// In header: <boost/units/systems/si/luminous_flux.hpp>

static const luminous_flux lumen;
```

## Global lumens

boost::units::si::lumens

## Synopsis

```
// In header: <boost/units/systems/si/luminous_flux.hpp>

static const luminous_flux lumens;
```

## Header <boost/units/systems/si/luminous\_intensity.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< luminous_intensity_dimension, si::system > luminous_intensity;

      static const luminous_intensity candela;
      static const luminous_intensity candelas;
    }
  }
}
```

## Global candela

boost::units::si::candela

## Synopsis

```
// In header: <boost/units/systems/si/luminous_intensity.hpp>

static const luminous_intensity candela;
```

## Global candelas

boost::units::si::candelas

## Synopsis

```
// In header: <boost/units/systems/si/luminous_intensity.hpp>

static const luminous_intensity candelas;
```

## Header <boost/units/systems/si/magnetic\_field\_intensity.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< magnetic_field_intensity_dimension, si::system > magnetic_field_intensity;
    }
  }
}
```

## Header <boost/units/systems/si/magnetic\_flux.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< magnetic_flux_dimension, si::system > magnetic_flux;

      static const magnetic_flux weber;
      static const magnetic_flux webers;
    }
  }
}
```

## Global weber

boost::units::si::weber

## Synopsis

```
// In header: <boost/units/systems/si/magnetic_flux.hpp>

static const magnetic_flux weber;
```

## Global webers

boost::units::si::webers

## Synopsis

```
// In header: <boost/units/systems/si/magnetic_flux.hpp>

static const magnetic_flux webers;
```

## Header <boost/units/systems/si/magnetic\_flux\_density.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< magnetic_flux_density_dimension, si::system > magnetic_flux_density;

      static const magnetic_flux_density tesla;
      static const magnetic_flux_density teslas;
    }
  }
}
```

## Global tesla

boost::units::si::tesla

## Synopsis

```
// In header: <boost/units/systems/si/magnetic_flux_density.hpp>

static const magnetic_flux_density tesla;
```

## Global teslas

boost::units::si::teslas

## Synopsis

```
// In header: <boost/units/systems/si/magnetic_flux_density.hpp>

static const magnetic_flux_density teslas;
```

## Header <boost/units/systems/si/mass.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< mass_dimension, si::system > mass;

      static const mass kilogram;
      static const mass kilograms;
      static const mass kilogramme;
      static const mass kilogrammes;
    }
  }
}
```



## Global kilogram

boost::units::si::kilogram

## Synopsis

```
// In header: <boost/units/systems/si/mass.hpp>

static const mass kilogram;
```

## Global kilograms

boost::units::si::kilograms

## Synopsis

```
// In header: <boost/units/systems/si/mass.hpp>

static const mass kilograms;
```

## Global kilogramme

boost::units::si::kilogramme

## Synopsis

```
// In header: <boost/units/systems/si/mass.hpp>

static const mass kilogramme;
```

## Global kilogrammes

boost::units::si::kilogrammes

## Synopsis

```
// In header: <boost/units/systems/si/mass.hpp>

static const mass kilogrammes;
```

## Header <boost/units/systems/si/mass\_density.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< mass_density_dimension, si::system > mass_density;

      static const mass_density kilogram_per_cubic_meter;
      static const mass_density kilograms_per_cubic_meter;
      static const mass_density kilogramme_per_cubic_metre;
      static const mass_density kilogrammes_per_cubic_metre;
    }
  }
}
```

## Global kilogram\_per\_cubic\_meter

boost::units::si::kilogram\_per\_cubic\_meter

## Synopsis

```
// In header: <boost/units/systems/si/mass_density.hpp>

static const mass_density kilogram_per_cubic_meter;
```

## Global kilograms\_per\_cubic\_meter

boost::units::si::kilograms\_per\_cubic\_meter

## Synopsis

```
// In header: <boost/units/systems/si/mass_density.hpp>

static const mass_density kilograms_per_cubic_meter;
```

## Global kilogramme\_per\_cubic\_metre

boost::units::si::kilogramme\_per\_cubic\_metre

## Synopsis

```
// In header: <boost/units/systems/si/mass_density.hpp>

static const mass_density kilogramme_per_cubic_metre;
```

## Global kilogrammes\_per\_cubic\_metre

boost::units::si::kilogrammes\_per\_cubic\_metre

## Synopsis

```
// In header: <boost/units/systems/si/mass_density.hpp>

static const mass_density kilogrammes_per_cubic_metre;
```

## Header <boost/units/systems/si/moment\_of\_inertia.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< moment_of_inertia_dimension, si::system > moment_of_inertia;
    }
  }
}
```

## Header <boost/units/systems/si/momentum.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< momentum_dimension, si::system > momentum;
    }
  }
}
```

## Header <boost/units/systems/si/permeability.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< permeability_dimension, si::system > permeability;
    }
  }
}
```

## Header <boost/units/systems/si/permittivity.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< permittivity_dimension, si::system > permittivity;
    }
  }
}
```



**Header <boost/units/systems/si/plane\_angle.hpp>**

```
namespace boost {  
  namespace units {  
    namespace si {  
      typedef unit< plane_angle_dimension, si::system > plane_angle;  
  
      static const plane_angle radian;  
      static const plane_angle radians;  
    }  
  }  
}
```

## Global radian

boost::units::si::radian

## Synopsis

```
// In header: <boost/units/systems/si/plane_angle.hpp>

static const plane_angle radian;
```

## Global radians

boost::units::si::radians

## Synopsis

```
// In header: <boost/units/systems/si/plane_angle.hpp>

static const plane_angle radians;
```

## Header <boost/units/systems/si/power.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< power_dimension, si::system > power;

      static const power watt;
      static const power watts;
    }
  }
}
```

## Global watt

boost::units::si::watt

## Synopsis

```
// In header: <boost/units/systems/si/power.hpp>

static const power watt;
```

## Global watts

boost::units::si::watts

## Synopsis

```
// In header: <boost/units/systems/si/power.hpp>

static const power watts;
```

## Header <boost/units/systems/si/prefixes.hpp>

```
BOOST_UNITS_METRIC_PREFIX(exponent, name)
```

```
namespace boost {
  namespace units {
    namespace si {
      BOOST_UNITS_METRIC_PREFIX(- 24, yocto);
      BOOST_UNITS_METRIC_PREFIX(- 21, zepto);
      BOOST_UNITS_METRIC_PREFIX(- 18, atto);
      BOOST_UNITS_METRIC_PREFIX(- 15, femto);
      BOOST_UNITS_METRIC_PREFIX(- 12, pico);
      BOOST_UNITS_METRIC_PREFIX(- 9, nano);
      BOOST_UNITS_METRIC_PREFIX(- 6, micro);
      BOOST_UNITS_METRIC_PREFIX(- 3, milli);
      BOOST_UNITS_METRIC_PREFIX(- 2, centi);
      BOOST_UNITS_METRIC_PREFIX(- 1, deci);
      BOOST_UNITS_METRIC_PREFIX(1, deka);
      BOOST_UNITS_METRIC_PREFIX(2, hecto);
      BOOST_UNITS_METRIC_PREFIX(3, kilo);
      BOOST_UNITS_METRIC_PREFIX(6, mega);
      BOOST_UNITS_METRIC_PREFIX(9, giga);
      BOOST_UNITS_METRIC_PREFIX(12, tera);
      BOOST_UNITS_METRIC_PREFIX(15, peta);
      BOOST_UNITS_METRIC_PREFIX(18, exa);
      BOOST_UNITS_METRIC_PREFIX(21, zetta);
      BOOST_UNITS_METRIC_PREFIX(24, yotta);
    }
  }
}
```

## Macro BOOST\_UNITS\_METRIC\_PREFIX

BOOST\_UNITS\_METRIC\_PREFIX

## Synopsis

```
// In header: <boost/units/systems/si/prefixes.hpp>
```

```
BOOST_UNITS_METRIC_PREFIX(exponent, name)
```

## Header <boost/units/systems/si/pressure.hpp>

```
namespace boost {  
    namespace units {  
        namespace si {  
            typedef unit< pressure_dimension, si::system > pressure;  
  
            static const pressure pascal;  
            static const pressure pascals;  
        }  
    }  
}
```

## Global pascal

boost::units::si::pascal

## Synopsis

```
// In header: <boost/units/systems/si/pressure.hpp>

static const pressure pascal;
```

## Global pascals

boost::units::si::pascals

## Synopsis

```
// In header: <boost/units/systems/si/pressure.hpp>

static const pressure pascals;
```

## Header <boost/units/systems/si/reluctance.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< reluctance_dimension, si::system > reluctance;
    }
  }
}
```

## Header <boost/units/systems/si/resistance.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< resistance_dimension, si::system > resistance;

      static const resistance ohm;
      static const resistance ohms;
    }
  }
}
```



## Global ohm

boost::units::si::ohm

## Synopsis

```
// In header: <boost/units/systems/si/resistance.hpp>

static const resistance ohm;
```

## Global ohms

boost::units::si::ohms

## Synopsis

```
// In header: <boost/units/systems/si/resistance.hpp>

static const resistance ohms;
```

## Header <boost/units/systems/si/resistivity.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< resistivity_dimension, si::system > resistivity;
    }
  }
}
```

## Header <boost/units/systems/si/solid\_angle.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< solid_angle_dimension, si::system > solid_angle;

      static const solid_angle steradian;
      static const solid_angle steradians;
    }
  }
}
```

## Global steradian

boost::units::si::steradian

## Synopsis

```
// In header: <boost/units/systems/si/solid_angle.hpp>

static const solid_angle steradian;
```

## Global steradians

boost::units::si::steradians

## Synopsis

```
// In header: <boost/units/systems/si/solid_angle.hpp>

static const solid_angle steradians;
```

## Header <boost/units/systems/si/surface\_density.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< surface_density_dimension, si::system > surface_density;

      static const surface_density kilogram_per_square_meter;
      static const surface_density kilograms_per_square_meter;
      static const surface_density kilogramme_per_square_metre;
      static const surface_density kilogrammes_per_square_metre;
    }
  }
}
```

## Global kilogram\_per\_square\_meter

boost::units::si::kilogram\_per\_square\_meter

## Synopsis

```
// In header: <boost/units/systems/si/surface_density.hpp>

static const surface_density kilogram_per_square_meter;
```

## Global kilograms\_per\_square\_meter

boost::units::si::kilograms\_per\_square\_meter

## Synopsis

```
// In header: <boost/units/systems/si/surface_density.hpp>

static const surface_density kilograms_per_square_meter;
```

## Global kilogramme\_per\_square\_metre

boost::units::si::kilogramme\_per\_square\_metre

## Synopsis

```
// In header: <boost/units/systems/si/surface_density.hpp>

static const surface_density kilogramme_per_square_metre;
```

## Global kilogrammes\_per\_square\_metre

boost::units::si::kilogrammes\_per\_square\_metre

## Synopsis

```
// In header: <boost/units/systems/si/surface_density.hpp>

static const surface_density kilogrammes_per_square_metre;
```

## Header <boost/units/systems/si/surface\_tension.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< surface_tension_dimension, si::system > surface_tension;

      static const surface_tension newton_per_meter;
      static const surface_tension newtons_per_meter;
    }
  }
}
```



## Global newton\_per\_meter

boost::units::si::newton\_per\_meter

## Synopsis

```
// In header: <boost/units/systems/si/surface_tension.hpp>

static const surface_tension newton_per_meter;
```

## Global newtons\_per\_meter

boost::units::si::newtons\_per\_meter

## Synopsis

```
// In header: <boost/units/systems/si/surface_tension.hpp>

static const surface_tension newtons_per_meter;
```

## Header <boost/units/systems/si/temperature.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< temperature_dimension, si::system > temperature;

      static const temperature kelvin;
      static const temperature kelvins;
    }
  }
}
```

## Global kelvin

boost::units::si::kelvin

## Synopsis

```
// In header: <boost/units/systems/si/temperature.hpp>

static const temperature kelvin;
```

## Global kelvins

boost::units::si::kelvins

## Synopsis

```
// In header: <boost/units/systems/si/temperature.hpp>

static const temperature kelvins;
```

## Header <boost/units/systems/si/time.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< time_dimension, si::system > time;

      static const time second;
      static const time seconds;
    }
  }
}
```

## Global second

boost::units::si::second

## Synopsis

```
// In header: <boost/units/systems/si/time.hpp>

static const time second;
```

## Global seconds

boost::units::si::seconds

## Synopsis

```
// In header: <boost/units/systems/si/time.hpp>

static const time seconds;
```

## Header <boost/units/systems/si/torque.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< torque_dimension, si::system > torque;

      static const torque newton_meter;
      static const torque newton_meters;
    }
  }
}
```

## Global newton\_meter

boost::units::si::newton\_meter

## Synopsis

```
// In header: <boost/units/systems/si/torque.hpp>

static const torque newton_meter;
```

## Global newton\_meters

boost::units::si::newton\_meters

## Synopsis

```
// In header: <boost/units/systems/si/torque.hpp>

static const torque newton_meters;
```

## Header <boost/units/systems/si/velocity.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< velocity_dimension, si::system > velocity;

      static const velocity meter_per_second;
      static const velocity meters_per_second;
      static const velocity metre_per_second;
      static const velocity metres_per_second;
    }
  }
}
```



## Global meter\_per\_second

boost::units::si::meter\_per\_second

## Synopsis

```
// In header: <boost/units/systems/si/velocity.hpp>

static const velocity meter_per_second;
```

## Global meters\_per\_second

boost::units::si::meters\_per\_second

## Synopsis

```
// In header: <boost/units/systems/si/velocity.hpp>

static const velocity meters_per_second;
```

## Global metre\_per\_second

boost::units::si::metre\_per\_second

## Synopsis

```
// In header: <boost/units/systems/si/velocity.hpp>

static const velocity metre_per_second;
```

## Global metres\_per\_second

boost::units::si::metres\_per\_second

## Synopsis

```
// In header: <boost/units/systems/si/velocity.hpp>

static const velocity metres_per_second;
```

## Header <boost/units/systems/si/volume.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< volume_dimension, si::system > volume;

      static const volume cubic_meter;
      static const volume cubic_meters;
      static const volume cubic_metre;
      static const volume cubic_metres;
    }
  }
}
```

## Global cubic\_meter

boost::units::si::cubic\_meter

## Synopsis

```
// In header: <boost/units/systems/si/volume.hpp>

static const volume cubic_meter;
```

## Global cubic\_meters

boost::units::si::cubic\_meters

## Synopsis

```
// In header: <boost/units/systems/si/volume.hpp>

static const volume cubic_meters;
```

## Global cubic\_metre

boost::units::si::cubic\_metre

## Synopsis

```
// In header: <boost/units/systems/si/volume.hpp>

static const volume cubic_metre;
```

## Global cubic\_metres

boost::units::si::cubic\_metres

## Synopsis

```
// In header: <boost/units/systems/si/volume.hpp>

static const volume cubic_metres;
```

## Header <boost/units/systems/si/wavenumber.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef unit< wavenumber_dimension, si::system > wavenumber;

      static const wavenumber reciprocal_meter;
      static const wavenumber reciprocal_meters;
      static const wavenumber reciprocal_metre;
      static const wavenumber reciprocal_metres;
    }
  }
}
```



## Global reciprocal\_meter

boost::units::si::reciprocal\_meter

## Synopsis

```
// In header: <boost/units/systems/si/wavenumber.hpp>

static const wavenumber reciprocal_meter;
```

## Global reciprocal\_meters

boost::units::si::reciprocal\_meters

## Synopsis

```
// In header: <boost/units/systems/si/wavenumber.hpp>

static const wavenumber reciprocal_meters;
```

## Global reciprocal\_metre

boost::units::si::reciprocal\_metre

## Synopsis

```
// In header: <boost/units/systems/si/wavenumber.hpp>

static const wavenumber reciprocal_metre;
```

## Global reciprocal\_metres

boost::units::si::reciprocal\_metres

## Synopsis

```
// In header: <boost/units/systems/si/wavenumber.hpp>

static const wavenumber reciprocal_metres;
```

## CGS System Reference

### Header <boost/units/systems/cgs.hpp>

Includes all the cgs unit headers

### Header <boost/units/systems/cgs/acceleration.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< acceleration_dimension, cgs::system > acceleration;

      static const acceleration gal;
      static const acceleration gals;
    }
  }
}
```

## Global gal

boost::units::cgs::gal

## Synopsis

```
// In header: <boost/units/systems/cgs/acceleration.hpp>

static const acceleration gal;
```

## Global gals

boost::units::cgs::gals

## Synopsis

```
// In header: <boost/units/systems/cgs/acceleration.hpp>

static const acceleration gals;
```

## Header <boost/units/systems/cgs/area.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< area_dimension, cgs::system > area;

      static const area square_centimeter;
      static const area square_centimeters;
      static const area square_centimetre;
      static const area square_centimetres;
    }
  }
}
```

## Global square\_centimeter

boost::units::cgs::square\_centimeter

## Synopsis

```
// In header: <boost/units/systems/cgs/area.hpp>

static const area square_centimeter;
```

## Global square\_centimeters

boost::units::cgs::square\_centimeters

## Synopsis

```
// In header: <boost/units/systems/cgs/area.hpp>

static const area square_centimeters;
```



## Global square\_centimetre

boost::units::cgs::square\_centimetre

## Synopsis

```
// In header: <boost/units/systems/cgs/area.hpp>

static const area square_centimetre;
```

## Global square\_centimetres

boost::units::cgs::square\_centimetres

## Synopsis

```
// In header: <boost/units/systems/cgs/area.hpp>

static const area square_centimetres;
```

## Header <boost/units/systems/cgs/base.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef make_system< centi-
meter_base_unit, gram_base_unit, boost::units::si::second_base_unit, biot_base_unit >::type sys-
tem; // placeholder class defining cgs unit system
      typedef unit< dimensionless_type, system > dimensionless; // various unit typedefs for
convenience
    }
  }
}
```

## Header <boost/units/systems/cgs/current.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< current_dimension, cgs::system > current;

      static const current biot;
      static const current biots;
    }
  }
}
```

## Global biot

boost::units::cgs::biot

## Synopsis

```
// In header: <boost/units/systems/cgs/current.hpp>

static const current biot;
```

## Global biots

boost::units::cgs::biots

## Synopsis

```
// In header: <boost/units/systems/cgs/current.hpp>

static const current biots;
```

Header <[boost/units/systems/cgs/dimensionless.hpp](#)>

## Global `cgs_dimensionless`

`boost::units::cgs::cgs_dimensionless`

## Synopsis

```
// In header: <boost/units/systems/cgs/dimensionless.hpp>

static const dimensionless cgs_dimensionless;
```

## Header `<boost/units/systems/cgs/dynamic_viscosity.hpp>`

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< dynamic_viscosity_dimension, cgs::system > dynamic_viscosity;

      static const dynamic_viscosity poise;
    }
  }
}
```

## Global poise

boost::units::cgs::poise

## Synopsis

```
// In header: <boost/units/systems/cgs/dynamic_viscosity.hpp>

static const dynamic_viscosity poise;
```

## Header <boost/units/systems/cgs/energy.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< energy_dimension, cgs::system > energy;

      static const energy erg;
      static const energy ergs;
    }
  }
}
```

## Global erg

boost::units::cgs::erg

## Synopsis

```
// In header: <boost/units/systems/cgs/energy.hpp>

static const energy erg;
```

## Global ergs

boost::units::cgs::ergs

## Synopsis

```
// In header: <boost/units/systems/cgs/energy.hpp>

static const energy ergs;
```

## Header <boost/units/systems/cgs/force.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< force_dimension, cgs::system > force;

      static const force dyne;
      static const force dynes;
    }
  }
}
```



## Global dyne

boost::units::cgs::dyne

## Synopsis

```
// In header: <boost/units/systems/cgs/force.hpp>

static const force dyne;
```

## Global dynes

boost::units::cgs::dynes

## Synopsis

```
// In header: <boost/units/systems/cgs/force.hpp>

static const force dynes;
```

## Header <boost/units/systems/cgs/frequency.hpp>

```
namespace boost {
    namespace units {
        namespace cgs {
            typedef unit< frequency_dimension, cgs::system > frequency;
        }
    }
}
```

## Header <boost/units/systems/cgs/io.hpp>

```
namespace boost {
    namespace units {
        std::string name_string(const reduce_unit< cgs::acceleration >::type &);
        std::string symbol_string(const reduce_unit< cgs::acceleration >::type &);
        std::string name_string(const reduce_unit< cgs::current >::type &);
        std::string symbol_string(const reduce_unit< cgs::current >::type &);
        std::string name_string(const reduce_unit< cgs::dynamic_viscosity >::type &);
        std::string symbol_string(const reduce_unit< cgs::dynamic_viscosity >::type &);
        std::string name_string(const reduce_unit< cgs::energy >::type &);
        std::string symbol_string(const reduce_unit< cgs::energy >::type &);
        std::string name_string(const reduce_unit< cgs::force >::type &);
        std::string symbol_string(const reduce_unit< cgs::force >::type &);
        std::string name_string(const reduce_unit< cgs::kinematic_viscosity >::type &);
        std::string symbol_string(const reduce_unit< cgs::kinematic_viscosity >::type &);
        std::string name_string(const reduce_unit< cgs::pressure >::type &);
        std::string symbol_string(const reduce_unit< cgs::pressure >::type &);
        std::string name_string(const reduce_unit< cgs::wavenumber >::type &);
        std::string symbol_string(const reduce_unit< cgs::wavenumber >::type &);
    }
}
```

## Header <boost/units/systems/cgs/kinematic\_viscosity.hpp>

```
namespace boost {
    namespace units {
        namespace cgs {
            typedef unit< kinematic_viscosity_dimension, cgs::system > kinematic_viscosity;

            static const kinematic_viscosity stoke;
            static const kinematic_viscosity stokes;
        }
    }
}
```

## Global stoke

boost::units::cgs::stoke

## Synopsis

```
// In header: <boost/units/systems/cgs/kinematic_viscosity.hpp>

static const kinematic_viscosity stoke;
```

## Global stokes

boost::units::cgs::stokes

## Synopsis

```
// In header: <boost/units/systems/cgs/kinematic_viscosity.hpp>

static const kinematic_viscosity stokes;
```

## Header <boost/units/systems/cgs/length.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< length_dimension, cgs::system > length;

      static const length centimeter;
      static const length centimeters;
      static const length centimetre;
      static const length centimetres;
    }
  }
}
```

## Global centimeter

boost::units::cgs::centimeter

## Synopsis

```
// In header: <boost/units/systems/cgs/length.hpp>

static const length centimeter;
```

## Global centimeters

boost::units::cgs::centimeters

## Synopsis

```
// In header: <boost/units/systems/cgs/length.hpp>

static const length centimeters;
```

## Global centimetre

boost::units::cgs::centimetre

## Synopsis

```
// In header: <boost/units/systems/cgs/length.hpp>

static const length centimetre;
```

## Global centimetres

boost::units::cgs::centimetres

## Synopsis

```
// In header: <boost/units/systems/cgs/length.hpp>

static const length centimetres;
```

## Header <boost/units/systems/cgs/mass.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< mass_dimension, cgs::system > mass;

      static const mass gram;
      static const mass grams;
      static const mass gramme;
      static const mass grammes;
    }
  }
}
```



## Global gram

boost::units::cgs::gram

## Synopsis

```
// In header: <boost/units/systems/cgs/mass.hpp>

static const mass gram;
```

## Global grams

boost::units::cgs::grams

## Synopsis

```
// In header: <boost/units/systems/cgs/mass.hpp>

static const mass grams;
```

## Global gramme

boost::units::cgs::gramme

## Synopsis

```
// In header: <boost/units/systems/cgs/mass.hpp>

static const mass gramme;
```

## Global grammes

boost::units::cgs::grammes

## Synopsis

```
// In header: <boost/units/systems/cgs/mass.hpp>

static const mass grammes;
```

### Header <boost/units/systems/cgs/mass\_density.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< mass_density_dimension, cgs::system > mass_density;
    }
  }
}
```

### Header <boost/units/systems/cgs/momentum.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< momentum_dimension, cgs::system > momentum;
    }
  }
}
```

### Header <boost/units/systems/cgs/power.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< power_dimension, cgs::system > power;
    }
  }
}
```

### Header <boost/units/systems/cgs/pressure.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< pressure_dimension, cgs::system > pressure;

      static const pressure barye;
      static const pressure baryes;
    }
  }
}
```

## Global barye

boost::units::cgs::barye

## Synopsis

```
// In header: <boost/units/systems/cgs/pressure.hpp>

static const pressure barye;
```

## Global baryes

boost::units::cgs::baryes

## Synopsis

```
// In header: <boost/units/systems/cgs/pressure.hpp>

static const pressure baryes;
```

## Header <boost/units/systems/cgs/time.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< time_dimension, cgs::system > time;

      static const time second;
      static const time seconds;
    }
  }
}
```

## Global second

boost::units::cgs::second

## Synopsis

```
// In header: <boost/units/systems/cgs/time.hpp>

static const time second;
```

## Global seconds

boost::units::cgs::seconds

## Synopsis

```
// In header: <boost/units/systems/cgs/time.hpp>

static const time seconds;
```

## Header <boost/units/systems/cgs/velocity.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< velocity_dimension, cgs::system > velocity;

      static const velocity centimeter_per_second;
      static const velocity centimeters_per_second;
      static const velocity centimetre_per_second;
      static const velocity centimetres_per_second;
    }
  }
}
```



## Global `centimeter_per_second`

`boost::units::cgs::centimeter_per_second`

## Synopsis

```
// In header: <boost/units/systems/cgs/velocity.hpp>

static const velocity centimeter_per_second;
```

## Global centimeters\_per\_second

boost::units::cgs::centimeters\_per\_second

## Synopsis

```
// In header: <boost/units/systems/cgs/velocity.hpp>

static const velocity centimeters_per_second;
```

## Global centimetre\_per\_second

boost::units::cgs::centimetre\_per\_second

## Synopsis

```
// In header: <boost/units/systems/cgs/velocity.hpp>

static const velocity centimetre_per_second;
```

## Global centimetres\_per\_second

boost::units::cgs::centimetres\_per\_second

## Synopsis

```
// In header: <boost/units/systems/cgs/velocity.hpp>

static const velocity centimetres_per_second;
```

## Header <boost/units/systems/cgs/volume.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< volume_dimension, cgs::system > volume;

      static const volume cubic_centimeter;
      static const volume cubic_centimeters;
      static const volume cubic_centimetre;
      static const volume cubic_centimetres;
    }
  }
}
```

## Global cubic\_centimeter

boost::units::cgs::cubic\_centimeter

## Synopsis

```
// In header: <boost/units/systems/cgs/volume.hpp>

static const volume cubic_centimeter;
```

## Global cubic\_centimeters

boost::units::cgs::cubic\_centimeters

## Synopsis

```
// In header: <boost/units/systems/cgs/volume.hpp>

static const volume cubic_centimeters;
```

## Global cubic\_centimetre

boost::units::cgs::cubic\_centimetre

## Synopsis

```
// In header: <boost/units/systems/cgs/volume.hpp>

static const volume cubic_centimetre;
```

## Global cubic\_centimetres

boost::units::cgs::cubic\_centimetres

## Synopsis

```
// In header: <boost/units/systems/cgs/volume.hpp>

static const volume cubic_centimetres;
```

## Header <boost/units/systems/cgs/wavenumber.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef unit< wavenumber_dimension, cgs::system > wavenumber;

      static const wavenumber kayser;
      static const wavenumber kayzers;
      static const wavenumber reciprocal_centimeter;
      static const wavenumber reciprocal_centimeters;
      static const wavenumber reciprocal_centimetre;
      static const wavenumber reciprocal_centimetres;
    }
  }
}
```



## Global kayser

boost::units::cgs::kayser

## Synopsis

```
// In header: <boost/units/systems/cgs/wavenumber.hpp>

static const wavenumber kayser;
```

## Global kayzers

boost::units::cgs::kaysers

## Synopsis

```
// In header: <boost/units/systems/cgs/wavenumber.hpp>

static const wavenumber kayzers;
```

## Global reciprocal\_centimeter

boost::units::cgs::reciprocal\_centimeter

## Synopsis

```
// In header: <boost/units/systems/cgs/wavenumber.hpp>

static const wavenumber reciprocal_centimeter;
```

## Global reciprocal\_centimeters

boost::units::cgs::reciprocal\_centimeters

## Synopsis

```
// In header: <boost/units/systems/cgs/wavenumber.hpp>

static const wavenumber reciprocal_centimeters;
```

## Global reciprocal\_centimetre

boost::units::cgs::reciprocal\_centimetre

## Synopsis

```
// In header: <boost/units/systems/cgs/wavenumber.hpp>

static const wavenumber reciprocal_centimetre;
```

## Global reciprocal\_centimetres

boost::units::cgs::reciprocal\_centimetres

## Synopsis

```
// In header: <boost/units/systems/cgs/wavenumber.hpp>

static const wavenumber reciprocal_centimetres;
```

## Trigonometry and Angle System Reference

### Header <boost/units/systems/angle/degrees.hpp>

```
namespace boost {
  namespace units {
    namespace degree {
      typedef make_system< boost::units::angle::degree_base_unit >::type system;
      typedef unit< dimensionless_type, system > dimensionless;
      typedef unit< plane_angle_dimension, system > plane_angle; // angle degree unit constant

      static const plane_angle degree;
      static const plane_angle degrees;
    }
  }
}
```

## Global degree

boost::units::degree::degree

## Synopsis

```
// In header: <boost/units/systems/angle/degrees.hpp>

static const plane_angle degree;
```

## Global degrees

boost::units::degree::degrees

## Synopsis

```
// In header: <boost/units/systems/angle/degrees.hpp>

static const plane_angle degrees;
```

## Header <boost/units/systems/angle/gradians.hpp>

```
namespace boost {
  namespace units {
    namespace gradian {
      typedef make_system< boost::units::angle::gradian_base_unit >::type system;
      typedef unit< dimensionless_type, system > dimensionless;
      typedef unit< plane_angle_dimension, system > plane_angle; // angle gradian unit constant

      static const plane_angle gradian;
      static const plane_angle radians;
    }
  }
}
```



## Global gradian

boost::units::gradian::gradian

## Synopsis

```
// In header: <boost/units/systems/angle/gradians.hpp>

static const plane_angle gradian;
```

## Global gradients

boost::units::gradian::gradians

## Synopsis

```
// In header: <boost/units/systems/angle/gradians.hpp>

static const plane_angle radians;
```

## Header <boost/units/systems/angle/revolutions.hpp>

```
namespace boost {
  namespace units {
    namespace revolution {
      typedef make_system< boost::units::angle::revolution_base_unit >::type system;
      typedef unit< dimensionless_type, system > dimensionless;
      typedef unit< plane_angle_dimension, system > plane_angle; // angle revolution unit constant

      static const plane_angle revolution;
      static const plane_angle revolutions;
    }
  }
}
```

## Global revolution

boost::units::revolution::revolution

## Synopsis

```
// In header: <boost/units/systems/angle/revolutions.hpp>

static const plane_angle revolution;
```

## Global revolutions

boost::units::revolution::revolutions

## Synopsis

```
// In header: <boost/units/systems/angle/revolutions.hpp>

static const plane_angle revolutions;
```

## Temperature System Reference

### Header <boost/units/systems/temperature/celsius.hpp>

```
namespace boost {
  namespace units {
    namespace celsius {
      typedef make_system< boost::units::temperature::celsius_base_unit >::type system;
      typedef unit< temperature_dimension, system > temperature;

      static const temperature degree;
      static const temperature degrees;
    }
  }
}
```

## Global degree

boost::units::celsius::degree

## Synopsis

```
// In header: <boost/units/systems/temperature/celsius.hpp>

static const temperature degree;
```

## Global degrees

boost::units::celsius::degrees

## Synopsis

```
// In header: <boost/units/systems/temperature/celsius.hpp>

static const temperature degrees;
```

## Header <boost/units/systems/temperature/fahrenheit.hpp>

```
namespace boost {
  namespace units {
    namespace fahrenheit {
      typedef make_system< boost::units::temperature::fahrenheit_base_unit >::type system;
      typedef unit< temperature_dimension, system > temperature;

      static const temperature degree;
      static const temperature degrees;
    }
  }
}
```

## Global degree

boost::units::fahrenheit::degree

## Synopsis

```
// In header: <boost/units/systems/temperature/fahrenheit.hpp>

static const temperature degree;
```

## Global degrees

boost::units::fahrenheit::degrees

## Synopsis

```
// In header: <boost/units/systems/temperature/fahrenheit.hpp>

static const temperature degrees;
```

## Abstract System Reference

### Header <boost/units/systems/abstract.hpp>

```
namespace boost {
namespace units {
template<> struct base_unit_info<abstract::length_unit_tag>;
template<> struct base_unit_info<abstract::mass_unit_tag>;
template<> struct base_unit_info<abstract::time_unit_tag>;
template<> struct base_unit_info<abstract::current_unit_tag>;
template<> struct base_unit_info<abstract::temperature_unit_tag>;
template<> struct base_unit_info<abstract::amount_unit_tag>;
template<> struct base_unit_info<abstract::luminous_intensity_unit_tag>;
template<> struct base_unit_info<abstract::plane_angle_unit_tag>;
template<> struct base_unit_info<abstract::solid_angle_unit_tag>;
namespace abstract {
struct length_unit_tag;
struct mass_unit_tag;
struct time_unit_tag;
struct current_unit_tag;
struct temperature_unit_tag;
struct amount_unit_tag;
struct luminous_intensity_unit_tag;
struct plane_angle_unit_tag;
struct solid_angle_unit_tag;

typedef make_system< length_unit_tag, mass_unit_tag, time_unit_tag, current_unit_tag, temperature_unit_tag, amount_unit_tag, luminous_intensity_unit_tag, plane_angle_unit_tag, solid_angle_unit_tag >::type system;
typedef unit< length_dimension, system > length; // abstract unit of length
typedef unit< mass_dimension, system > mass; // abstract unit of mass
typedef unit< time_dimension, system > time; // abstract unit of time
typedef unit< current_dimension, system > current; // abstract unit of current
typedef unit< temperature_dimension, system > temperature; // abstract unit of temperature
typedef unit< amount_dimension, system > amount; // abstract unit of amount
typedef unit< luminous_intensity_dimension, system > luminous_intensity; // abstract unit of luminous intensity
typedef unit< plane_angle_dimension, system > plane_angle; // abstract unit of plane angle
typedef unit< solid_angle_dimension, system > solid_angle; // abstract unit of solid angle
}
}
}
```



## Struct length\_unit\_tag

boost::units::abstract::length\_unit\_tag

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct length_unit_tag {
};
```

## Struct mass\_unit\_tag

boost::units::abstract::mass\_unit\_tag

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct mass_unit_tag {
};
```

## Struct time\_unit\_tag

boost::units::abstract::time\_unit\_tag

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct time_unit_tag {
};
```

## Struct `current_unit_tag`

`boost::units::abstract::current_unit_tag`

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct current_unit_tag {
};
```

## Struct temperature\_unit\_tag

boost::units::abstract::temperature\_unit\_tag

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct temperature_unit_tag {
};
```

## Struct amount\_unit\_tag

boost::units::abstract::amount\_unit\_tag

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct amount_unit_tag {
};
```

## Struct luminous\_intensity\_unit\_tag

boost::units::abstract::luminous\_intensity\_unit\_tag

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct luminous_intensity_unit_tag {
};
```

## Struct plane\_angle\_unit\_tag

boost::units::abstract::plane\_angle\_unit\_tag

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct plane_angle_unit_tag {
};
```



## Struct solid\_angle\_unit\_tag

boost::units::abstract::solid\_angle\_unit\_tag

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct solid_angle_unit_tag {
};
```

## Struct `base_unit_info<abstract::length_unit_tag>`

`boost::units::base_unit_info<abstract::length_unit_tag>`

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct base_unit_info<abstract::length_unit_tag> {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

### `base_unit_info` public static functions

1. `static std::string name();`
2. `static std::string symbol();`

## Struct `base_unit_info<abstract::mass_unit_tag>`

`boost::units::base_unit_info<abstract::mass_unit_tag>`

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct base_unit_info<abstract::mass_unit_tag> {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

### `base_unit_info` public static functions

1. `static std::string name();`
2. `static std::string symbol();`

## Struct `base_unit_info<abstract::time_unit_tag>`

`boost::units::base_unit_info<abstract::time_unit_tag>`

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct base_unit_info<abstract::time_unit_tag> {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

**`base_unit_info` public static functions**

1. `static std::string name();`
2. `static std::string symbol();`

## Struct `base_unit_info<abstract::current_unit_tag>`

`boost::units::base_unit_info<abstract::current_unit_tag>`

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct base_unit_info<abstract::current_unit_tag> {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

### `base_unit_info` public static functions

1. `static std::string name();`
2. `static std::string symbol();`

## Struct `base_unit_info<abstract::temperature_unit_tag>`

`boost::units::base_unit_info<abstract::temperature_unit_tag>`

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct base_unit_info<abstract::temperature_unit_tag> {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

### `base_unit_info` public static functions

1. `static std::string name();`
2. `static std::string symbol();`

## Struct `base_unit_info<abstract::amount_unit_tag>`

`boost::units::base_unit_info<abstract::amount_unit_tag>`

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct base_unit_info<abstract::amount_unit_tag> {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

### `base_unit_info` public static functions

1. `static std::string name();`
2. `static std::string symbol();`

## Struct `base_unit_info<abstract::luminous_intensity_unit_tag>`

`boost::units::base_unit_info<abstract::luminous_intensity_unit_tag>`

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct base_unit_info<abstract::luminous_intensity_unit_tag> {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

### `base_unit_info` public static functions

1. `static std::string name();`
2. `static std::string symbol();`



## Struct `base_unit_info<abstract::plane_angle_unit_tag>`

`boost::units::base_unit_info<abstract::plane_angle_unit_tag>`

## Synopsis

```
// In header: <boost/units/systems/abstract.hpp>

struct base_unit_info<abstract::plane_angle_unit_tag> {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

### `base_unit_info` public static functions

1. `static std::string name();`
2. `static std::string symbol();`

**Struct `base_unit_info<abstract::solid_angle_unit_tag>`**

`boost::units::base_unit_info<abstract::solid_angle_unit_tag>`

**Synopsis**

```
// In header: <boost/units/systems/abstract.hpp>

struct base_unit_info<abstract::solid_angle_unit_tag> {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

**Description**

**`base_unit_info` public static functions**

1. `static std::string name();`
2. `static std::string symbol();`

**Base Units by Category****Angle Base Units Reference**

Header `<boost/units/base_units/angle/arcminute.hpp>`

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<angle::arcminute_base_unit>;
        namespace angle {
            typedef scaled_base_unit< degree_base_unit, scale< 60, static_rational<-1> > > arcminute_base_unit;
        }
    }
}
```

**Struct `base_unit_info<angle::arcminute_base_unit>`**`boost::units::base_unit_info<angle::arcminute_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/angle/arcminute.hpp>

struct base_unit_info<angle::arcminute_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`

2. `static const char * symbol();`

**Header `<boost/units/base_units/angle/arcsecond.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<angle::arcsecond_base_unit>;
        namespace angle {
            typedef scaled_base_unit< degree_base_unit, scale< 3600, static_rational<-1 > > > arcsecond_base_unit;
        }
    }
}
```

**Struct `base_unit_info<angle::arcsecond_base_unit>`**`boost::units::base_unit_info<angle::arcsecond_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/angle/arcsecond.hpp>

struct base_unit_info<angle::arcsecond_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/angle/degree.hpp>`**

```
namespace boost {
    namespace units {
        namespace angle {
            struct degree_base_unit;
        }
    }
}
```

## Struct degree\_base\_unit

boost::units::angle::degree\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/angle/degree.hpp>

struct degree_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**degree\_base\_unit** public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## Header [<boost/units/base\\_units/angle/gradian.hpp>](#)

```
namespace boost {
    namespace units {
        namespace angle {
            struct gradian_base_unit;
        }
    }
}
```

## Struct `gradian_base_unit`

`boost::units::angle::gradian_base_unit`

# Synopsis

```
// In header: <boost/units/base_units/angle/gradian.hpp>

struct gradian_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

### `gradian_base_unit` public static functions

1. `static const char * name();`
2. `static const char * symbol();`

## Header `<boost/units/base_units/angle/radian.hpp>`

```
namespace boost {
    namespace units {
        namespace angle {
            struct radian_base_unit;
        }
    }
}
```

## Struct `radian_base_unit`

`boost::units::angle::radian_base_unit`

# Synopsis

```
// In header: <boost/units/base_units/angle/radian.hpp>

struct radian_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

**`radian_base_unit` public static functions**

1. `static std::string name();`
2. `static std::string symbol();`

## Header `<boost/units/base_units/angle/revolution.hpp>`

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<angle::revolution_base_unit>;
        namespace angle {
            typedef scaled_base_unit< degree_base_unit, scale< 360, static_rational< 1 > > > revolution_base_unit;
        }
    }
}
```

**Struct `base_unit_info<angle::revolution_base_unit>`**`boost::units::base_unit_info<angle::revolution_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/angle/revolution.hpp>

struct base_unit_info<angle::revolution_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/angle/steradian.hpp>`**

```
namespace boost {
    namespace units {
        namespace angle {
            struct steradian_base_unit;
        }
    }
}
```



## Struct `steradian_base_unit`

`boost::units::angle::steradian_base_unit`

# Synopsis

```
// In header: <boost/units/base_units/angle/steradian.hpp>

struct steradian_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

**`steradian_base_unit` public static functions**

1. `static std::string name();`
2. `static std::string symbol();`

## Astronomical Base Units Reference

Header [`<boost/units/base\_units/astronomical/astronomical\_unit.hpp>`](#)

```
namespace boost {
    namespace units {
        namespace astronomical {
            struct astronomical_unit_base_unit;
        }
    }
}
```

## Struct astronomical\_unit\_base\_unit

boost::units::astronomical::astronomical\_unit\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/astronomical/astronomical_unit.hpp>

struct astronomical_unit_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**astronomical\_unit\_base\_unit public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## Header **<boost/units/base\_units/astronomical/light\_day.hpp>**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<astronomical::light_day_base_unit>;
        namespace astronomical {
            typedef scaled_base_unit< boost::units::astronomical::light_second_base_unit, scale< 86400, static_rational< 1 > > > light_day_base_unit;
        }
    }
}
```

**Struct `base_unit_info<astronomical::light_day_base_unit>`**`boost::units::base_unit_info<astronomical::light_day_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/astronomical/light_day.hpp>

struct base_unit_info<astronomical::light_day_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/astronomical/light_hour.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<astronomical::light_hour_base_unit>;
        namespace astronomical {
            typedef scaled_base_unit< boost::units::astronomic
al::light_second_base_unit, scale< 3600, static_rational< 1 > > > light_hour_base_unit;
        }
    }
}
```

**Struct `base_unit_info<astronomical::light_hour_base_unit>`**`boost::units::base_unit_info<astronomical::light_hour_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/astronomical/light_hour.hpp>

struct base_unit_info<astronomical::light_hour_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/astronomical/light_minute.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<astronomical::light_minute_base_unit>;
        namespace astronomical {
            typedef scaled_base_unit< boost::units::astronomical::light_second_base_unit, scale< 60, static_rational< 1 > > > light_minute_base_unit;
        }
    }
}
```

**Struct `base_unit_info<astronomical::light_minute_base_unit>`**`boost::units::base_unit_info<astronomical::light_minute_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/astronomical/light_minute.hpp>

struct base_unit_info<astronomical::light_minute_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/astronomical/light_second.hpp>`**

```
namespace boost {
    namespace units {
        namespace astronomical {
            struct light_second_base_unit;
        }
    }
}
```

## Struct `light_second_base_unit`

`boost::units::astronomical::light_second_base_unit`

# Synopsis

```
// In header: <boost/units/base_units/astronomical/light_second.hpp>

struct light_second_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

`light_second_base_unit` public static functions

1. `static const char * name();`
2. `static const char * symbol();`

## Header `<boost/units/base_units/astronomical/light_year.hpp>`

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<astronomical::light_year_base_unit>;
        namespace astronomical {
            typedef scaled_base_unit< boost::units::astronomic
al::light_second_base_unit, scale< 31557600, static_rational< 1 > > > light_year_base_unit;
        }
    }
}
```

**Struct `base_unit_info<astronomical::light_year_base_unit>`**`boost::units::base_unit_info<astronomical::light_year_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/astronomical/light_year.hpp>

struct base_unit_info<astronomical::light_year_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/astronomical/parsec.hpp>`**

```
namespace boost {
    namespace units {
        namespace astronomical {
            struct parsec_base_unit;
        }
    }
}
```

**Struct parsec\_base\_unit**

boost::units::astronomical::parsec\_base\_unit

## Synopsis

```
// In header: <boost/units/base_units/astronomical/parsec.hpp>

struct parsec_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

**Description****parsec\_base\_unit public static functions**

1. 

```
static const char * name();
```

2. 

```
static const char * symbol();
```

## CGS Base Units Reference

**Header <boost/units/base\_units/cgs/biot.hpp>**

```
namespace boost {
    namespace units {
        namespace cgs {
            typedef scaled_base_unit< boost::units::si::ampere_base_unit, scale< 10, static_rational<
al<-1 > > > biot_base_unit;
        }
    }
}
```

**Header <boost/units/base\_units/cgs/centimeter.hpp>**

```
namespace boost {
    namespace units {
        namespace cgs {
            typedef scaled_base_unit< boost::units::si::meter_base_unit, scale< 10, static_rational<
al<-2 > > > centimeter_base_unit;
        }
    }
}
```



**Header** <boost/units/base\_units/cgs/gram.hpp>

```
namespace boost {  
    namespace units {  
        namespace cgs {  
            struct gram_base_unit;  
        }  
    }  
}
```

## Struct gram\_base\_unit

boost::units::cgs::gram\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/cgs/gram.hpp>

struct gram_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

**gram\_base\_unit** public static functions

1. 

```
static std::string name();
```
2. 

```
static std::string symbol();
```

## Imperial Base Units Reference

Header **<boost/units/base\_units/imperial/drachm.hpp>**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::drachm_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pound_base_unit, scale< 16, static_ratio< 1
al<-2 > > > drachm_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::drachm_base_unit>`**`boost::units::base_unit_info<imperial::drachm_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/drachm.hpp>

struct base_unit_info<imperial::drachm_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`

2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/fluid_ounce.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::fluid_ounce_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pint_base_unit, scale< 20, static_rational<-1 > > > fluid_ounce_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::fluid_ounce_base_unit>`**`boost::units::base_unit_info<imperial::fluid_ounce_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/fluid_ounce.hpp>

struct base_unit_info<imperial::fluid_ounce_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/foot.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::foot_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< yard_base_unit, scale< 3, static_rational<-1 > > > foot_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::foot_base_unit>`**`boost::units::base_unit_info<imperial::foot_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/foot.hpp>

struct base_unit_info<imperial::foot_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/furlong.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::furlong_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< yard_base_unit, scale< 220, static_rational< 1 > > > furlong_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::furlong_base_unit>`**`boost::units::base_unit_info<imperial::furlong_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/furlong.hpp>

struct base_unit_info<imperial::furlong_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/gallon.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::gallon_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pint_base_unit, scale< 8, static_rational< 1 > > > gal↵
lon_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::gallon_base_unit>`**`boost::units::base_unit_info<imperial::gallon_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/gallon.hpp>

struct base_unit_info<imperial::gallon_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/gill.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::gill_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pint_base_unit, scale< 4, static_rational<-1 > > > gill_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::gill_base_unit>`**`boost::units::base_unit_info<imperial::gill_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/gill.hpp>

struct base_unit_info<imperial::gill_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/grain.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::grain_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pound_base_unit, scale< 7000, static_ratio< 1 >>> grain_base_unit;
        }
    }
}
```



**Struct `base_unit_info<imperial::grain_base_unit>`**`boost::units::base_unit_info<imperial::grain_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/grain.hpp>

struct base_unit_info<imperial::grain_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/hundredweight.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::hundredweight_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pound_base_unit, scale< 112, static_rational< 1 > > > hundredweight_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::hundredweight_base_unit>`**`boost::units::base_unit_info<imperial::hundredweight_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/hundredweight.hpp>

struct base_unit_info<imperial::hundredweight_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/inch.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::inch_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< yard_base_unit, scale< 36, static_rational<-1 > > > inch_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::inch_base_unit>`**`boost::units::base_unit_info<imperial::inch_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/inch.hpp>

struct base_unit_info<imperial::inch_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/league.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::league_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< yard_base_unit, scale< 5280, static_ratio< 1 > > > league_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::league_base_unit>`**`boost::units::base_unit_info<imperial::league_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/league.hpp>

struct base_unit_info<imperial::league_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/imperial/mile.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::mile_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< yard_base_unit, scale< 1760, static_ratio< 1 >>> mile_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::mile_base_unit>`**`boost::units::base_unit_info<imperial::mile_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/mile.hpp>

struct base_unit_info<imperial::mile_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/ounce.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::ounce_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pound_base_unit, scale< 2, static_ratio< 16 >>> ounce_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::ounce_base_unit>`**`boost::units::base_unit_info<imperial::ounce_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/ounce.hpp>

struct base_unit_info<imperial::ounce_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/pint.hpp>`**

```
namespace boost {
    namespace units {
        namespace imperial {
            struct pint_base_unit;
        }
    }
}
```

**Struct pint\_base\_unit**

boost::units::imperial::pint\_base\_unit

## Synopsis

```
// In header: <boost/units/base_units/imperial/pint.hpp>

struct pint_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

**Description**

**pint\_base\_unit** public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header** **<boost/units/base\_units/imperial/pound.hpp>**

```
namespace boost {
    namespace units {
        namespace imperial {
            struct pound_base_unit;
        }
    }
}
```

## Struct pound\_base\_unit

boost::units::imperial::pound\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/imperial/pound.hpp>

struct pound_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**pound\_base\_unit** public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## Header **<boost/units/base\_units/imperial/quart.hpp>**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::quart_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pint_base_unit, scale< 2, static_rational< 1 > > > quart_base_unit;
        }
    }
}
```



**Struct `base_unit_info<imperial::quart_base_unit>`**`boost::units::base_unit_info<imperial::quart_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/quart.hpp>

struct base_unit_info<imperial::quart_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/imperial/quarter.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::quarter_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pound_base_unit, scale< 28, static_ratio< 1 > > > quarter_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::quarter_base_unit>`**`boost::units::base_unit_info<imperial::quarter_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/quarter.hpp>

struct base_unit_info<imperial::quarter_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/stone.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::stone_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pound_base_unit, scale< 14, static_ratio< 1 > > > stone_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::stone_base_unit>`**`boost::units::base_unit_info<imperial::stone_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/stone.hpp>

struct base_unit_info<imperial::stone_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/thou.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::thou_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< yard_base_unit, scale< 36000, static_ratio< 1 >>> thou_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::thou_base_unit>`**`boost::units::base_unit_info<imperial::thou_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/thou.hpp>

struct base_unit_info<imperial::thou_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/ton.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<imperial::ton_base_unit>;
        namespace imperial {
            typedef scaled_base_unit< pound_base_unit, scale< 2240, static_ratio< 1 >>> ton_base_unit;
        }
    }
}
```

**Struct `base_unit_info<imperial::ton_base_unit>`**`boost::units::base_unit_info<imperial::ton_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/imperial/ton.hpp>

struct base_unit_info<imperial::ton_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/imperial/yard.hpp>`**

```
namespace boost {
    namespace units {
        namespace imperial {
            struct yard_base_unit;
        }
    }
}
```

## Struct `yard_base_unit`

`boost::units::imperial::yard_base_unit`

# Synopsis

```
// In header: <boost/units/base_units/imperial/yard.hpp>

struct yard_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**`yard_base_unit` public static functions**

1. `static const char * name();`

2. `static const char * symbol();`

## Metric Base Units Reference

Header [`<boost/units/base\_units/metric/angstrom.hpp>`](#)

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<metric::angstrom_base_unit>;
        namespace metric {
            typedef scaled_base_unit< boost::units::si::meter_base_unit, scale< 10, static_rational<
1<-10 > > > angstrom_base_unit;
        }
    }
}
```

**Struct `base_unit_info<metric::angstrom_base_unit>`**`boost::units::base_unit_info<metric::angstrom_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/metric/angstrom.hpp>

struct base_unit_info<metric::angstrom_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/metric/are.hpp>`**

```
namespace boost {
    namespace units {
        namespace metric {
            struct are_base_unit;
        }
    }
}
```

## Struct are\_base\_unit

boost::units::metric::are\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/metric/are.hpp>

struct are_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**are\_base\_unit** public static functions

1. 

```
static const char * name();
```

2. 

```
static const char * symbol();
```

## Header [<boost/units/base\\_units/metric/atmosphere.hpp>](#)

```
namespace boost {
    namespace units {
        namespace metric {
            struct atmosphere_base_unit;
        }
    }
}
```



## Struct atmosphere\_base\_unit

boost::units::metric::atmosphere\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/metric/atmosphere.hpp>

struct atmosphere_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**atmosphere\_base\_unit** public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## Header [<boost/units/base\\_units/metric/bar.hpp>](#)

```
namespace boost {
    namespace units {
        namespace metric {
            struct bar_base_unit;
        }
    }
}
```

## Struct `bar_base_unit`

`boost::units::metric::bar_base_unit`

# Synopsis

```
// In header: <boost/units/base_units/metric/bar.hpp>

struct bar_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

`bar_base_unit` public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

Header `<boost/units/base_units/metric/barn.hpp>`

```
namespace boost {
    namespace units {
        namespace metric {
            struct barn_base_unit;
        }
    }
}
```

**Struct barn\_base\_unit**`boost::units::metric::barn_base_unit`

## Synopsis

```
// In header: <boost/units/base_units/metric/barn.hpp>

struct barn_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

**Description****barn\_base\_unit public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/metric/day.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<metric::day_base_unit>;
        namespace metric {
            typedef scaled_base_unit< boost::units::si::second_base_unit, scale< 86400, static_rational< 1 > > > day_base_unit;
        }
    }
}
```

**Struct `base_unit_info<metric::day_base_unit>`**`boost::units::base_unit_info<metric::day_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/metric/day.hpp>

struct base_unit_info<metric::day_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/metric/fermi.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<metric::fermi_base_unit>;
        namespace metric {
            typedef scaled_base_unit< boost::units::si::meter_base_unit, scale< 10, static_rational<
al<-15 > > > fermi_base_unit;
        }
    }
}
```

**Struct `base_unit_info<metric::fermi_base_unit>`**`boost::units::base_unit_info<metric::fermi_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/metric/fermi.hpp>

struct base_unit_info<metric::fermi_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/metric/hectare.hpp>`**

```
namespace boost {
    namespace units {
        namespace metric {
            struct hectare_base_unit;
        }
    }
}
```

**Struct hectare\_base\_unit**

boost::units::metric::hectare\_base\_unit

## Synopsis

```
// In header: <boost/units/base_units/metric/hectare.hpp>

struct hectare_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

**Description****hectare\_base\_unit public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header [<boost/units/base\\_units/metric/hour.hpp>](#)**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<metric::hour_base_unit>;
        namespace metric {
            typedef scaled_base_unit< boost::units::si::second_base_unit, scale< 60, static_rationa
al< 2 > > > hour_base_unit;
        }
    }
}
```

**Struct `base_unit_info<metric::hour_base_unit>`**`boost::units::base_unit_info<metric::hour_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/metric/hour.hpp>

struct base_unit_info<metric::hour_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/metric/knot.hpp>`**

```
namespace boost {
    namespace units {
        namespace metric {
            struct knot_base_unit;
        }
    }
}
```

**Struct knot\_base\_unit**

boost::units::metric::knot\_base\_unit

## Synopsis

```
// In header: <boost/units/base_units/metric/knot.hpp>

struct knot_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

**Description**

**knot\_base\_unit** public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header** [\*\*<boost/units/base\\_units/metric/liter.hpp>\*\*](#)

```
namespace boost {
    namespace units {
        namespace metric {
            struct liter_base_unit;
        }
    }
}
```



## Struct liter\_base\_unit

boost::units::metric::liter\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/metric/liter.hpp>

struct liter_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**liter\_base\_unit** public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## Header **<boost/units/base\_units/metric/micron.hpp>**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<metric::micron_base_unit>;
        namespace metric {
            typedef scaled_base_unit< boost::units::si::meter_base_unit, scale< 10, static_rational<-6> > > micron_base_unit;
        }
    }
}
```

**Struct `base_unit_info<metric::micron_base_unit>`**`boost::units::base_unit_info<metric::micron_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/metric/micron.hpp>

struct base_unit_info<metric::micron_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/metric/minute.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<metric::minute_base_unit>;
        namespace metric {
            typedef scaled_base_unit< boost::units::si::second_base_unit, scale< 60, static_ratio< 1 > > > minute_base_unit;
        }
    }
}
```

**Struct `base_unit_info<metric::minute_base_unit>`**`boost::units::base_unit_info<metric::minute_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/metric/minute.hpp>

struct base_unit_info<metric::minute_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/metric/mmHg.hpp>`**

```
namespace boost {
    namespace units {
        namespace metric {
            struct mmHg_base_unit;
        }
    }
}
```

## Struct mmHg\_base\_unit

boost::units::metric::mmHg\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/metric/mmHg.hpp>

struct mmHg_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

### mmHg\_base\_unit public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## Header [<boost/units/base\\_units/metric/nautical\\_mile.hpp>](#)

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<metric::nautical_mile_base_unit>;
        namespace metric {
            typedef scaled_base_unit< boost::units::si::meter_base_unit, scale< 1852, static_ratio< 1 >>> nautical_mile_base_unit;
        }
    }
}
```

**Struct `base_unit_info<metric::nautical_mile_base_unit>`**`boost::units::base_unit_info<metric::nautical_mile_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/metric/nautical_mile.hpp>

struct base_unit_info<metric::nautical_mile_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/metric/ton.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<metric::ton_base_unit>;
        namespace metric {
            typedef scaled_base_unit< boost::units::si::kilogram_base_unit, scale< 1000, static_rational< 1 > > > ton_base_unit;
        }
    }
}
```

**Struct `base_unit_info<metric::ton_base_unit>`**`boost::units::base_unit_info<metric::ton_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/metric/ton.hpp>

struct base_unit_info<metric::ton_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/metric/torr.hpp>`**

```
namespace boost {
    namespace units {
        namespace metric {
            struct torr_base_unit;
        }
    }
}
```

## Struct torr\_base\_unit

boost::units::metric::torr\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/metric/torr.hpp>

struct torr_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**torr\_base\_unit public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## Header [<boost/units/base\\_units/metric/year.hpp>](#)

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<metric::year_base_unit>;
        namespace metric {
            typedef scaled_base_unit< boost::units::si::second_base_unit, scale< 31557600, static_ratio< 1 > > > year_base_unit;
        }
    }
}
```

**Struct `base_unit_info<metric::year_base_unit>`**`boost::units::base_unit_info<metric::year_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/metric/year.hpp>

struct base_unit_info<metric::year_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## SI Base Units Reference

**Header `<boost/units/base_units/si/ampere.hpp>`**

```
namespace boost {
    namespace units {
        namespace si {
            struct ampere_base_unit;
        }
    }
}
```



## Struct `ampere_base_unit`

`boost::units::si::ampere_base_unit`

# Synopsis

```
// In header: <boost/units/base_units/si/ampere.hpp>

struct ampere_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

**`ampere_base_unit` public static functions**

1. `static std::string name();`
2. `static std::string symbol();`

## Header `<boost/units/base_units/si/candela.hpp>`

```
namespace boost {
    namespace units {
        namespace si {
            struct candela_base_unit;
        }
    }
}
```

## Struct `candela_base_unit`

`boost::units::si::candela_base_unit`

# Synopsis

```
// In header: <boost/units/base_units/si/candela.hpp>

struct candela_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

**`candela_base_unit` public static functions**

1. `static std::string name();`
2. `static std::string symbol();`

**Header** [`<boost/units/base\_units/si/kelvin.hpp>`](#)

```
namespace boost {
    namespace units {
        namespace si {
            struct kelvin_base_unit;
        }
    }
}
```

**Struct kelvin\_base\_unit**

boost::units::si::kelvin\_base\_unit

## Synopsis

```
// In header: <boost/units/base_units/si/kelvin.hpp>

struct kelvin_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

**Description****kelvin\_base\_unit public static functions**

1. 

```
static std::string name();
```
2. 

```
static std::string symbol();
```

**Header <boost/units/base\_units/si/kilogram.hpp>**

```
namespace boost {
    namespace units {
        namespace si {
            typedef scaled_base_unit< boost::units::cgs::gram_base_unit, scale< 10, static_ration_1
al< 3 > > > kilogram_base_unit;
        }
    }
}
```

**Header <boost/units/base\_units/si/meter.hpp>**

```
namespace boost {
    namespace units {
        namespace si {
            struct meter_base_unit;
        }
    }
}
```

## Struct meter\_base\_unit

boost::units::si::meter\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/si/meter.hpp>

struct meter_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

**meter\_base\_unit** public static functions

1. 

```
static std::string name();
```
2. 

```
static std::string symbol();
```

**Header** [\*\*<boost/units/base\\_units/si/mole.hpp>\*\*](#)

```
namespace boost {
    namespace units {
        namespace si {
            struct mole_base_unit;
        }
    }
}
```

**Struct mole\_base\_unit**

boost::units::si::mole\_base\_unit

## Synopsis

```
// In header: <boost/units/base_units/si/mole.hpp>

struct mole_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

**Description****mole\_base\_unit public static functions**

1. 

```
static std::string name();
```
2. 

```
static std::string symbol();
```

**Header <boost/units/base\_units/si/second.hpp>**

```
namespace boost {
    namespace units {
        namespace si {
            struct second_base_unit;
        }
    }
}
```

## Struct `second_base_unit`

`boost::units::si::second_base_unit`

# Synopsis

```
// In header: <boost/units/base_units/si/second.hpp>

struct second_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

`second_base_unit` public static functions

1. `static std::string name();`
2. `static std::string symbol();`

## Temperature Base Units Reference

Header [`<boost/units/base\_units/temperature/celsius.hpp>`](#)

```
namespace boost {
    namespace units {
        namespace temperature {
            struct celsius_base_unit;
        }
    }
}
```

**Struct celsius\_base\_unit**

boost::units::temperature::celsius\_base\_unit

## Synopsis

```
// In header: <boost/units/base_units/temperature/celsius.hpp>

struct celsius_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

**Description**

**celsius\_base\_unit** public static functions

1. 

```
static std::string name();
```
2. 

```
static std::string symbol();
```

**Header** [\*\*<boost/units/base\\_units/temperature/fahrenheit.hpp>\*\*](#)

```
namespace boost {
    namespace units {
        namespace temperature {
            struct fahrenheit_base_unit;
        }
    }
}
```

## Struct fahrenheit\_base\_unit

boost::units::temperature::fahrenheit\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/temperature/fahrenheit.hpp>

struct fahrenheit_base_unit {

    // public static functions
    static std::string name();
    static std::string symbol();
};
```

## Description

**fahrenheit\_base\_unit** public static functions

1. 

```
static std::string name();
```
2. 

```
static std::string symbol();
```

## US Base Units Reference

Header **<boost/units/base\_units/us/cup.hpp>**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::cup_base_unit>;
        namespace us {
            typedef scaled_base_unit< pint_base_unit, scale< 2, static_rational<-1 > > > cup_base_unit;
        }
    }
}
```



**Struct `base_unit_info<us::cup_base_unit>`**`boost::units::base_unit_info<us::cup_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/cup.hpp>

struct base_unit_info<us::cup_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/dram.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::dram_base_unit>;
        namespace us {
            typedef scaled_base_unit< pound_base_unit, scale< 16, static_ratio< 1
al<-2 > > > dram_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::dram_base_unit>`**`boost::units::base_unit_info<us::dram_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/dram.hpp>

struct base_unit_info<us::dram_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/fluid_dram.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::fluid_dram_base_unit>;
        namespace us {
            typedef scaled_base_unit< pint_base_unit, scale< 2, static_rational<-7 > > > fluid_dram_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::fluid_dram_base_unit>`**

`boost::units::base_unit_info<us::fluid_dram_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/fluid_dram.hpp>

struct base_unit_info<us::fluid_dram_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

### Header `<boost/units/base_units/us/fluid_ounce.hpp>`

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::fluid_ounce_base_unit>;
        namespace us {
            typedef scaled_base_unit< pint_base_unit, scale< 16, static_rational<-1 > > > fluid_ounce_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::fluid_ounce_base_unit>`**`boost::units::base_unit_info<us::fluid_ounce_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/fluid_ounce.hpp>

struct base_unit_info<us::fluid_ounce_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/us/foot.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::foot_base_unit>;
        namespace us {
            typedef scaled_base_unit< yard_base_unit, scale< 3, static_rational<-1 > > > foot_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::foot_base_unit>`**`boost::units::base_unit_info<us::foot_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/foot.hpp>

struct base_unit_info<us::foot_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/gallon.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::gallon_base_unit>;
        namespace us {
            typedef scaled_base_unit< pint_base_unit, scale< 2, static_rational< 3 > > > gal↵
lon_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::gallon_base_unit>`**`boost::units::base_unit_info<us::gallon_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/gallon.hpp>

struct base_unit_info<us::gallon_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/gill.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::gill_base_unit>;
        namespace us {
            typedef scaled_base_unit< pint_base_unit, scale< 2, static_rational<-2 > > > gill_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::gill_base_unit>`**`boost::units::base_unit_info<us::gill_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/gill.hpp>

struct base_unit_info<us::gill_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/grain.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::grain_base_unit>;
        namespace us {
            typedef scaled_base_unit< pound_base_unit, scale< 7000, static_ratio< 1 >>> grain_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::grain_base_unit>`**`boost::units::base_unit_info<us::grain_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/grain.hpp>

struct base_unit_info<us::grain_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/hundredweight.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::hundredweight_base_unit>;
        namespace us {
            typedef scaled_base_unit< pound_base_unit, scale< 100, static_rational< 1 > > > hundredweight_base_unit;
        }
    }
}
```



**Struct `base_unit_info<us::hundredweight_base_unit>`**`boost::units::base_unit_info<us::hundredweight_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/hundredweight.hpp>

struct base_unit_info<us::hundredweight_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/inch.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::inch_base_unit>;
        namespace us {
            typedef scaled_base_unit< yard_base_unit, scale< 36, static_rational<-1 > > > inch_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::inch_base_unit>`**`boost::units::base_unit_info<us::inch_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/inch.hpp>

struct base_unit_info<us::inch_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/us/mil.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::mil_base_unit>;
        namespace us {
            typedef scaled_base_unit< yard_base_unit, scale< 36000, static_ratio< 1
al<-1 > > > mil_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::mil_base_unit>`**`boost::units::base_unit_info<us::mil_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/mil.hpp>

struct base_unit_info<us::mil_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/us/mile.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::mile_base_unit>;
        namespace us {
            typedef scaled_base_unit< yard_base_unit, scale< 1760, static_ratio< 1 >>> mile_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::mile_base_unit>`**`boost::units::base_unit_info<us::mile_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/mile.hpp>

struct base_unit_info<us::mile_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/minim.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::minim_base_unit>;
        namespace us {
            typedef scaled_base_unit< pint_base_unit, scale< 7680, static_rational<-1 > > > minim_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::minim_base_unit>`**`boost::units::base_unit_info<us::minim_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/minim.hpp>

struct base_unit_info<us::minim_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/ounce.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::ounce_base_unit>;
        namespace us {
            typedef scaled_base_unit< pound_base_unit, scale< 2, static_ratio< 16 > > > ounce_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::ounce_base_unit>`**`boost::units::base_unit_info<us::ounce_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/ounce.hpp>

struct base_unit_info<us::ounce_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/us/pint.hpp>`**

```
namespace boost {
    namespace units {
        namespace us {
            struct pint_base_unit;
        }
    }
}
```

## Struct pint\_base\_unit

boost::units::us::pint\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/us/pint.hpp>

struct pint_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**pint\_base\_unit public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## Header [<boost/units/base\\_units/us/pound.hpp>](#)

```
namespace boost {
    namespace units {
        namespace us {
            struct pound_base_unit;
        }
    }
}
```

## Struct pound\_base\_unit

boost::units::us::pound\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/us/pound.hpp>

struct pound_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**pound\_base\_unit** public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header** [\*\*<boost/units/base\\_units/us/pound\\_force.hpp>\*\*](#)

```
namespace boost {
    namespace units {
        namespace us {
            struct pound_force_base_unit;
        }
    }
}
```



## Struct pound\_force\_base\_unit

boost::units::us::pound\_force\_base\_unit

# Synopsis

```
// In header: <boost/units/base_units/us/pound_force.hpp>

struct pound_force_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

## Description

**pound\_force\_base\_unit** public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## Header [<boost/units/base\\_units/us/quart.hpp>](#)

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::quart_base_unit>;
        namespace us {
            typedef scaled_base_unit< pint_base_unit, scale< 2, static_rational< 1 > > > quart_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::quart_base_unit>`**`boost::units::base_unit_info<us::quart_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/quart.hpp>

struct base_unit_info<us::quart_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/tablespoon.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::tablespoon_base_unit>;
        namespace us {
            typedef scaled_base_unit< pint_base_unit, scale< 2, static_rational<-5 > > > table-
            spoon_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::tablespoon_base_unit>`**`boost::units::base_unit_info<us::tablespoon_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/tablespoon.hpp>

struct base_unit_info<us::tablespoon_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/teaspoon.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::teaspoon_base_unit>;
        namespace us {
            typedef scaled_base_unit< pint_base_unit, scale< 96, static_rational<-1 > > > tea↓
            spoon_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::teaspoon_base_unit>`**`boost::units::base_unit_info<us::teaspoon_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/teaspoon.hpp>

struct base_unit_info<us::teaspoon_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

**Header `<boost/units/base_units/us/ton.hpp>`**

```
namespace boost {
    namespace units {
        template<> struct base_unit_info<us::ton_base_unit>;
        namespace us {
            typedef scaled_base_unit< pound_base_unit, scale< 2000, static_ratio< 1 >>> ton_base_unit;
        }
    }
}
```

**Struct `base_unit_info<us::ton_base_unit>`**`boost::units::base_unit_info<us::ton_base_unit>`

## Synopsis

```
// In header: <boost/units/base_units/us/ton.hpp>

struct base_unit_info<us::ton_base_unit> {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

### Description

**`base_unit_info` public static functions**

1. `static const char * name();`
2. `static const char * symbol();`

**Header `<boost/units/base_units/us/yard.hpp>`**

```
namespace boost {
    namespace units {
        namespace us {
            struct yard_base_unit;
        }
    }
}
```

**Struct yard\_base\_unit**

boost::units::us::yard\_base\_unit

## Synopsis

```
// In header: <boost/units/base_units/us/yard.hpp>

struct yard_base_unit {

    // public static functions
    static const char * name();
    static const char * symbol();
};
```

**Description**

**yard\_base\_unit** public static functions

1. 

```
static const char * name();
```
2. 

```
static const char * symbol();
```

## Alphabetical Listing of Base Units

ampere  
angstrom  
arcminute  
arcsecond  
are  
astronomical\_unit  
atmosphere  
bar  
barn  
biot  
candela  
celsius  
centimeter  
cup  
day  
degree  
drachm  
dram  
fahrenheit  
fermi  
fluid\_dram  
fluid\_ounce (us)  
fluid\_ounce (imperial)  
foot (us)  
foot (imperial)  
furlong  
gallon (us)  
gallon (imperial)

gill (us)  
gill (imperial)  
gadian  
grain (us)  
grain (imperial)  
gram  
hectare  
hour  
hundredweight (us)  
hundredweight (imperial)  
inch (us)  
inch (imperial)  
kelvin  
kilogram  
knot  
league  
light\_day  
light\_hour  
light\_minute  
light\_second  
light\_year  
liter  
meter  
micron  
mil  
mile (us)  
mile (imperial)  
minim  
minute  
mmHg  
mole  
nautical\_mile  
ounce (us)  
ounce (imperial)  
parsec  
pint (us)  
pint (imperial)  
pound (us)  
pound (imperial)  
pound\_force  
quart (us)  
quart (imperial)  
quarter  
radian  
revolution  
second  
steradian  
stone  
tablespoon  
teaspoon  
thou  
ton (us)  
ton (metric)  
ton (imperial)  
torr  
yard (us)  
yard (imperial)  
year

## Installation

The core header files are located in `boost/units`. Unit system headers are located in `<boost/units/systems>`. There are no source files for the library itself - the library is header-only. Example programs demonstrating various aspects of the library can be found in `boost/libs/units/example`. Programs for unit testing are provided in `boost/libs/units/test`.

## FAQ

### How does one distinguish between quantities that are physically different but have the same units (such as energy and torque)?

Because Boost.Units includes plane and solid angle units in the SI system, torque and energy are, in fact, distinguishable (see [torque](#)). In addition, energy is a true [scalar](#) quantity, while torque, despite having the same units as energy if plane angle is not included, is in fact a [pseudovector](#). Thus, a value type representing pseudovectors and encapsulating their algebra could also be implemented.

There are, however, a few SI units that are dimensionally indistinguishable within the SI system. These include the [becquerel](#), which has units identical to frequency (Hz), and the [sievert](#), which is degenerate with the [gray](#). In cases such as this, the proper way to treat this difference is to recognize that expanding the set of base dimensions can provide disambiguation. For example, adding a base dimension for radioactive decays would allow the becquerel to be written as decays/second, differentiating it from the signature of hertz, which is simply 1/second.

### Angles are treated as units

If you don't like this, you can just ignore the angle units and go on your merry way (periodically screwing up when a routine wants degrees and you give it radians instead...)

### Why are there homogeneous systems? Aren't heterogeneous systems sufficient?

Consider the following code:

```
cout << sin(asin(180.0 * degrees));
```

What should this print? If only heterogeneous systems are available it would print 3.14159+ rad Why? Well, `asin` would return a `quantity<dimensionless>` effectively losing the information that degrees are being used. In order to propagate this extra information we need homogeneous systems.

### Why can't I construct a quantity directly from the value type?

This only breaks generic code--which ought to break anyway. The only literal value that ought to be converted to a quantity by generic code is zero, which should be handled by the default constructor. In addition, consider the search and replace problem allowing this poses:

```
quantity<si::length>    q(1.0);
```

Here, the intent is clear - we want a length of one in the SI system, which is one meter. However, imagine some well-intentioned coder attempting to reuse this code, but to have it perform the calculations in the CGS unit system instead. After searching for `si::` and replacing it with `cg::`, we have:



```
quantity<cgs::length> q(1.0);
```

Unfortunately, the meaning of this statement has suddenly changed from one meter to one centimeter. In contrast, as implemented, we begin with:

```
quantity<si::length> q(1.0*si::meter);
```

and, after search and replace:

```
quantity<cgs::length> q(1.0*cgs::meter);
```

which gives us an error. Even if the code has a `@using namespace boost::units::si;` declaration, the latter is still safe, with:

```
using namespace boost::units::si;  
quantity<length> q(1.0*meter);
```

going to

```
using namespace boost::units::cgs;  
quantity<length> q(1.0*meter);
```

The latter will involve an explicit conversion from meters to centimeters, but the value remains correct.

## Why are conversions explicit by default?

Safety and the potential for unintended conversions leading to precision loss and hidden performance costs. Options are provided for forcing implicit conversions between specific units to be allowed.

## Acknowledgements

Matthias C. Schabel would like to acknowledge the Department of Defense for its support of this work under the Prostate Cancer Research Program New Investigator Award W81XWH-04-1-0042 and the National Institutes of Health for their support of this work under the NIBIB Mentored Quantitative Research Development Award K25EB005077.

Thanks to David Walthall for his assistance in debugging and testing on a variety of platforms and Torsten Maehne for his work on interfacing the Boost Units and Boost Lambda libraries.

Thanks to:

- Paul Bristow,
- Michael Fawcett,
- Ben FrantzDale,
- Ron Garcia,
- David Greene,
- Peder Holt,
- Janek Kozicki,
- Andy Little,
- Kevin Lynch,

- Torsten Maehne
- Noah Roberts,
- Andrey Semashev,
- David Walthall,
- Deane Yang,

and all the members of the Boost mailing list who provided their input into the design and implementation of this library.

## Help Wanted

Any help in the following areas would be much appreciated:

- testing on other compilers and operating systems
- performance testing on various architectures
- tutorials

## Version Info

[Boost root](#)

Last edit to Quickbook file M:\data\boost\branches\release\doc\pdf\..\libs\units\doc\units.qbk was at 03:20:07 PM on 2011-Feb-22.



### Tip

This should appear on the pdf version (but may be redundant on html).

## Release Notes

1.2 (March 2010)

- Added autoprefix ready for Boost 1.43

1.0.0 (August 1, 2008) :

- Initial release with Boost 1.36

0.7.1 (March 14, 2007) :

- Boost.Typeof emulation support.
- attempting to rebind a heterogeneous\_system to a different set of dimensions now fails.
- cmath.hpp now works with como-win32.
- minor changes to the tests and examples to make msvc 7.1 happy.

0.7.0 (March 13, 2007) :

- heterogeneous and mixed system functionality added.
- added fine-grained implicit unit conversion on a per fundamental dimension basis.
- added a number of utility metafunction classes and predicates.
- [boost/units/operators.hpp](#) now uses `BOOST_TYPEOF` when possible.
- angular units added in [boost/units/systems/angle/gradians.hpp](#) and [boost/units/systems/angle/gradians.hpp](#). Implicit conversion of radians between trigonometric, SI, and CGS systems is allowed.
- a variety of [unit](#) and [quantity](#) tests added.
- examples now provide self-tests.

0.6.2 (February 22, 2007) :

- changed template order in `unit` so dimension precedes unit system
- added `homogeneous_system<S>` for unit systems
- incorporated changes to [boost/units/dimension.hpp](#) (compile-time sorting by predicate), [boost/units/conversion.hpp](#) (thread-safe implementation of quantity conversions), and [boost/units/io.hpp](#) (now works with any `std::basic_ostream`) by SW
- added abstract units in [boost/units/systems/abstract.hpp](#) to allow abstract dimensional analysis
- new example demonstrating implementation of code based on requirements from Michael Fawcett ([radar\\_beam\\_height.cpp](#))

0.6.1 (February 13, 2007) :

- added metafunctions to test if a type is
  - a valid dimension list (`is_dimension_list<D>`)
  - a unit (`is_unit<T>` and `is_unit_of_system<U, System>`)
  - a quantity (`is_quantity<T>` and `is_quantity_of_system<Q, System>`)
- quantity conversion factor is now computed at compile time
- static constants now avoid ODR problems
- `unit_example_14.cpp` now uses `Boost.Timer`
- numerous minor fixes suggested by SW

0.6.0 (February 8, 2007) :

- incorporated Steven Watanabe's optimized code for `dimension.hpp`, leading to **dramatic** decreases in compilation time (nearly a factor of 10 for `unit_example_4.cpp` in my tests).

0.5.8 (February 7, 2007) :

- fixed `#include` in [boost/units/systems/si/base.hpp](#) (thanks to Michael Fawcett and Steven Watanabe)
- removed references to obsolete `base_type` in [\\_\\_unit\\_info](#) (thanks to Michael Fawcett)
- moved functions in [boost/units/cmath.hpp](#) into `boost::units` namespace (thanks to Steven Watanabe)
- fixed `#include` guards to be consistently named `BOOST_UNITS_XXX` (thanks to Steven Watanabe)

## 0.5.7 (February 5, 2007) :

- changed quantity conversion helper to increase flexibility
- minor documentation changes
- submitted for formal review as a Boost library

## 0.5.6 (January 22, 2007) :

- added IEEE 1541 standard binary prefixes along with SI prefixes to and extended algebra of `scale` and `scaled_value` classes (thanks to Kevin Lynch)
- split SI units into separate header files to minimize the "kitchen sink" include problem (thanks to Janek Kozicki)
- added convenience classes for declaring fundamental dimensions and composite dimensions with integral powers (`fundamental_dimension` and `composite_dimension` respectively)

## 0.5.5 (January 18, 2007) :

- template parameter order in `quantity` switched and default `value_type` of `double` added (thanks to Andrey Semashev and Paul Bristow)
- added implicit `value_type` conversion where allowed (thanks to Andrey Semashev)
- added `quantity_cast` for three cases (thanks to Andrey Semashev):
  - constructing `quantity` from raw `value_type`
  - casting from one `value_type` to another
  - casting from one unit to another (where conversion is allowed)
- added `metre` and `metres` and related constants to the SI system for the convenience of our Commonwealth friends...

## 0.5.4 (January 12, 2007) :

- completely reimplemented unit conversion to allow for arbitrary unit conversions between systems
- strict quantity construction is default; quantities can be constructed from bare values by using static member `from_value`

## 0.5.3 (December 12, 2006) :

- added Boost.Serialization support to `unit` and `quantity` classes
- added option to enforce strict construction of quantities (only constructible by multiplication of scalar by unit or quantity by unit) by preprocessor `MCS_STRICT_QUANTITY_CONSTRUCTION` switch

## 0.5.2 (December 4, 2006) :

- added `<cmath>` wrappers in the `std` namespace for functions that can support quantities

## 0.5.1 (November 3, 2006) :

- converted to Boost Software License
- boostified directory structure and file paths

## 0.5 (November 2, 2006) :

- completely reimplemented SI and CGS unit systems and changed syntax for quantities

- significantly streamlined `pow` and `root` so for most applications it is only necessary to define `power_typeof_helper` and `root_typeof_helper` to gain this functionality
- added a selection of physical constants from the CODATA tables
- added a skeleton `complex` class that correctly supports both `complex<quantity<Y,Unit> >` and `quantity<complex<Y>,Unit>` as an example
- investigate using `Boost.Typeof` for compilers that do not support `typeof`

0.4 (October 13, 2006) :

- `pow<R>` and `root<R>` improved for user-defined types
- added unary `+` and unary `-` operators
- added new example of interfacing with `boost::math::quaternion`
- added optional preprocessor switch to enable implicit unit conversions (`BOOST_UNITS_ENABLE_IMPLICIT_UNIT_CONVERSIONS`)

0.3 (September 6, 2006) :

- Support for `op(x x, Y y)` for g++ added. This is automatically active when compiling with gcc and can be optionally enabled by defining the preprocessor constant `BOOST_UNITS_HAS_TYPEOF`

0.2 (September 4, 2006) : Second alpha release based on slightly modified code from 0.1 release

0.1 (December 13, 2003) : written as a Boost demonstration of MPL-based dimensional analysis in 2003.

## TODO

- Document concepts
- Implementation of I/O is rudimentary; consider methods of i18n using facets
- Consider runtime variant, perhaps using overload like `quantity<runtime,Y>`