Boost.Units 0.9.0

Matthias C. Schabel

Steven Watanabe

Table of Contents

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 and 10.0 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. Metrowerks CodeWarrior 9.2 on Windows XP.

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.

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, area = [L]^2, velocity = [L]^1/[T]^1, and 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. kg^1 m^1/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/io.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>

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

quantity<energy>
work(const quantity<force>& F,const quantity<length>& dx)
{
    return F*dx;
}

int main()
{
    /// test calcuation of work
    quantity<force>     F(2.0*newton);
    quantity<length>    dx(2.0*meter);
    quantity<energy>    E(work(F,dx));

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

    /// check complex quantities
    typedef std::complex<double>    complex_type;

    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
              << "I*Z = " << i*z << std::endl
              << "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 m kg s^(-2)
dx  = 2 m
E   = 4 m^2 kg s^(-2)

V    = (12.5,0) m^2 kg s^(-3) A^(-1)
I    = (3,4) A
Z    = (1.5,-2) m^2 kg s^(-3) A^(-2)
I*Z = (12.5,0) m^2 kg s^(-3) A^(-1)
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 Systeme 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 $\{D_1, D_2, \ldots, D_p\}$ and a set of p rational exponents $\{R_1, R_2, \ldots, R_p\}$, any possible (composite) dimension can be written as $D = \{D_1^{R_1}, D_2^{R_2}, \ldots, D_p^{R_p}\}$.

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

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

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 linear combination of base units. Thus, the SI unit corresponding to the dimension of force is kg m s^-2, 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^(1/2), 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 `unit`s. 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 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);
```

We also specialize the `base_unit_info` class for each fundamental dimension tag to provide information needed for I/O:

```
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.

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.

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 units reduce to being 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 precision loss 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`.

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.

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):

- $\mathbb{N} + \mathbb{N} \rightarrow \mathbb{N}$
- $\mathbb{N} - \mathbb{N} \rightarrow \mathbb{Z}$

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

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

The macros needed for defining conversion can be found in `boost/units/conversion.hpp` and `boost/units/absolute.hpp` (for conversion offsets)

The macro `BOOST_UNITS_DEFINE_CONVERSION_FACTOR` specifies the factor to convert from the first argument 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, 1> {};
```

Now, we want to be able to convert feet to meters and vice versa. A foot is 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 can use the SI length.

```
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 have exactly the same effect.

If these conversions have been defined, then converting between scaled forms of these units will 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 arbitray conversion 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);
```

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 `boost/units/detail/utility.hpp`)

```
length_dimension  = list<dim<length_base_dimension, static_rational<1l, 1l> >, dimensionles
mass_dimension    = list<dim<mass_base_dimension, static_rational<1l, 1l> >, dimensionless_
time_dimension    = list<dim<time_base_dimension, static_rational<1l, 1l> >, dimensionless_
energy_dimension  = list<dim<length_base_dimension, static_rational<2l, 1l> >, list<dim<mas
LM_type      = list<dim<length_base_dimension, static_rational<1l, 1l> >, list<dim<mass_bas
L_T_type     = list<dim<length_base_dimension, static_rational<1l, 1l> >, list<dim<time_bas
V_type       = list<dim<length_base_dimension, static_rational<1l, 1l> >, list<dim<time_bas
```

Unit Example

(unit.cpp)

This example demonstrates the use of the simple but functional unit system implemented in `libs/units/example/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

(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 calcuation 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;

    #if BOOST_UNITS_HAS_TYPEOF
    using namespace constants::codata;
    return (P*V/(R*T));
    #else
    return P*V/(8.314472*(joules/(kelvin*mole))*T);
    #endif // BOOST_UNITS_HAS_TYPEOF
}
```

```
/// 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. 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 :

[kitchen_sink_function_snippet_1]

[kitchen_sink_function_snippet_2]

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) ##
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 cm^2 g s^-2
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`.

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

typedef 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

} // namespace units

} // namespace boost

BOOST_UNITS_DEFINE_CONVERSION_FACTOR(boost::units::nautical::length_base_unit,
                                     boost::units::si::meter_base_unit,
                                     double, 1.852e3);

namespace boost {

namespace units {
```

and the imperial foot :

```
namespace imperial {

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

typedef make_system<length_base_unit>::type system;

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

static const length foot,feet;

} // imperial

} // namespace units

} // namespace boost

BOOST_UNITS_DEFINE_CONVERSION_FACTOR(boost::units::imperial::length_base_unit,
                                     boost::units::si::meter_base_unit,
                                     double, 1.0/3.28083989501312);

namespace boost {

namespace units {
```

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<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
const quantity<nautical::length> beam_height_2(radar_beam_height(radar_range,quantity<nauti
const quantity<si::length>       beam_height_3(radar_beam_height< quantity<si::length> >(ra
const quantity<nautical::length> beam_height_4(radar_beam_height< quantity<nautical::length
```

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.

First a look at the output:

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

sstream1 << L << std::endl
         << M << std::endl
         << L*M << std::endl
         << L/M << std::endl
         << std::endl;

sstream1 << 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;

sstream1 << 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;
```

printing

```
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:

[heterogeneous_unit_snippet_2]

yielding

```
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:

```
// direct method
//typedef make_system<temperature::fahrenheit_base_unit>::type  system;
//typedef unit<temperature_dimension,system>                    temperature;

// simpler method for single-unit systems
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);
```

For convenience we make conversions implicit:

```
template<>
struct is_implicitly_convertible<
           fahrenheit::temperature,
           si::temperature > : public mpl::true_ { };

template<>
struct is_implicitly_convertible<
           absolute<fahrenheit::temperature>,
           absolute<si::temperature> > : public mpl::true_ { };
```

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<absolute<si::temperature> >            T3p = T1p;
quantity<si::temperature>                       T2v(T1v);
quantity<si::temperature>                       T3v = T1v;
```

Note the use of `absolute` to wrap a unit.

Runtime Conversion Factor Example

(runtime_conversion_factor.cpp)

The Units library does not require that the conversion factors be compile time constants.

```
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 possible to define base units that do not have 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)

You can overload the ostream operator for a unit if it has a special symbol, in this case Newtons.

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

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

Conversion Factor

(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 << "\""
                         << 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;
            }
        }
    }
```

```
}
```

Utilities

Relatively complete SI and CGS unit systems are provided in `boost/units/systems/si.hpp` and `boost/units/systems/cgs.hpp`, 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 Tag,class System1,class System2> struct base_unit_is_implicitly_convertible;
template<class S1,class D1,class S2,class D2> struct is_implicitly_convertible< unit<D1,S1>

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,Sy
template<class Dim,class System,class Y> struct is_quantity_of_dimension< quantity<unit<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,Sy
template<class System,class Y> struct is_dimensionless_quantity< quantity<unit<dimensionles
```

Reference

Units Reference

Header <boost/units/absolute.hpp>

```


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 Y>
      std::ostream & operator<<(std::ostream & os, const absolute< Y > & aval);
  }
}
```

Class template absolute

boost::units::absolute

Synopsis

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

  // construct/copy/destruct
  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/destruct

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

```
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>

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

Class template base_dimension

boost::units::base_dimension

Synopsis

```
template<typename Derived, long N>
class base_dimension {
public:
  // types
  typedef unspecified dimension_type;  // A convenience typedef. Equivalent to boost::units
  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>

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

Class template base_unit

boost::units::base_unit

Synopsis

```
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>

```
namespace boost {
  namespace units {

    // 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 > &

    // 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 > &

    // 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 > &

    // atan2 of value_type returning angle in radians
    template<typename Y, typename System>
      quantity< unit< plane_angle_dimension, homogeneous_system< System > >, Y >
      atan2(const quantity< unit< dimensionless_type, homogeneous_system< System > >, Y > &
            const quantity< unit< dimensionless_type, homogeneous_system< System > >, Y > &
  }
}
```

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

```
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

```
BOOST_UNITS_NO_COMPILER_CHECK
```

Description

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

Macro BOOST_UNITS_CHECK_HOMOGENEOUS_UNITS

BOOST_UNITS_CHECK_HOMOGENEOUS_UNITS

Synopsis

```
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>

```


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

```
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()));
    }
};
}
}
```

`conversion_helper` public static functions

1.
```
static To convert(const From &) ;
```

Macro BOOST_UNITS_DEFINE_CONVERSION_FACTOR

BOOST_UNITS_DEFINE_CONVERSION_FACTOR

Synopsis

```
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.

Macro BOOST_UNITS_DEFINE_CONVERSION_FACTOR_TEMPLATE

BOOST_UNITS_DEFINE_CONVERSION_FACTOR_TEMPLATE

Synopsis

```
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

```
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

```
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

```
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

```
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 a 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

```
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 had argument from the corresponding exponents of the left had 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

```
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

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

Header <boost/units/dimensionless_quantity.hpp>

```
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

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

Header <boost/units/dimensionless_type.hpp>

```
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

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

Header <boost/units/dimensionless_unit.hpp>

```
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

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

Header <boost/units/get_dimension.hpp>

```
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

```
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

```
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

```
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

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

Header <boost/units/get_system.hpp>

```
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

```
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

```
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

```
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

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

Header <boost/units/heterogeneous_system.hpp>

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

Struct template heterogeneous_system

boost::units::heterogeneous_system

Synopsis

```
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

```
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>

```
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;

    enum format_mode { symbol_fmt =  0, name_fmt, raw_fmt, typename_fmt };
    template<typename T> std::string to_string(const T & t);
    template<integer_type N>
      std::string to_string(const static_rational< N > &);
    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);
    format_mode get_format(std::ios_base & ios);
    void set_format(std::ios_base & ios, format_mode new_mode);
    std::ios_base & typename_format(std::ios_base & ios);
    std::ios_base & raw_format(std::ios_base & ios);
    std::ios_base & symbol_format(std::ios_base & ios);
    std::ios_base & name_format(std::ios_base & ios);
    template<typename Dimension, typename System>
      std::string typename_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 > &);
    namespace io_impl {
      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 Dimension, typename System>
        std::string name_string(const unit< Dimension, heterogeneous_system< System > > &);
    }
  }
}
```

Struct template base_unit_info

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

Synopsis

```
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() ;
```

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

Function template operator<<

boost::units::operator<<

Synopsis

```
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 an `unit` as a list of base units and exponents

for `symbol_format` this gives e.g. "m s^-1" or "J" for `name_format` this gives e.g. "meter second^-1" or "joule" for `raw_format` this gives e.g. "m s^-1" or "meter kilogram^2 second^-2" for `typename_format` this gives the typename itself (currently demangled only on GCC)

Header <boost/units/is_dim.hpp>

```
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

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

Struct template is_dim<dim< T, V >>

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

Synopsis

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

Header <boost/units/is_dimension_list.hpp>

```
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

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

Struct template is_dimension_list<list< Item, Next >>

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

Synopsis

```
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

```
struct is_dimension_list<dimensionless_type> {
};
```

Header <boost/units/is_dimensionless.hpp>

```
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

```
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

```
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

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

Header <boost/units/is_dimensionless_quantity.hpp>

```
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

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

Header <boost/units/is_dimensionless_unit.hpp>

```
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

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

Header <boost/units/is_quantity.hpp>

```
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

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

Struct template is_quantity<quantity< Unit, Y >>

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

Synopsis

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

Header <boost/units/is_quantity_of_dimension.hpp>

```
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

```
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

```
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>

```
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

```
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

```
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>

```
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

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

Struct template is_unit<unit< Dim, System >>

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

Synopsis

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

Header <boost/units/is_unit_of_dimension.hpp>

```
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

```
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

```
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

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

Header <boost/units/is_unit_of_system.hpp>

```
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

```
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

```
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

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

Header <boost/units/limits.hpp>

```
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

```
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 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 float_denorm_style has_denorm;
  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_round_style round_style;
};
```

Description

`numeric_limits` public static functions

1.
```
static quantity_type epsilon() ;
```

2.
```
static quantity_type round_error() ;
```

3.
```
static quantity_type infinity() ;
```

4.
```
static quantity_type quiet_NaN() ;
```

5.
```
static quantity_type signaling_NaN() ;
```

6.
```
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_i
  }
}
```

Struct template make_scaled_unit

boost::units::make_scaled_unit

Synopsis

```
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

```
template<typename Dimension, typename UnitList, typename OldScale, 
         typename Scale> 
struct make_scaled_unit<unit< Dimension, heterogeneous_system< heterogeneous_system_impl< U
  // types
  typedef unit< Dimension, heterogeneous_system< heterogeneous_system_impl< UnitList, Dimen
};
```

Header <boost/units/make_system.hpp>

```
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

```
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. 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

```
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

```
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

```
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

```
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

```
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

```
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

```
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 for computing the result of raising 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.

`power_typeof_helper` public static functions

1.
```
static type value(const BaseType & base) ;
```

Struct template root_typeof_helper

boost::units::root_typeof_helper

Synopsis

```
template<typename Radicand, typename Index> 
struct root_typeof_helper {
  // types
  typedef unspecified type;  // specifies the result type

  // public static functions
  static type value(const BaseType &) ;
};
```

Description

A helper for computing taking 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.

`root_typeof_helper` public static functions

1.
```
static type value(const BaseType & base) ;
```

Header <boost/units/pow.hpp>

Raise values to exponents known at compile-time.

```
namespace boost {
  namespace units {

    // raise a value to an integer power 
    template<typename Rat, typename Y> 
      power_typeof_helper< Y, Rat >::type pow(const Y & x);

    // take the integer root of a value 
    template<typename Rat, typename Y> 
      root_typeof_helper< Y, Rat >::type root(const Y & x);
  }
}
```

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>;

    // 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

```
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(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 &);
  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(const this_type & source);
```

3.
```
template<typename YY>
  quantity(const quantity< Unit, YY > & source, unspecified = 0);
```

4.
```
template<typename YY>
  quantity(const quantity< Unit, YY > & source, unspecified = 0);
```

5.
```
template<typename Unit2, typename YY>
  quantity(const quantity< Unit2, YY > & source, unspecified = 0);
```

6.
```
template<typename Unit2, typename YY>
  quantity(const quantity< Unit2, YY > & source, unspecified = 0);
```

7.
```
quantity(const value_type & val);
```

8.
```
quantity& operator=(const this_type & source);
```

9.
```
template<typename YY> quantity& operator=(const quantity< Unit, YY > & source);
```

10.
```
template<typename Unit2, typename YY>
  quantity& operator=(const quantity< Unit2, YY > & source);
```

`quantity` private member functions

1.
```
 BOOST_MPL_ASSERT_NOT(unspecified) ;
```

`quantity` public member functions

1.
```
const value_type & value() const;
```

2.
```
template<typename Unit2, typename YY>
  this_type & operator+=(const quantity< Unit2, YY > & source) ;
```

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) ;
```

6.
```
this_type & operator *=(const value_type & source) ;
```

7.
```
this_type & operator/=(const value_type & source) ;
```

`quantity` public static functions

1.
```
static this_type from_value(const value_type & val) ;
```

`value_type`

Specializations

- Class template quantity<BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y>

Class template quantity<BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y>

boost::units::quantity<BOOST_UNITS_DIMENSIONLESS_UNIT(System), Y>

Synopsis

```
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) ;
  template<typename System2, typename Y2>
     quantity(const quantity< unit< dimensionless_type, System2 >, Y2 > &,
              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)
```

3.
```
template<typename System2>
  quantity& operator=(const quantity< BOOST_UNITS_DIMENSIONLESS_UNIT(System2), Y > & sour
```

`quantity` public member functions

1.
```
 quantity() ;
```

2.
```
 quantity(value_type val) ;
```

`value_type`

3.
```
 quantity(const this_type & source) ;
```

4.
```
template<typename YY>
   quantity(const quantity< unit< dimension_type, system_type >, YY > & source,
            unspecified = 0) ;
```

5.
```
template<typename YY>
   quantity(const quantity< unit< dimension_type, system_type >, YY > & source,
            unspecified = 0) ;
```

6.
```
template<typename System2, typename Y2>
   quantity(const quantity< unit< dimensionless_type, System2 >, Y2 > & source,
            unspecified = 0, unspecified = 0) ;
```

7.
```
template<typename System2, typename Y2>
   quantity(const quantity< unit< dimensionless_type, System2 >, Y2 > & source,
            unspecified = 0, unspecified = 0) ;
```

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;
```

`value_type`

10.
```
const value_type & value() const;
```

11.
```
this_type & operator+=(const this_type & source) ;
```

12.
```
this_type & operator-=(const this_type & source) ;
```

13.
```
this_type & operator *=(const value_type & val) ;
```

14.
```
this_type & operator/=(const value_type & val) ;
```

`quantity` public static functions

1.
```
static this_type from_value(const value_type & val) ;
```

`value_type`

Header <boost/units/reduce_unit.hpp>

```
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

```
template<typename Unit>
struct reduce_unit {
  // types
  typedef unspecified type;
};
```

Header <boost/units/scale.hpp>

```
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

```
template<long Base, typename Exponent> 
struct scale {
  // types
  typedef Exponent exponent;  
  typedef double   value_type;

  // public static functions
  static value_type value() ;
  static std::string name() ;
  static std::string symbol() ;
  static const long base;
};
```

Description

class representing a scaling factor such as 10^3 The exponent should be a static rational.

scale public static functions

1.
```
static value_type value() ;
```

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

3.
```
static std::string symbol() ;
```

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

```
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

```
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

```
template<integer_type Value>
struct static_abs {

  // public member functions
   BOOST_STATIC_CONSTANT(integer_type, value) ;
};
```

Description

`static_abs` public member functions

1.
```
BOOST_STATIC_CONSTANT(integer_type, value) ;
```

Class template static_rational

boost::units::static_rational — Compile time rational number.

Synopsis

```
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 

  // construct/copy/destruct
  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/destruct

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 S1, typename S2> struct is_implicitly_convertible;

    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 — (e.g. meters, Kelvin, feet, etc...)

Synopsis

```
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/destruct
  unit();
  unit(const this_type &);
  unit& operator=(const this_type &);

  // private member functions
   BOOST_STATIC_ASSERT(unspecified) ;
   BOOST_STATIC_ASSERT((is_dimension_list< Dim >::value==true)) ;
};
```

Description

unit public construct/copy/destruct

1.
```
unit();
```

2.
```
unit(const this_type &);
```

3.
```
unit& operator=(const this_type &);
```

unit private member functions

1.
```
 BOOST_STATIC_ASSERT(unspecified) ;
```

2.
```
 BOOST_STATIC_ASSERT((is_dimension_list< Dim >::value==true)) ;
```

Struct template reduce_unit<unit< Dim, System >>

boost::units::reduce_unit<unit< Dim, System >> — Returns a unique type for every unit.

Synopsis

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

Struct template is_implicitly_convertible

boost::units::is_implicitly_convertible

Synopsis

```
template<typename S1, typename S2>
struct is_implicitly_convertible {
};
```

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

```
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 stati

  // 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

```
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

  // 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.

Dimensions Reference

Header <boost/units/physical_dimensions/absorbed_dose.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, time_base_dimension,-2 >::type abs
  }
}
```

Header <boost/units/physical_dimensions/acceleration.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 1, time_base_dimension,-2 >::type acc
  }
}
```

Header <boost/units/physical_dimensions/action.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/activity.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< time_base_dimension,-1 >::type activity_dimension;  // deriv
  }
}
```

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
  }
}
```

Struct amount_base_dimension

boost::units::amount_base_dimension — base dimension of amount

Synopsis

```
struct amount_base_dimension {
};
```

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_
  }
}
```

Header <boost/units/physical_dimensions/angular_velocity.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< time_base_dimension,-1, plane_angle_base_dimension, 1 >::typ
  }
}
```

Header <boost/units/physical_dimensions/area.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2 >::type area_dimension;  // derived
  }
}
```

Header <boost/units/physical_dimensions/capacitance.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-2, mass_base_dimension,-1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/conductance.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-2, mass_base_dimension,-1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/conductivity.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-3, mass_base_dimension,-1, time_base_
  }
}
```

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 elec
  }
}
```

Struct current_base_dimension

boost::units::current_base_dimension — base dimension of current

Synopsis

```
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 dos
  }
}
```

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_
  }
}
```

Header <boost/units/physical_dimensions/electric_charge.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< time_base_dimension, 1, current_base_dimension, 1 >::type el
  }
}
```

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_
  }
}
```

Header <boost/units/physical_dimensions/energy.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_
  }
}
```

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_
  }
}
```

Header <boost/units/physical_dimensions/force.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 1, mass_base_dimension, 1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/frequency.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< time_base_dimension,-1 >::type frequency_dimension;  // deri
  }
}
```

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_
  }
}
```

Header <boost/units/physical_dimensions/illuminance.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-2, luminous_intensity_base_dimension,
  }
}
```

Header <boost/units/physical_dimensions/impedance.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/inductance.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/kinematic_viscosity.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, time_base_dimension,-1 >::type kin
  }
}
```

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
  }
}
```

Struct length_base_dimension

boost::units::length_base_dimension — base dimension of length

Synopsis

```
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,
  }
}
```

Header <boost/units/physical_dimensions/luminous_flux.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< luminous_intensity_base_dimension, 1, solid_angle_base_dimen
  }
}
```

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;
  }
}
```

Struct luminous_intensity_base_dimension

boost::units::luminous_intensity_base_dimension — base dimension of luminous intensity

Synopsis

```
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 
  }
}
```

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_
  }
}
```

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_
  }
}
```

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

```
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 mas
  }
}
```

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_
  }
}
```

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_
  }
}
```

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_angl
  }
}
```

Header <boost/units/physical_dimensions/momentum.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 1, mass_base_dimension, 1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/permeability.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 1, mass_base_dimension, 1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/permittivity.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-3, mass_base_dimension,-1, time_base_
  }
}
```

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 dime
  }
}
```

Struct plane_angle_base_dimension

boost::units::plane_angle_base_dimension — base dimension of plane angle

Synopsis

```
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_
  }
}
```

Header <boost/units/physical_dimensions/pressure.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-1, mass_base_dimension, 1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/reluctance.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-2, mass_base_dimension,-1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/resistance.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, mass_base_dimension, 1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/resistivity.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 3, mass_base_dimension, 1, time_base_
  }
}
```

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 dime
  }
}
```

Struct solid_angle_base_dimension

boost::units::solid_angle_base_dimension — base dimension of solid angle

Synopsis

```
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 spe
  }
}
```

Header <boost/units/physical_dimensions/specific_heat_capacity.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 2, time_base_dimension,-2, temperatur
  }
}
```

Header <boost/units/physical_dimensions/specific_volume.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 3, mass_base_dimension,-1 >::type spe
  }
}
```

Header <boost/units/physical_dimensions/stress.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-1, mass_base_dimension, 1, time_base_
  }
}
```

Header <boost/units/physical_dimensions/surface_density.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-2, mass_base_dimension, 1 >::type sur
  }
}
```

Header <boost/units/physical_dimensions/surface_tension.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< mass_base_dimension, 1, time_base_dimension,-2 >::type surfa
  }
}
```

Header <boost/units/physical_dimensions/temperature.hpp>

```
namespace boost {
  namespace units {
    struct temperature_base_dimension;

    typedef temperature_base_dimension::dimension_type temperature_dimension;  // dimension
  }
}
```

Struct temperature_base_dimension

boost::units::temperature_base_dimension — base dimension of temperature

Synopsis

```
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_
  }
}
```

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

```
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_
  }
}
```

Header <boost/units/physical_dimensions/velocity.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 1, time_base_dimension,-1 >::type vel
  }
}
```

Header <boost/units/physical_dimensions/volume.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension, 3 >::type volume_dimension;  // deriv
  }
}
```

Header <boost/units/physical_dimensions/wavenumber.hpp>

```
namespace boost {
  namespace units {
    typedef derived_dimension< length_base_dimension,-1 >::type wavenumber_dimension;  // d
  }
}
```

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

```
static const absorbed_dose gray;
```

Global grays

boost::units::si::grays

Synopsis

```
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

```
static const acceleration meter_per_second_squared;
```

Global meters_per_second_squared

boost::units::si::meters_per_second_squared

Synopsis

```
static const acceleration meters_per_second_squared;
```

Global metre_per_second_squared

boost::units::si::metre_per_second_squared

Synopsis

```
static const acceleration metre_per_second_squared;
```

Global metres_per_second_squared

boost::units::si::metres_per_second_squared

Synopsis

```
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

```
static const activity becquerel;
```

Global becquerels

boost::units::si::becquerels

Synopsis

```
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

```
static const amount mole;
```

Global moles

boost::units::si::moles

Synopsis

```
static const amount moles;
```

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

```
static const angular_velocity radian_per_second;
```

Global radians_per_second

boost::units::si::radians_per_second

Synopsis

```
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

```
static const area square_meter;
```

Global square_meters

boost::units::si::square_meters

Synopsis

```
static const area square_meters;
```

Global square_metre

boost::units::si::square_metre

Synopsis

```
static const area square_metre;
```

Global square_metres

boost::units::si::square_metres

Synopsis

```
static const area square_metres;
```

Header <boost/units/systems/si/base.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      typedef make_system< meter_base_unit, kilogram_base_unit, second_base_unit, ampere_ba
      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

```
static const capacitance farad;
```

Global farads

boost::units::si::farads

Synopsis

```
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 c
      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

```
static const catalytic_activity katal;
```

Global katals

boost::units::si::katals

Synopsis

```
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 {

          // alpha particle mass
           BOOST_UNITS_PHYSICAL_CONSTANT(m_alpha, quantity< mass >,
                                         6.64465620e-27 * kilograms,
                                         3.3e-34 * kilograms);

          // 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);
        }
      }
    }
  }
}
```

Header <boost/units/systems/si/codata/atomic-nuclear_constants.hpp>

```
namespace boost {
  namespace units {
    namespace si {
      namespace constants {
        namespace codata {

          // fine structure constant
           BOOST_UNITS_PHYSICAL_CONSTANT(alpha, quantity< dimensionless >,
                                         7.2973525376e-3 * dimensionless,
                                         5.0e-12 * dimensionless);

          // 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);
        }
      }
    }
  }
}
```

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 {

          // deuteron mass
           BOOST_UNITS_PHYSICAL_CONSTANT(m_d, quantity< mass >,
                                         3.34358320e-27 * kilograms,
                                         1.7e-34 * kilograms);

          // 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);
        }
      }
    }
  }
}
```

Header <boost/units/systems/si/codata/electromagnetic_constants.hpp>

CODATA recommended values of fundamental electromagnetic constants 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 {

          // electron mass
           BOOST_UNITS_PHYSICAL_CONSTANT(m_e, quantity< mass >,
                                         9.10938215e-31 * kilograms,
                                         4.5e-38 * kilograms);

          // 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 magenton 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);
        }
      }
    }
  }
}
```

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 {

          // helion mass
           BOOST_UNITS_PHYSICAL_CONSTANT(m_h, quantity< mass >,
                                         5.00641192e-27 * kilograms,
                                         2.5e-34 * kilograms);

          // 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);
          }
        }
      }
    }
}
```

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 {

          // muon mass
           BOOST_UNITS_PHYSICAL_CONSTANT(m_mu, quantity< mass >,
                                         1.88353130e-28 * kilograms,
                                         1.1e-35 * kilograms);

          // 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);
        }
      }
    }
  }
}
```

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 {

          // neutron mass
           BOOST_UNITS_PHYSICAL_CONSTANT(m_n, quantity< mass >,
                                         1.674927211e-27 * kilograms,
                                         8.4e-35 * kilograms);

          // 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);
        }
      }
    }
  }
}
```

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 >,
                                          5.878933e10 *hertz/ kelvin,
                                          1.0e15 *hertz/ kelvin);
        }
      }
    }
  }
}
```

Header <boost/units/systems/si/codata/proton_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 {

          // proton mass
           BOOST_UNITS_PHYSICAL_CONSTANT(m_p, quantity< mass >,
                                         1.672621637e-27 * kilograms,
                                         8.3e-35 * kilograms);

          // 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);
        }
      }
    }
  }
}
```

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 {

          // tau mass
           BOOST_UNITS_PHYSICAL_CONSTANT(m_tau, quantity< mass >,
                                         3.16777e-27 * kilograms,
                                         5.2e-31 * kilograms);

          // 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);
        }
      }
    }
  }
}
```

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 {

          // triton mass
           BOOST_UNITS_PHYSICAL_CONSTANT(m_t, quantity< mass >,
                                         5.00735588e-27 * kilograms,
                                         2.5e-34 * kilograms);

          // 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);
        }
      }
    }
  }
}
```

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_ove
          typedef divide_typeof_helper< electric_charge, mass >::type electric_charge_over_
          typedef divide_typeof_helper< mass, amount >::type mass_over_amount;
          typedef divide_typeof_helper< energy, magnetic_flux_density >::type energy_over_m
          typedef divide_typeof_helper< frequency, magnetic_flux_density >::type frequency_
          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
          typedef divide_typeof_helper< divide_typeof_helper< power, area >::type, power_ty
          typedef multiply_typeof_helper< power, area >::type power_area;
          typedef divide_typeof_helper< power_area, solid_angle >::type power_area_over_sol
          typedef multiply_typeof_helper< length, temperature >::type length_temperature;
          typedef divide_typeof_helper< frequency, temperature >::type frequency_over_tempe
          typedef divide_typeof_helper< divide_typeof_helper< force, current >::type, curre
          typedef divide_typeof_helper< capacitance, length >::type capacitance_over_length
          typedef divide_typeof_helper< divide_typeof_helper< divide_typeof_helper< volume,
          typedef multiply_typeof_helper< energy, time >::type energy_time;
          typedef divide_typeof_helper< electric_charge, amount >::type electric_charge_ove
        }
      }
    }
  }
}
```

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 {

          // speed of light
           BOOST_UNITS_PHYSICAL_CONSTANT(c, quantity< velocity >,
                                         299792458.0 *meters/ second,
                                         0.0 *meters/ second);

          // magnetic constant (exactly 4 pi x 10^(-7) - error is due to finite precision o
           BOOST_UNITS_PHYSICAL_CONSTANT(mu_0,
                                         quantity< force_over_current_squared >,
                                         12.56637061435917295385057353311801153679e-7 *newt
                                         0.0 *newtons/ampere/ ampere);

          // electric constant
           BOOST_UNITS_PHYSICAL_CONSTANT(epsilon_0,
                                         quantity< capacitance_over_length >,
                                         8.854187817620389850536563031710750260608e-12 *far
                                         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);
        }
      }
    }
  }
}
```

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

```
static const conductance siemen;
```

Global siemens

boost::units::si::siemens

Synopsis

```
static const conductance siemens;
```

Global mho

boost::units::si::mho

Synopsis

```
static const conductance mho;
```

Global mhos

boost::units::si::mhos

Synopsis

```
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

```
static const current ampere;
```

Global amperes

boost::units::si::amperes

Synopsis

```
static const current amperes;
```

Header <boost/units/systems/si/dimensionless.hpp>

Global si_dimensionless

boost::units::si::si_dimensionless

Synopsis

```
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

```
static const dose_equivalent sievert;
```

Global sieverts

boost::units::si::sieverts

Synopsis

```
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

```
static const electric_charge coulomb;
```

Global coulombs

boost::units::si::coulombs

Synopsis

```
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

```
static const electric_potential volt;
```

Global volts

boost::units::si::volts

Synopsis

```
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

```
static const energy joule;
```

Global joules

boost::units::si::joules

Synopsis

```
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

```
static const force newton;
```

Global newtons

boost::units::si::newtons

Synopsis

```
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

```
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

```
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

```
static const inductance henry;
```

Global henrys

boost::units::si::henrys

Synopsis

```
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

```
static const length meter;
```

Global meters

boost::units::si::meters

Synopsis

```
static const length meters;
```

Global metre

boost::units::si::metre

Synopsis

```
static const length metre;
```

Global metres

boost::units::si::metres

Synopsis

```
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

```
static const luminous_flux lumen;
```

Global lumens

boost::units::si::lumens

Synopsis

```
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

```
static const luminous_intensity candela;
```

Global candelas

boost::units::si::candelas

Synopsis

```
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_intensi
    }
  }
}
```

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

```
static const magnetic_flux weber;
```

Global webers

boost::units::si::webers

Synopsis

```
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

```
static const magnetic_flux_density tesla;
```

Global teslas

boost::units::si::teslas

Synopsis

```
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

```
static const mass kilogram;
```

Global kilograms

boost::units::si::kilograms

Synopsis

```
static const mass kilograms;
```

Global kilogramme

boost::units::si::kilogramme

Synopsis

```
static const mass kilogramme;
```

Global kilogrammes

boost::units::si::kilogrammes

Synopsis

```
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

```
static const mass_density kilogram_per_cubic_meter;
```

Global kilograms_per_cubic_meter

boost::units::si::kilograms_per_cubic_meter

Synopsis

```
static const mass_density kilograms_per_cubic_meter;
```

Global kilogramme_per_cubic_metre

boost::units::si::kilogramme_per_cubic_metre

Synopsis

```
static const mass_density kilogramme_per_cubic_metre;
```

Global kilogrammes_per_cubic_metre

boost::units::si::kilogrammes_per_cubic_metre

Synopsis

```
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

```
static const plane_angle radian;
```

Global radians

boost::units::si::radians

Synopsis

```
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

```
static const power watt;
```

Global watts

boost::units::si::watts

Synopsis

```
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

```
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

```
static const pressure pascal;
```

Global pascals

boost::units::si::pascals

Synopsis

```
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

```
static const resistance ohm;
```

Global ohms

boost::units::si::ohms

Synopsis

```
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

```
static const solid_angle steradian;
```

Global steradians

boost::units::si::steradians

Synopsis

```
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

```
static const surface_density kilogram_per_square_meter;
```

Global kilograms_per_square_meter

boost::units::si::kilograms_per_square_meter

Synopsis

```
static const surface_density kilograms_per_square_meter;
```

Global kilogramme_per_square_metre

boost::units::si::kilogramme_per_square_metre

Synopsis

```
static const surface_density kilogramme_per_square_metre;
```

Global kilogrammes_per_square_metre

boost::units::si::kilogrammes_per_square_metre

Synopsis

```
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

```
static const surface_tension newton_per_meter;
```

Global newtons_per_meter

boost::units::si::newtons_per_meter

Synopsis

```
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

```
static const temperature kelvin;
```

Global kelvins

boost::units::si::kelvins

Synopsis

```
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

```
static const time second;
```

Global seconds

boost::units::si::seconds

Synopsis

```
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

```
static const torque newton_meter;
```

Global newton_meters

boost::units::si::newton_meters

Synopsis

```
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

```
static const velocity meter_per_second;
```

Global meters_per_second

boost::units::si::meters_per_second

Synopsis

```
static const velocity meters_per_second;
```

Global metre_per_second

boost::units::si::metre_per_second

Synopsis

```
static const velocity metre_per_second;
```

Global metres_per_second

boost::units::si::metres_per_second

Synopsis

```
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

```
static const volume cubic_meter;
```

Global cubic_meters

boost::units::si::cubic_meters

Synopsis

```
static const volume cubic_meters;
```

Global cubic_metre

boost::units::si::cubic_metre

Synopsis

```
static const volume cubic_metre;
```

Global cubic_metres

boost::units::si::cubic_metres

Synopsis

```
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

```
static const wavenumber reciprocal_meter;
```

Global reciprocal_meters

boost::units::si::reciprocal_meters

Synopsis

```
static const wavenumber reciprocal_meters;
```

Global reciprocal_metre

boost::units::si::reciprocal_metre

Synopsis

```
static const wavenumber reciprocal_metre;
```

Global reciprocal_metres

boost::units::si::reciprocal_metres

Synopsis

```
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

```
static const acceleration gal;
```

Global gals

boost::units::cgs::gals

Synopsis

```
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

```
static const area square_centimeter;
```

Global square_centimeters

boost::units::cgs::square_centimeters

Synopsis

```
static const area square_centimeters;
```

Global square_centimetre

boost::units::cgs::square_centimetre

Synopsis

```
static const area square_centimetre;
```

Global square_centimetres

boost::units::cgs::square_centimetres

Synopsis

```
static const area square_centimetres;
```

Header <boost/units/systems/cgs/base.hpp>

```
namespace boost {
  namespace units {
    namespace cgs {
      typedef make_system< centimeter_base_unit, gram_base_unit, boost::units::si::second_b
      typedef unit< dimensionless_type, system > dimensionless;  // various unit typedefs f
    }
  }
}
```

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

```
static const current biot;
```

Global biots

boost::units::cgs::biots

Synopsis

```
static const current biots;
```

Header <boost/units/systems/cgs/dimensionless.hpp>

Global cgs_dimensionless

boost::units::cgs::cgs_dimensionless

Synopsis

```
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

```
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

```
static const energy erg;
```

Global ergs

boost::units::cgs::ergs

Synopsis

```
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

```
static const force dyne;
```

Global dynes

boost::units::cgs::dynes

Synopsis

```
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

```
static const kinematic_viscosity stoke;
```

Global stokes

boost::units::cgs::stokes

Synopsis

```
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

```
static const length centimeter;
```

Global centimeters

boost::units::cgs::centimeters

Synopsis

```
static const length centimeters;
```

Global centimetre

boost::units::cgs::centimetre

Synopsis

```
static const length centimetre;
```

Global centimetres

boost::units::cgs::centimetres

Synopsis

```
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

```
static const mass gram;
```

Global grams

boost::units::cgs::grams

Synopsis

```
static const mass grams;
```

Global gramme

boost::units::cgs::gramme

Synopsis

```
static const mass gramme;
```

Global grammes

boost::units::cgs::grammes

Synopsis

```
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

```
static const pressure barye;
```

Global baryes

boost::units::cgs::baryes

Synopsis

```
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

```
static const time second;
```

Global seconds

boost::units::cgs::seconds

Synopsis

```
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

```
static const velocity centimeter_per_second;
```

Global centimeters_per_second

boost::units::cgs::centimeters_per_second

Synopsis

```
static const velocity centimeters_per_second;
```

Global centimetre_per_second

boost::units::cgs::centimetre_per_second

Synopsis

```
static const velocity centimetre_per_second;
```

Global centimetres_per_second

boost::units::cgs::centimetres_per_second

Synopsis

```
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

```
static const volume cubic_centimeter;
```

Global cubic_centimeters

boost::units::cgs::cubic_centimeters

Synopsis

```
static const volume cubic_centimeters;
```

Global cubic_centimetre

boost::units::cgs::cubic_centimetre

Synopsis

```
static const volume cubic_centimetre;
```

Global cubic_centimetres

boost::units::cgs::cubic_centimetres

Synopsis

```
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 kaysers;
      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

```
static const wavenumber kayser;
```

Global kaysers

boost::units::cgs::kaysers

Synopsis

```
static const wavenumber kaysers;
```

Global reciprocal_centimeter

boost::units::cgs::reciprocal_centimeter

Synopsis

```
static const wavenumber reciprocal_centimeter;
```

Global reciprocal_centimeters

boost::units::cgs::reciprocal_centimeters

Synopsis

```
static const wavenumber reciprocal_centimeters;
```

Global reciprocal_centimetre

boost::units::cgs::reciprocal_centimetre

Synopsis

```
static const wavenumber reciprocal_centimetre;
```

Global reciprocal_centimetres

boost::units::cgs::reciprocal_centimetres

Synopsis

```
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 cons

      static const plane_angle degree;
      static const plane_angle degrees;
    }
  }
}
```

Global degree

boost::units::degree::degree

Synopsis

```
static const plane_angle degree;
```

Global degrees

boost::units::degree::degrees

Synopsis

```
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 con

      static const plane_angle gradian;
      static const plane_angle gradians;
    }
  }
}
```

Global gradian

boost::units::gradian::gradian

Synopsis

```
static const plane_angle gradian;
```

Global gradians

boost::units::gradian::gradians

Synopsis

```
static const plane_angle gradians;
```

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 c

      static const plane_angle revolution;
      static const plane_angle revolutions;
    }
  }
}
```

Global revolution

boost::units::revolution::revolution

Synopsis

```
static const plane_angle revolution;
```

Global revolutions

boost::units::revolution::revolutions

Synopsis

```
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

```
static const temperature degree;
```

Global degrees

boost::units::celsius::degrees

Synopsis

```
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

```
static const temperature degree;
```

Global degrees

boost::units::fahrenheit::degrees

Synopsis

```
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,
      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 tempe
      typedef unit< amount_dimension, system > amount;  // abstract unit of amount
      typedef unit< luminous_intensity_dimension, system > luminous_intensity;  // abstract
      typedef unit< plane_angle_dimension, system > plane_angle;  // abstract unit of plane
      typedef unit< solid_angle_dimension, system > solid_angle;  // abstract unit of solid
    }
  }
}
```

Struct length_unit_tag

boost::units::abstract::length_unit_tag

Synopsis

```
struct length_unit_tag {
};
```

Struct mass_unit_tag

boost::units::abstract::mass_unit_tag

Synopsis

```
struct mass_unit_tag {
};
```

Struct time_unit_tag

boost::units::abstract::time_unit_tag

Synopsis

```
struct time_unit_tag {
};
```

Struct current_unit_tag

boost::units::abstract::current_unit_tag

Synopsis

```
struct current_unit_tag {
};
```

Struct temperature_unit_tag

boost::units::abstract::temperature_unit_tag

Synopsis

```
struct temperature_unit_tag {
};
```

Struct amount_unit_tag

boost::units::abstract::amount_unit_tag

Synopsis

```
struct amount_unit_tag {
};
```

Struct luminous_intensity_unit_tag

boost::units::abstract::luminous_intensity_unit_tag

Synopsis

```
struct luminous_intensity_unit_tag {
};
```

Struct plane_angle_unit_tag

boost::units::abstract::plane_angle_unit_tag

Synopsis

```
struct plane_angle_unit_tag {
};
```

Struct solid_angle_unit_tag

boost::units::abstract::solid_angle_unit_tag

Synopsis

```
struct solid_angle_unit_tag {
};
```

Struct base_unit_info<abstract::length_unit_tag>

boost::units::base_unit_info<abstract::length_unit_tag>

Synopsis

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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 > > > arcmi
    }
  }
}
```

Struct base_unit_info<angle::arcminute_base_unit>

boost::units::base_unit_info<angle::arcminute_base_unit>

Synopsis

```
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 > > > arc
    }
  }
}
```

Struct base_unit_info<angle::arcsecond_base_unit>

boost::units::base_unit_info<angle::arcsecond_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(angle, degree, "degree", "deg",
                                              6.28318530718/ 360.,
                                              boost::units::angle::radian_base_unit,
                                              - 101);
```

Header <boost/units/base_units/angle/gradian.hpp>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(angle, gradian, "gradian",
                                              "grad", 6.28318530718/ 400.,
                                              boost::units::angle::radian_base_unit,
                                              - 102);
```

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

```
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 > > > revo
    }
  }
}
```

Struct base_unit_info<angle::revolution_base_unit>

boost::units::base_unit_info<angle::revolution_base_unit>

Synopsis

```
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

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(astronomical,
                                              astronomical_unit,
                                              "astronomical unit", "a.u.",
                                              149597870691. 0,
                                              boost::units::si::meter_base_unit,
                                              - 207);
```

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<
    }
  }
}
```

Struct base_unit_info<astronomical::light_day_base_unit>

boost::units::base_unit_info<astronomical::light_day_base_unit>

Synopsis

```
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::astronomical::light_second_base_unit, scale<
    }
  }
}
```

Struct base_unit_info<astronomical::light_hour_base_unit>

boost::units::base_unit_info<astronomical::light_hour_base_unit>

Synopsis

```
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<
    }
  }
}
```

Struct base_unit_info<astronomical::light_minute_base_unit>

boost::units::base_unit_info<astronomical::light_minute_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(astronomical, light_second,
                                              "light second", "lsc",
                                              2. 99792458e8,
                                              boost::units::si::meter_base_unit,
                                              - 201);
```

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::astronomical::light_second_base_unit, scale<
    }
  }
}
```

Struct base_unit_info<astronomical::light_year_base_unit>

boost::units::base_unit_info<astronomical::light_year_base_unit>

Synopsis

```
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>

```

 BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(astronomical, parsec, "parsec",
                                               "psc", 3. 0856775813e16,
                                               boost::units::si::meter_base_unit,
                                               - 206);
```

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_ratio
    }
  }
}
```

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_ration
    }
  }
}
```

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

```
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_rational<-2 > > > drachm
    }
  }
}
```

Struct base_unit_info<imperial::drachm_base_unit>

boost::units::base_unit_info<imperial::drachm_base_unit>

Synopsis

```
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_o
    }
  }
}
```

Struct base_unit_info<imperial::fluid_ounce_base_unit>

boost::units::base_unit_info<imperial::fluid_ounce_base_unit>

Synopsis

```
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_bas
    }
  }
}
```

Struct base_unit_info<imperial::foot_base_unit>

boost::units::base_unit_info<imperial::foot_base_unit>

Synopsis

```
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 > > > furlon
    }
  }
}
```

Struct base_unit_info<imperial::furlong_base_unit>

boost::units::base_unit_info<imperial::furlong_base_unit>

Synopsis

```
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 > > > gallon_b
    }
  }
}
```

Struct base_unit_info<imperial::gallon_base_unit>

boost::units::base_unit_info<imperial::gallon_base_unit>

Synopsis

```
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_bas
    }
  }
}
```

Struct base_unit_info<imperial::gill_base_unit>

boost::units::base_unit_info<imperial::gill_base_unit>

Synopsis

```
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_rational<-1 > > > grai
    }
  }
}
```

Struct base_unit_info<imperial::grain_base_unit>

boost::units::base_unit_info<imperial::grain_base_unit>

Synopsis

```
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 > > > hundr
    }
  }
}
```

Struct base_unit_info<imperial::hundredweight_base_unit>

boost::units::base_unit_info<imperial::hundredweight_base_unit>

Synopsis

```
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_ba
    }
  }
}
```

Struct base_unit_info<imperial::inch_base_unit>

boost::units::base_unit_info<imperial::inch_base_unit>

Synopsis

```
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_rational< 1 > > > leagu
    }
  }
}
```

Struct base_unit_info<imperial::league_base_unit>

boost::units::base_unit_info<imperial::league_base_unit>

Synopsis

```
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_rational< 1 > > > mile_
    }
  }
}
```

Struct base_unit_info<imperial::mile_base_unit>

boost::units::base_unit_info<imperial::mile_base_unit>

Synopsis

```
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_rational<-4 > > > ounce_b
    }
  }
}
```

Struct base_unit_info<imperial::ounce_base_unit>

boost::units::base_unit_info<imperial::ounce_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(imperial, pint, "pint (imp.)",
                                              "pt", 4.54609e-3/ 8.,
                                              si::volume, - 303);
```

Header <boost/units/base_units/imperial/pound.hpp>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(imperial, pound, "pound", "lb",
                                              453. 59237,
                                              cgs::gram_base_unit, - 302);
```

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_ba
    }
  }
}
```

Struct base_unit_info<imperial::quart_base_unit>

boost::units::base_unit_info<imperial::quart_base_unit>

Synopsis

```
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_rational< 1 > > > quarte
    }
  }
}
```

Struct base_unit_info<imperial::quarter_base_unit>

boost::units::base_unit_info<imperial::quarter_base_unit>

Synopsis

```
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_rational< 1 > > > stone_
    }
  }
}
```

Struct base_unit_info<imperial::stone_base_unit>

boost::units::base_unit_info<imperial::stone_base_unit>

Synopsis

```
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_rational<-1 > > > thou_
    }
  }
}
```

Struct base_unit_info<imperial::thou_base_unit>

boost::units::base_unit_info<imperial::thou_base_unit>

Synopsis

```
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_rational< 1 > > > ton_
    }
  }
}
```

Struct base_unit_info<imperial::ton_base_unit>

boost::units::base_unit_info<imperial::ton_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(imperial, yard, "yard", "yd",
                                              0. 9144, si::meter_base_unit,
                                              - 301);
```

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_ration
    }
  }
}
```

Struct base_unit_info<metric::angstrom_base_unit>

boost::units::base_unit_info<metric::angstrom_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(metric, are, "are", "a",
                                              1. 0e2, si::area, 10);
```

Header <boost/units/base_units/metric/atmosphere.hpp>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(metric, atmosphere,
                                              "atmosphere", "atm",
                                              1. 01325e5, si::pressure, 33);
```

Header <boost/units/base_units/metric/bar.hpp>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(metric, bar, "bar", "bar",
                                              1. 0e5, si::pressure, 14);
```

Header <boost/units/base_units/metric/barn.hpp>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(metric, barn, "barn", "b",
                                              1.0e- 28, si::area, 11);
```

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_ra
    }
  }
}
```

Struct base_unit_info<metric::day_base_unit>

boost::units::base_unit_info<metric::day_base_unit>

Synopsis

```
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_ration
    }
  }
}
```

Struct base_unit_info<metric::fermi_base_unit>

boost::units::base_unit_info<metric::fermi_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(metric, hectare, "hectare",
                                              "ha", 1. 0e4, si::area, 12);
```

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_ratio
    }
  }
}
```

Struct base_unit_info<metric::hour_base_unit>

boost::units::base_unit_info<metric::hour_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(metric, knot, "knot", "kt",
                                              1852./ 3600.,
                                              boost::units::si::velocity,
                                              - 403);
```

Header <boost/units/base_units/metric/liter.hpp>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(metric, liter, "liter", "L",
                                              1.0e- 3, si::volume, 13);
```

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_ration
    }
  }
}
```

Struct base_unit_info<metric::micron_base_unit>

boost::units::base_unit_info<metric::micron_base_unit>

Synopsis

```
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
    }
  }
}
```

Struct base_unit_info<metric::minute_base_unit>

boost::units::base_unit_info<metric::minute_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(metric, mmHg,
                                              "millimeters mercury", "mmHg",
                                              133. 322, si::pressure, - 404);
```

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_rati
    }
  }
}
```

Struct base_unit_info<metric::nautical_mile_base_unit>

boost::units::base_unit_info<metric::nautical_mile_base_unit>

Synopsis

```
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_r
    }
  }
}
```

Struct base_unit_info<metric::ton_base_unit>

boost::units::base_unit_info<metric::ton_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(metric, torr, "torr", "Torr",
                                              1.01325e5/760. 0,
                                              si::pressure, - 401);
```

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
    }
  }
}
```

Struct base_unit_info<metric::year_base_unit>

boost::units::base_unit_info<metric::year_base_unit>

Synopsis

```
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

```
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

```
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

```
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
    }
  }
}
```

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

```
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

```
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

```
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

```
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

```
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_
    }
  }
}
```

Struct base_unit_info<us::cup_base_unit>

boost::units::base_unit_info<us::cup_base_unit>

Synopsis

```
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_rational<-2 > > > dram_b
    }
  }
}
```

Struct base_unit_info<us::dram_base_unit>

boost::units::base_unit_info<us::dram_base_unit>

Synopsis

```
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_dr
    }
  }
}
```

Struct base_unit_info<us::fluid_dram_base_unit>

boost::units::base_unit_info<us::fluid_dram_base_unit>

Synopsis

```
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_o
    }
  }
}
```

Struct base_unit_info<us::fluid_ounce_base_unit>

boost::units::base_unit_info<us::fluid_ounce_base_unit>

Synopsis

```
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_bas
    }
  }
}
```

Struct base_unit_info<us::foot_base_unit>

boost::units::base_unit_info<us::foot_base_unit>

Synopsis

```
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 > > > gallon_b
    }
  }
}
```

Struct base_unit_info<us::gallon_base_unit>

boost::units::base_unit_info<us::gallon_base_unit>

Synopsis

```
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_bas
    }
  }
}
```

Struct base_unit_info<us::gill_base_unit>

boost::units::base_unit_info<us::gill_base_unit>

Synopsis

```
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_rational<-1 > > > grai
    }
  }
}
```

Struct base_unit_info<us::grain_base_unit>

boost::units::base_unit_info<us::grain_base_unit>

Synopsis

```
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 > > > hundr
    }
  }
}
```

Struct base_unit_info<us::hundredweight_base_unit>

boost::units::base_unit_info<us::hundredweight_base_unit>

Synopsis

```
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_ba
    }
  }
}
```

Struct base_unit_info<us::inch_base_unit>

boost::units::base_unit_info<us::inch_base_unit>

Synopsis

```
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_rational<-1 > > > mil_
    }
  }
}
```

Struct base_unit_info<us::mil_base_unit>

boost::units::base_unit_info<us::mil_base_unit>

Synopsis

```
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_rational< 1 > > > mile_
    }
  }
}
```

Struct base_unit_info<us::mile_base_unit>

boost::units::base_unit_info<us::mile_base_unit>

Synopsis

```
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_
    }
  }
}
```

Struct base_unit_info<us::minim_base_unit>

boost::units::base_unit_info<us::minim_base_unit>

Synopsis

```
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_rational<-4 > > > ounce_b
    }
  }
}
```

Struct base_unit_info<us::ounce_base_unit>

boost::units::base_unit_info<us::ounce_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(us, pint, "pint (U.S.)", "pt",
                                              0.4731765e- 3, si::volume,
                                              - 503);
```

Header <boost/units/base_units/us/pound.hpp>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(us, pound, "pound", "lb",
                                              453. 59237,
                                              cgs::gram_base_unit, - 502);
```

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_ba
    }
  }
}
```

Struct base_unit_info<us::quart_base_unit>

boost::units::base_unit_info<us::quart_base_unit>

Synopsis

```
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 > > > tablespo
    }
  }
}
```

Struct base_unit_info<us::tablespoon_base_unit>

boost::units::base_unit_info<us::tablespoon_base_unit>

Synopsis

```
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 > > > teaspoo
    }
  }
}
```

Struct base_unit_info<us::teaspoon_base_unit>

boost::units::base_unit_info<us::teaspoon_base_unit>

Synopsis

```
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_rational< 1 > > > ton_
    }
  }
}
```

Struct base_unit_info<us::ton_base_unit>

boost::units::base_unit_info<us::ton_base_unit>

Synopsis

```
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>

```
BOOST_UNITS_DEFINE_BASE_UNIT_WITH_CONVERSIONS(us, yard, "yard", "yd",
                                              0. 9144, si::meter_base_unit,
                                              - 501);
```

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)
gradian
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)
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; 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)?

In cases such as this, the proper way to treat this difference is to recognize that the underlying value types are distinct. For the particular case of energy vs. torque, energy is a true scalar quantity, while torque, despite having the same units as energy, is in fact a pseudovector. Thus, to properly treat torque quantities, a value type representing pseudovectors and encapsulating their algebra would have to be implemented. Then, one would write something like this:

```
quantity<energy,double>           E;
quantity<energy,pseudovector>     tau;
```

naturally, a typedef for torque could also be added to make the intent more transparent.

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 propogate 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 can be handled by the default constructor.

Why are conversions explicit by default?

Safety. Implicit conversions are dangerous and should not occur without a good reason.

Acknowledgements

Thanks to David Walthall for his assistance in debugging and testing on a variety of platforms.

Thanks to:

- Paul Bristow,
- Michael Fawcett,
- Ben FrantzDale,
- Ron Garcia,
- David Greene,
- Peder Holt,
- Janek Kozicki,
- Andy Little,
- Kevin Lynch,
- 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 compilers other than gcc 4.0.1 under Mac OSX, and MSVC 8.0, Metrowerks CodeWarrior 9.2, MSVC 7.1, and gcc 3.4.4 under Windows
- performance testing on various architectures
- tutorials on getting started and implementing new unit systems

Release Notes

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/trig.hpp` - implicit conversion of radians between trigonometric, SI, and CGS systems 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>`