Get your `lambda<T>`

How do I get a cool shirt like Marshall?

`http://cierecloud.com/cppnow/`

ciere.com

Introduction to Modern C++ Techniques

Professional C++ Training

ciere consulting

Michael Caisse

http://ciere.com/cppnow12
follow @consultciere

ciere.com

Introduction to Modern C++ Techniques

Professional C++ Training

ciere consulting

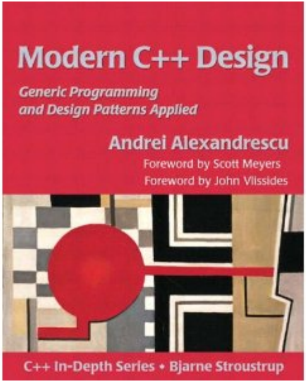
Michael Caisse

http://ciere.com/cppnow12
follow @consultciere

Alternate Title

The session I wish I had on my first BoostCon visit

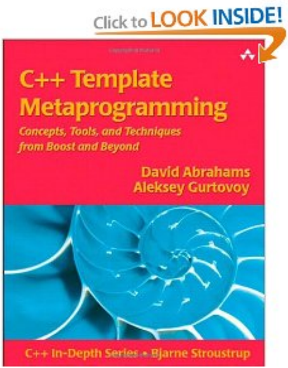
Books



Modern C++ Design
Generic Programming and Design Patterns Applied

Andrei Alexandrescu

Books



C++ Template
Metaprogramming
Concepts, Tools, and Techniques from
Boost and Beyond

David Abrahams
Aleksey Gurtovoy

People / Libraries

- Joel de Guzman and Boost.Fusion
- Hartmut Kaiser and Boost.Spirit - Karma
- Barend Gehrels and Boost.Geometry
- Steven Watanabe and Boost.Units

Part I

Modern C++ Overview

Outline

ciere.com

Functor

A functor is a *function object*
or
It is an object that we can treat as if it is callable

ciere.com

Functor

Why would we do this?

- Objects can contain state

Functor

There are a slew of algorithms that want something that is Callable.

How do we make an object callable?

Functor

```
struct mod
{
  mod(int m) : m_(m) {}
  int operator()(int i){ return i%m; }

  int m_;
};
```

```
mod f(4);
cout << f(42) << endl;
```

Output

2

ciere.com

Functor

```
struct mod
{
  mod(int m) : m_(m) {}
  int operator()(int i){ return i%m; }

  int m_;
};
```

```
mod f(4);
cout << f(42) << endl;
```

Output

2

Outline

RAII

Resource Aquisition Is Initialization

- Invented by Stroustrup to handle resource deallocation
- Constructor / Destructor
- Essential for writing exception-safe code

RAII - Example

```
void foo()
{
  boost::mutex::scoped_lock lock( queue_mutex );
  if( !my_queue.empty() )
  {
    bar( my_queue.front() );
    my_queue.pop_front();
  }
}
```

RAII - Exercise

```
void amazingly_bad_function()
{
  special_resource.get();
  special_resource.update( some_call() );
  special_resource.release();
}
```

RAII - Exercise

```
template< typename T >
struct raii_helper : boost::noncopyable
{
    raii_helper(T& v) : v_(v) { v_.get(); }
    ~raii_helper() { v_.release(); }

private:
    T& v_;
};


void better_function()
{
    raii_helper<special_t> lock(special_resource);
    special_resource.update( some_call() );
}
```

Outline

2 Overview

3 Getting Started
- Functor
- RAII
- Concepts

4 Moving Along
- Policy Classes
- CRTP
- Type Traits
- Tag Dispatching
- Substitution Failuer is not an Error

5 More Meta Programming
- Units

Concepts

How will users know how to use our helper?

```
template< typename T >
struct raii_helper
{
  raii_helper(T & v);
};
```

Concepts

- Provide the set of requirements to be met by a type
- Types meeting the requirements *model* the *concept*
- *Concepts* can extend their requirements which is called *refinement*
- Typically *spelled* with CamelCase

ciere.com

Concepts

- Provide the set of requirements to be met by a type
- Types meeting the requirements *model* the *concept*
- *Concepts* can extend their requirements which is called *refinement*
- Typically *spelled* with CamelCase

ciere.com

Concepts

- Provide the set of requirements to be met by a type
- Types meeting the requirements *model* the *concept*
- *Concepts* can extend their requirements which is called *refinement*
- Typically *spelled* with CamelCase

Concepts

- Provide the set of requirements to be met by a type
- Types meeting the requirements *model* the *concept*
- *Concepts* can extend their requirements which is called *refinement*
- Typically *spelled* with CamelCase

Concept

Requirements may include:

- C++ expressions that must compile successfully
- Associated types that are related to the model. Usually accessed via typedefs
- Run-time characteristics of the objects that must always be true. Usually pre/post conditions
- Complexity guarantees

Concept

Requirements may include:

- C++ expressions that must compile successfully
- Associated types that are related to the model. Usually accessed via typedefs
- Run-time characteristics of the objects that must always be true. Usually pre/post conditions
- Complexity guarantees

ciere.com

Concept

Requirements may include:

- C++ expressions that must compile successfully
- Associated types that are related to the model. Usually accessed via typedefs
- Run-time characteristics of the objects that must always be true. Usually pre/post conditions
- Complexity guarantees

Concept

Requirements may include:

- C++ expressions that must compile successfully
- Associated types that are related to the model. Usually accessed via typedefs
- Run-time characteristics of the objects that must always be true. Usually pre/post conditions
- Complexity guarantees

Concept - example ForwardIterator

Characteristic	Valid Expressions
Can be default-constructed	`X a;` `X();`
Can be copied and copy-constructed	`X b(a);` `b = a;`
Accepts equality/inequality comparisons. Equal iterators imply the same element is pointed	`a == b` `a != b`
Can be dereferenced (when not null)	`*a` `a->m`

Can be incremented (when not null)	`++a`
	`a++`
	`*a++`

Concept - BidirectionalIterator *refinement*

Characteristic	Valid Expressions
Can be default-constructed	`X a;` `X();`
Can be copied and copy-constructed	`X b(a);` `b = a;`
Accepts equality inequality comparisons. Equal iterators imply the same element is pointed	`a == b` `a != b`
Can be dereferenced (when not null)	`*a` `a->m`

Can be incremented (when not null)	`++a`
	`a++`
	`*a++`

Can be decremented (when not null)	`--a`
	`a--`
	`*a--`

ciere.com

Outline

Run-time Policy

A *Policy* allows us to specializes behavior.

```
template< typename T >
void fill( vector<int> & v, T done )
{
    int i = 0;
    while( !done() )
    {
        v.push_back( i++ );
    }
}

vector<int> stuff;
fill( stuff,
        [&]{ return stuff.size() >= 8; } );
```

Policy Class

A *Policy Class* is a template parameter that specializes behavior and is selected at compile time.

```
std::basic_string< charT, traits, Alloc >

namespace std {
  typedef basic_string<char> string;
}
```

Policy Class - Example

```
template< typename T
        , typename CheckingPolicy=NoChecking
        , typename BadPointerPolicy=BadPointerDoNothing  >
class pointer_wrapper
{
public:
   pointer_wrapper()                     : value_(0)   {}
   explicit pointer_wrapper(T* p) : value_(p)   {}

   operator T*()
   {
      if( ! CheckingPolicy::check_pointer(value_) )
      {
         return BadPointerPolicy::handle_bad_pointer(value_);
      }
      else{  return value_;    }
   }

private:
   T* value_;
};
```

Policy Class - Example

```
struct NoChecking
{
   template< typename T >
   static bool check_pointer( T* ){ return true; }
};


template<  typename T
         , typename CheckingPolicy=NoChecking
         , typename BadPointerPolicy=BadPointerDoNothing  >
class pointer_wrapper
{

   operator T*()
   {
      if( ! CheckingPolicy::check_pointer(value_) )
      {
         return BadPointerPolicy::handle_bad_pointer(value_);
      }
      else{  return value_;    }
   }
```

Policy Class - Example

```
struct NullChecking
{
   template< typename T >
   static bool check_pointer( T* p ){ return(p!=0); }
};


template<  typename T
         , typename CheckingPolicy=NoChecking
         , typename BadPointerPolicy=BadPointerDoNothing  >
class pointer_wrapper
{

   operator T*()
   {
      if( ! CheckingPolicy::check_pointer(value_) )
      {
         return BadPointerPolicy::handle_bad_pointer(value_);
      }
      else{  return value_;    }
   }
```

Policy Class - Example

```
struct BadPointerDoNothing
{
   template< typename T >
   static T* handle_bad_pointer( T* p )
   {
      std::cout << "pointer is moldy" << std::endl;
      return p;
   }
};

template<  typename T
         , typename CheckingPolicy=NoChecking
         , typename BadPointerPolicy=BadPointerDoNothing  >
class pointer_wrapper
{
   operator T*()
   {
      if( ! CheckingPolicy::check_pointer(value_) )
      {
         return BadPointerPolicy::handle_bad_pointer(value_);
      }
      else{  return value_;    }
   }
```

Policy Class - Example

```
template<  typename T
         , typename CheckingPolicy=NoChecking
         , typename BadPointerPolicy=BadPointerDoNothing  >
class pointer_wrapper



pointer_wrapper<int> your_number( new int );
*your_number = 42;
std::cout << "your_number: " << *your_number << std::endl;

pointer_wrapper<int> my_number;
*my_number = 42;
std::cout << "my_number: " << *my_number << std::endl;
```

Output

```
your_number:  42
Segmentation fault
```

Policy Class - Example

```
template<  typename T
         , typename CheckingPolicy=NoChecking
         , typename BadPointerPolicy=BadPointerDoNothing  >
class pointer_wrapper
```

```
pointer_wrapper<int> your_number( new int );
*your_number = 42;
std::cout << "your_number: " << *your_number << std::endl;

pointer_wrapper<int> my_number;
*my_number = 42;
std::cout << "my_number: " << *my_number << std::endl;
```

Output

```
your_number:  42
Segmentation fault
```

Policy Class - Example

```
template<  typename T
         , typename CheckingPolicy=NoChecking
         , typename BadPointerPolicy=BadPointerDoNothing  >
class pointer_wrapper
```

```
pointer_wrapper<int,NullChecking> your_number( new int );
*your_number = 42;
std::cout << "your_number: " << *your_number << std::endl;

pointer_wrapper<int,NullChecking> my_number;
*my_number = 42;
std::cout << "my_number: " << *my_number << std::endl;
```

Output

```
your_number:  42
pointer is moldy
Segmentation fault
```

Policy Class - Example

```
template<  typename T
         , typename CheckingPolicy=NoChecking
         , typename BadPointerPolicy=BadPointerDoNothing  >
class pointer_wrapper
```

```
pointer_wrapper<int,NullChecking> your_number( new int );
*your_number = 42;
std::cout << "your_number: " << *your_number << std::endl;

pointer_wrapper<int,NullChecking> my_number;
*my_number = 42;
std::cout << "my_number: " << *my_number << std::endl;
```

Output

```
your_number:  42
pointer is moldy
Segmentation fault
```

Outline

CRTP

Curiously Recurring Template Pattern

```
struct derived : base<derived>
{
  ...
};
```

- Static Polymorphism
- Injecting Behaviour

CRTP

Curiously Recurring Template Pattern

```
struct derived : base<derived>
{
   ...
};
```

- Static Polymorphism
- Injecting Behaviour

ciere.com

CRTP

Curiously Recurring Template Pattern

```
struct derived : base<derived>
{
   ...
};
```

- Static Polymorphism
- Injecting Behaviour

ciere.com

Static Polymorphism

Curiously Recurring Template Pattern

```
template <class Derived>
struct base
{
    void interface()
    {
        // ...
        static_cast<Derived*>(this)->implementation();
        // ...
    }
};

struct derived : base<derived>
{
     void implementation();
};
```

Static Polymorphism

```
template <typename Derived>
struct cloneable
{
   Derived* clone() const
   {
      return new Derived(static_cast<Derived const&>(*this));
   }
};

struct bar : cloneable<bar>
{
   int value;
};

bar my_bar;
my_bar.value = 42;
bar* my_clone = my_bar.clone();
std::cout << "my_clone: " << my_clone->value << std::endl;
```

Output

```
my_clone: 42
```

Static Polymorphism

```
template <typename Derived>
struct cloneable
{
   Derived* clone() const
   {
      return new Derived(static_cast<Derived const&>(*this));
   }
};

struct bar : cloneable<bar>
{
   int value;
};

bar my_bar;
my_bar.value = 42;
bar* my_clone = my_bar.clone();
std::cout << "my_clone: " << my_clone->value << std::endl;
```

Output

```
my_clone: 42
```

Static Polymorphism

```
template <typename Derived>
struct cloneable
{
    Derived* clone() const
    {
        return new Derived(static_cast<Derived const&>(*this));
    }
};

struct bar : cloneable<bar>
{
    int value;
};

bar my_bar;
my_bar.value = 42;
bar* my_clone = my_bar.clone();
std::cout << "my_clone: " << my_clone->value << std::endl;
```

Output

```
my_clone: 42
```

Static Polymorphism

```
template <typename Derived>
struct cloneable
{
    Derived* clone() const
    {
        return new Derived(static_cast<Derived const&>(*this));
    }
};

struct bar : cloneable<bar>
{
    int value;
};

bar my_bar;
my_bar.value = 42;
bar* my_clone = my_bar.clone();
std::cout << "my_clone: " << my_clone->value << std::endl;
```

Output

```
my_clone: 42
```

Inject Behaviour

```
struct handler : boost::enabled_shared_from_this<handler>
{
...
};
```

Inject Behaviour

```
template<class T>
class enable_shared_from_this
{
public:
    shared_ptr<T> shared_from_this()
    {
        shared_ptr<T> p( weak_this_ );
        return p;
    }

    shared_ptr<T const> shared_from_this() const
    {
        shared_ptr<T const> p( weak_this_ );
        return p;
    }

private:
    mutable weak_ptr<T> weak_this_;
};
```

Exercise

Using CRTP, write a base class that add a method:

```
float profit();
```

Such that profit is calculated as:

$$profit = 42 \times \frac{output}{input}$$

and output and input are members of the derived type

Exercise

```cpp
template< class Derived >
struct enable_profit
{
    float profit()
    {
        Derived const & derived =
            static_cast<Derived const&>(*this);

        return 42 * derived.output / derived.input;
    }
};


struct department : enable_profit<department>
{
    int output;
    int input;
};
```

Outline

Type Traits

- Query characteristics about types
- Template specialization is pattern matching

Check if we have an `int`

```
template< typename T >
struct is_int
{
   static const bool value = false;
};

template<>
struct is_int<int>
{
   static const bool value = true;
};


cout << "float : " << is_int<float>::value << "  ";
cout << "int   : " << is_int<int>::value   << "  ";
cout << "bar   : " << is_int<bar>::value   << endl;
```

Output

```
float :  0 int :  1 bar :  0
```

Type Traits

Check if we have an `int`

```
template< typename T >
struct is_int
{
   static const bool value = false;
};

template<>
struct is_int<int>
{
   static const bool value = true;
};


cout << "float : " << is_int<float>::value << "  ";
cout << "int   : " << is_int<int>::value   << "  ";
cout << "bar   : " << is_int<bar>::value   << endl;
```

Output

```
float :  0 int :  1 bar :  0
```

Type Traits

Two types the same?

```
template< typename T1, typename T2 >
struct is_same
{
   static const bool value = false;
};

template<typename T>
struct is_same<T,T>
{
   static const bool value = true;
};


cout << "same: " << is_same<int,bar>::value << endl;
cout << "same: " << is_same<bar,bar>::value << endl;
```

Output

```
same: 0
same: 1
```

Type Traits

Two types the same?

```
template< typename T1, typename T2 >
struct is_same
{
   static const bool value = false;
};

template<typename T>
struct is_same<T,T>
{
   static const bool value = true;
};


cout << "same: " << is_same<int,bar>::value << endl;
cout << "same: " << is_same<bar,bar>::value << endl;
```

Output

```
same: 0
same: 1
```

Type Traits

- Boost.TypeTraits
- C++11 : `<type_traits>`

Exercise

Write a type trait that will be true if the type is a `boost::shared_ptr`

Exercise

Write a type trait that will be true if the type is a
`boost::shared_ptr`

```
template< typename T>
struct is_shared_ptr
{
   static const bool value = false;
};

template< typename T >
struct is_shared_ptr< boost::shared_ptr<T> >
{
   static const bool value = true;
};
```

Outline

Distance between two points

Old-school

Common base class.

- Intrusive
- Restrictive
- Requires runtime wrappers

Modern C++

Tag dispatching or SFINAE

- Non-intrusive
- Completely generic
- Compile time - fast runtime

ciere.com

Distance between two points

Old-school

Common base class.

- Intrusive
- Restrictive
- Requires runtime wrappers

Modern C++

Tag dispatching or SFINAE

- Non-intrusive
- Completely generic
- Compile time - fast runtime

Generic to the Max - the point

```
struct mypoint
{
    double x, y;
};

double distance(mypoint const& a, mypoint const& b)
{
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}
```

ciere.com

Generic to the Max - a different point

```
struct yourpoint
{
    double x, y;
};

template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return std::sqrt(dx * dx + dy * dy);
}
```

Generic to the Max - hidden points

```
class secret_point
{
   public:
      double get_x() const;
      double get_y() const;

   private:
      ...
};

template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
    double dx = get<0>(a) - get<0>(b);
    double dy = get<1>(a) - get<1>(b);
    return std::sqrt(dx * dx + dy * dy);
}
```

Generic to the Max - traits

```
namespace traits
{
    template <typename P, int D>
    struct access {};
}

namespace traits
{
    template <>
    struct access<mypoint, 0>
    {
        static double get(mypoint const& p)
        {
            return p.x;
        }
    };
```

Generic to the Max - traits

```
namespace traits
{
    template <typename P, int D>
    struct access {};
}

namespace traits
{
    template <>
    struct access<mypoint, 0>
    {
        static double get(mypoint const& p)
        {
            return p.x;
        }
    };
```

Generic to the Max - specializing

```
namespace traits
{
    template <>
    struct access<mypoint, 0>
    {
        static double get(mypoint const& p)
        {
            return p.x;
        }
    };

    template <>
    struct access<mypoint, 1>
    {
        static double get(mypoint const& p)
        {
            return p.y;
        }
    };
}
```

Generic to the Max - specializing

```
namespace traits
{
    template <>
    struct access<secret_point, 0>
    {
        static double get(secret_point const& p)
        {
            return p.get_x();
        }
    };

    template <>
    struct access<secret_point, 1>
    {
        static double get(secret_point const& p)
        {
            return p.get_y();
        }
    };
}
```

Generic to the Max - a lot of typing

```
namespace traits
{
    template <typename P, int D>
    struct access {};
}


template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
   double dx =
      trait::access<P1,0>::get(a) -
      trait::access<P2,0>::get(b);

   double dy =
      trait::access<P1,1>::get(a) -
      trait::access<P2,1>::get(b);

   return std::sqrt(dx * dx + dy * dy);
}
```

Generic to the Max - use a proxy

```
namespace traits
{
    template <typename P, int D>
    struct access {};
}

template <int D, typename P>
inline double get(P const& p)
{
    return traits::access<P, D>::get(p);
}
```

Generic to the Max - We did it!?

```
template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
    double dx = get<0>(a) - get<0>(b);
    double dy = get<1>(a) - get<1>(b);
    return std::sqrt(dx * dx + dy * dy);
}
```

```
template <int D, typename P>
inline double get(P const& p)
{
    return traits::access<P, D>::get(p);
}
```

Factorial - with Lambda

```
std::function<int(int)> fact;

fact =
    [&fact](int n)->int
        {
            if(n==0){ return 1; }
            else
            {
                return (n * fact(n-1));
            }
        } ;

cout << "factorial(4) : " << fact(4) << endl;
```

ciere.com

Factorial - Compile Time

Using TMP, calculate the factorial of N at compile time.

- Print result for factorial of 3
- Print result for factorial of 5

Factorial - Compile Time

```
#include <iostream>

template< int N >
struct factorial
{
   enum{ value = N*factorial<N-1>::value };
};

template<>
struct factorial<0>
{
   enum{ value = 1 };
};

int main()
{
   std::cout << "fact of 5: " << factorial<5>::value << "\n";
   std::cout << "fact of 3: " << factorial<3>::value << "\n";
   return 1;
}
```

Factorial - Compile Time

```cpp
#include <iostream>

template< int N >
struct factorial
{
   static const int value = N*factorial<N-1>::value;
};

template<>
struct factorial<0>
{
   static const int value = 1;
};

int main()
{
   std::cout << "fact of 5: " << factorial<5>::value << "\n";
   std::cout << "fact of 3: " << factorial<3>::value << "\n";
   return 1;
}
```

Generic to the Max - Euclidean distance

For Cartesian coordinates, if **p** and **q** are points in Euclidian *n*-space, the distance from **p** to **q** is:

$$d(p, q) = d(q, p) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2}$$

Generic to the Max - Euclidean distance

For Cartesian coordinates, if **p** and **q** are points in Euclidian *n*-space, the distance from **p** to **q** is:

$$d(p, q) = d(q, p) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2}$$

Generic to the Max - Dimension Agnostic

```
template <typename P1, typename P2, int D>
struct pythagoras
{
    static double apply(P1 const& a, P2 const& b)
    {
        double d = get<D-1>(a) - get<D-1>(b);
        return d * d + pythagoras<P1, P2, D-1>::apply(a, b);
    }
};
```

```
template <typename P1, typename P2 >
struct pythagoras<P1, P2, 0>
{
    static double apply(P1 const&, P2 const&)
    {
        return 0;
    }
};
```

Generic to the Max - Dimension Agnostic

```
template <typename P1, typename P2, int D>
struct pythagoras
{
    static double apply(P1 const& a, P2 const& b)
    {
        double d = get<D-1>(a) - get<D-1>(b);
        return d * d + pythagoras<P1, P2, D-1>::apply(a, b);
    }
};
```

```
template <typename P1, typename P2 >
struct pythagoras<P1, P2, 0>
{
    static double apply(P1 const&, P2 const&)
    {
        return 0;
    }
};
```

Generic to the Max - Dimension Agnostic

Was:

```
template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
    double dx = get<0>(a) - get<0>(b);
    double dy = get<1>(a) - get<1>(b);
    return std::sqrt(dx * dx + dy * dy);
}
```

Now:

```
template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
    BOOST_STATIC_ASSERT(
        ( dimension<P1>::value == dimension<P2>::value ) );

    return sqrt(
        pythagoras<P1,P2, dimension<P1>::value>::apply(a,b) );
}
```

Generic to the Max - more traits

```
namespace traits
{
    template <typename P>
    struct dimension {};
}
```

```
namespace traits
{
    template <>
    struct dimension<mypoint> : boost::mpl::int_<2>
    {};
}
```

Generic to the Max - more traits

```
namespace traits
{
    template <typename P>
    struct dimension {};
}
```

```
namespace traits
{
    template <>
    struct dimension<mypoint> : boost::mpl::int_<2>
    {};
}
```

Generic to the Max - more traits and a meta-function

```cpp
namespace traits
{
    template <typename P>
    struct dimension {};
}
```

```cpp
namespace traits
{
    template <>
    struct dimension<mypoint> : boost::mpl::int_<2>
    {};
}
```

```cpp
template <typename P>
struct dimension : traits::dimension<P>
{};
```

Generic to the Max - Now we did it!?

```
template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
    BOOST_STATIC_ASSERT(
       ( dimension<P1>::value == dimension<P2>::value ) );

    return sqrt(
       pythagoras<P1,P2, dimension<P1>::value>::apply(a,b) );
}
```

Generic to the Max

The return type is a double `)o:`

```
template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
    BOOST_STATIC_ASSERT(
        ( dimension<P1>::value == dimension<P2>::value ) );

    return sqrt(
        pythagoras<P1,P2, dimension<P1>::value>::apply(a,b) );
}
```

Generic to the Max - Coordinate Type

```
namespace traits
{
    template <typename P>
    struct coordinate_type{};

    // specialization for our mypoint
    template <>
    struct coordinate_type<mypoint>
    {
        typedef double type;
    };
}


template <typename P>
struct coordinate_type :    traits::coordinate_type<P> {};
```

Generic to the Max - Generic Pythagoras

```
template <typename P1, typename P2, int D>
struct pythagoras
{
    static double apply(P1 const& a, P2 const& b)
    {
        double d = get<D-1>(a) - get<D-1>(b);
        return d * d + pythagoras<P1, P2, D-1>::apply(a, b);
    }
};

    static computation_t apply(P1 const& a, P2 const& b)
    {
        computation_t d = get<D-1>(a) - get<D-1>(b);
        return d * d + pythagoras <P1,P2, D-1> ::apply(a,b);
    }
```

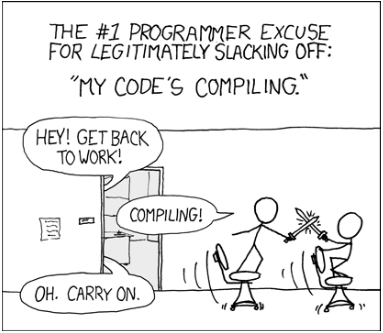
Generic to the Max - Generic Pythagoras

```
template <typename P1, typename P2, int D>
struct pythagoras
{
    typedef typename select_most_precise
        <
            typename coordinate_type<P1>::type,
            typename coordinate_type<P2>::type
        >::type computation_t;

    static computation_t apply(P1 const& a, P2 const& b)
    {
        computation_t d = get<D-1>(a) - get<D-1>(b);
        return d * d + pythagoras <P1,P2, D-1> ::apply(a,b);
    }
};
```

Generic to the Max - Generic Pythagoras

```
template <typename P1, typename P2, int D>
struct pythagoras
{
    typedef typename select_most_precise
        <
            typename coordinate_type<P1>::type,
            typename coordinate_type<P2>::type
        >::type computation_t;

    static computation_t apply(P1 const& a, P2 const& b)
    {
        computation_t d = get<D-1>(a) - get<D-1>(b);
        return d * d + pythagoras <P1,P2, D-1> ::apply(a,b);
    }
};
```

Modern C++ - Abuse Your Compiler



Exercise

$$profit = 42 \times \frac{input}{output}$$

Exercise

$$profit_{float} = 42 \times \frac{output}{input}$$

```
struct vendor_type{ float output, input; };
vendor_type vendor::get_in_out();
```

```
class department{
   int get_widgets() const;
   int get_managers() const;  };
```

ouput : widgets

input : managers

```
class sw_engineer{
   int cppnow_attend_count() const;
   int daily_cups_of_coffee() const;
   int manager_count() const;          };
```

ouput : years attending C++Now! * inverse of cups of coffee

input : manager count

Exercise

$$profit_{float} = 42 \times \frac{output}{input}$$

```
struct vendor_type{ float output, input; };
vendor_type vendor::get_in_out();
```

```
class department{
   int get_widgets() const;
   int get_managers() const;   };
```

ouput : widgets

input : managers

```
class sw_engineer{
   int cppnow_attend_count() const;
   int daily_cups_of_coffee() const;
   int manager_count() const;          };
```

ouput : years attending C++Now! * inverse of cups of coffee

input : manager count

Exercise

$$profit_{float} = 42 \times \frac{output}{input}$$

```
struct vendor_type{ float output, input; };
vendor_type vendor::get_in_out();
```

```
class department{
   int get_widgets() const;
   int get_managers() const;  };
```

ouput : widgets

input : managers

```
class sw_engineer{
   int cppnow_attend_count() const;
   int daily_cups_of_coffee() const;
   int manager_count() const;          };
```

ouput : years attending C++Now! * inverse of cups of coffee

input : manager count

Exercise

$$profit_{float} = 42 \times \frac{output}{input}$$

```
struct vendor_type{ float output, input; };
vendor_type vendor::get_in_out();
```

```
class department{
   int get_widgets() const;
   int get_managers() const;   };
```

ouput : widgets

input : managers

```
class sw_engineer{
   int cppnow_attend_count() const;
   int daily_cups_of_coffee() const;
   int manager_count() const;          };
```

ouput : years attending C++Now! * inverse of cups of coffee

input : manager count

Exercise

$$profit_{float} = 42 \times \frac{output}{input}$$

```
struct vendor_type{ float output, input; };
vendor_type vendor::get_in_out();
```

```
class department{
   int get_widgets() const;
   int get_managers() const;   };
```

ouput : widgets

input : managers

```
class sw_engineer{
   int cppnow_attend_count() const;
   int daily_cups_of_coffee() const;
   int manager_count() const;          };
```

ouput : years attending C++Now! * inverse of cups of coffee

input : manager count

Exercise

```
namespace tag
{
   template< typename T >
   struct profit{};

   template<>
   struct profit<vendor_type>
   {
      static int output( vendor_type const& v){ return v.output; }
      static int input( vendor_type const& v){ return v.input; }
   };

   template<>
   struct profit<department>
   {
      static int output(department const& v){ return v.get_widgets(); }
      static int input(department const& v){ return v.get_managers(); }
   };

   template<>
   struct profit<sw_engineer>
   {
      static int output(sw_engineer const& v)
      { return v.cppnow_attend_count()
            *  v.daily_cups_of_coffee();
      }
      static int input(sw_engineer const& v){ return v.manager_count(); }
   };
}

template< typename T >
float profit( T const & v )
{
   return 42 * tag::profit<T>::output(v) / tag::profit<T>::input(v) ;
}
```

Outline

SFINAE

Substitution Failure is Not an Error

How to abuse SFINAE

```
typedef Foo foo_t;

make_instance<foo_t> f;
foo_t foo = f( config );
```

```
typedef boost::shared_ptr<Bar> shared_bar_t;

make_instance<shared_bar_t> f;
shared_bar_t bar = f( config );
```

enable_if

```
template<  typename InstType
         , class    SharedEnabled = void
         >
struct make_instance
{
   InstType operator()(special::value& v) const
   {
      InstType inst;
      fill_adapted<InstType>::apply(inst, v);
      return inst;
   }
};
```

enable_if

```
template< typename InstType  >
struct make_instance
     <  InstType
      , typename enable_if< is_shared_ptr<InstType> >::type
     >
{
   InstType operator()(special::value& v) const
   {
      InstType
         inst( new typename InstType::element_type );

      fill_adapted<typename InstType::element_type>
         ::apply(*inst, v);

      return inst;
   }
};
```

enable_if from `ciere::json`

```
template <typename T>
value( T val,
       typename enable_if<is_floating_point<T>, T>::type* = 0)
   : base_type( (double_t(val)) )
{}


template <typename T>
value( T val,
       typename enable_if<
          boost::mpl::or_<
                is_integral<T>
              , is_enum<T>
            >, T>::type* = 0)
   : base_type( (int_t(val)) )
{}
```

enable_if

- Learn `boost::enable_if`
- Learn about existing type traits
- Use judiciously

Outline

Dimensional Analysis

The Word Problem

Butler drops a slushie from the second story cottage window. The window is 6-meters above the garden.

How long does it take for the slushie to plumet to the tomato bush?

Dimensional Analysis

Equation

$$h = \frac{1}{2}gt^2 \quad \rightarrow \quad t = \sqrt{\frac{2h}{g}}$$

Where

$$g \approx 9.8\,\frac{\mathrm{m}}{\mathrm{s}^2}$$

$$h = 6\,\mathrm{m}$$

Dimensional Analysis

Units

$$\mathrm{s} = \sqrt{\frac{\mathrm{m}}{\frac{\mathrm{m}}{\mathrm{s}^2}}} \rightarrow \sqrt{\mathrm{s}^2} \rightarrow \mathrm{s} = \mathrm{s}$$

ciere.com

Dimensional Analysis

Units

$$\mathrm{s} = \frac{\mathrm{m}}{\frac{\mathrm{m}}{\mathrm{s}^2}} \rightarrow \mathrm{s}^2 \neq \mathrm{s}$$

ciere.com

Dimensional Analysis - In Code

```
double d = 6.0;
double g = 9.8;
double t = 2.0 * d / g;

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output

```
d:  6
g:  9.8
t:  1.22449
```

Dimensional Analysis - In Code

```
double d = 6.0;
double g = 9.8;
double t = 2.0 * d / g;

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output

```
d:  6
g:  9.8
t:  1.22449
```

ciere.com

Dimensional Analysis - In Code

```
double d = 6.0;
double g = 9.8;
double t = 2.0 * d / g;

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output

```
d:  6
g:  9.8
t:  1.22449
```

ciere.com

Dimensional Analysis - In Code

```
double d = 6.0;
double g = 9.8;
double t = 2.0 * d / g;

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output

```
d:  6
g:  9.8
t:  1.22449
```

ciere.com

Dimensional Analysis - In Code

```
double d = 6.0;
double g = 9.8;
double t = 2.0 * d / g;

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output

```
d:  6
g:  9.8
t:  1.22449
```

Dimensional Analysis - In Code

```
quantity< length >       d( 6.0 * meters );
quantity< acceleration > g( 9.8 * meters_per_second_squared );
quantity< time >         t = 2.0 * d / g;

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output - clang

```
units.cpp:13:33: error: no viable conversion from ...
...
(aka 'quantity<unit_type, value_type>') to 'quantity<si::time>'
   quantity< si::time >         t = 2.0 * d / g;
                                ^   ~~~~~~~~~~~
```

ciere.com

Dimensional Analysis - In Code

```
quantity< length >       d( 6.0 * meters );
quantity< acceleration > g( 9.8 * meters_per_second_squared );
quantity< time >         t = 2.0 * d / g;

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output - clang

```
units.cpp:13:33: error: no viable conversion from ...
...
(aka 'quantity<unit_type, value_type>') to 'quantity<si::time>'
   quantity< si::time >         t = 2.0 * d / g;
                                ^   ~~~~~~~~~~~
```

ciere.com

Dimensional Analysis - In Code

```
quantity< length >       d( 6.0 * meters );
quantity< acceleration > g( 9.8 * meters_per_second_squared );
quantity< time >         t = 2.0 * d / g;

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output - clang

```
units.cpp:13:33: error: no viable conversion from ...
...
(aka 'quantity<unit_type, value_type>') to 'quantity<si::time>'
   quantity< si::time >        t = 2.0 * d / g;
                               ^   ~~~~~~~~~~~
```

ciere.com

Dimensional Analysis - In Code

```
quantity< length >       d( 6.0 * meters );
quantity< acceleration > g( 9.8 * meters_per_second_squared );
quantity< time >         t = 2.0 * d / g;

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output - clang

```
units.cpp:13:33: error: no viable conversion from ...
...
(aka 'quantity<unit_type, value_type>') to 'quantity<si::time>'
   quantity< si::time >          t = 2.0 * d / g;
                                 ^   ~~~~~~~~~~~
```

ciere.com

Dimensional Analysis - In Code

```
quantity< length >       d( 6.0 * meters );
quantity< acceleration > g( 9.8 * meters_per_second_squared );
quantity< time >         t = 2.0 * d / g;

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output - clang

```
units.cpp:13:33: error: no viable conversion from ...
...
(aka 'quantity<unit_type, value_type>') to 'quantity<si::time>'
   quantity< si::time >         t = 2.0 * d / g;
                                ^   ~~~~~~~~~~~
```

ciere.com

Dimensional Analysis - In Code

```
quantity< length >       d( 6.0 * meters );
quantity< acceleration > g( 9.8 * meters_per_second_squared );
quantity< time >         t = sqrt( 2.0 * d / g );

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output

```
d:  6 m
g:  9.8 m s^-2
t:  1.10657 s
```

ciere.com

Dimensional Analysis - In Code

```
quantity< length >       d( 6.0 * meters );
quantity< acceleration > g( 9.8 * meters_per_second_squared );
quantity< time >         t = sqrt( 2.0 * d / g );

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output

```
d:  6 m
g:  9.8 m s^-2
t:  1.10657 s
```

ciere.com

Dimensional Analysis - In Code

```
quantity< length >       d( 6.0 * meters );
quantity< acceleration > g( 9.8 * meters_per_second_squared );
quantity< time >         t = sqrt( 2.0 * d / g );

std::cout << "d: " << d << std::endl;
std::cout << "g: " << g << std::endl;
std::cout << "t: " << t << std::endl;
```

Output

```
d:  6 m
g:  9.8 m s^-2
t:  1.10657 s
```

ciere.com

Dimensional Analysis - At Compile-Time

- No runtime execution cost
- Errors reported at compile time

Dimensional Analysis - Representations

At runtime we might ...

```
typedef int dimension[7];     // m l t ...
dimension const mass          = {1,0,0,0,0,0,0};
dimension const length        = {0,1,0,0,0,0,0};
dimension const time          = {0,0,1,0,0,0,0};
...
dimension const acceleration = {0,1,-2,0,0,0,0};
```

Dimensional Analysis - Representations

At runtime we might ...

```
typedef int dimension[7];     // m l t ...
dimension const mass         = {1,0,0,0,0,0,0};
dimension const length       = {0,1,0,0,0,0,0};
dimension const time         = {0,0,1,0,0,0,0};
...
dimension const acceleration = {0,1,-2,0,0,0,0};
```

ciere.com

Dimensional Analysis - Representations

At runtime we might ...

```
typedef int dimension[7];     // m l t ...
dimension const mass          = {1,0,0,0,0,0,0};
dimension const length        = {0,1,0,0,0,0,0};
dimension const time          = {0,0,1,0,0,0,0};
...
dimension const acceleration = {0,1,-2,0,0,0,0};
```

Dimensional Analysis - Representations

At runtime we might ...

```
typedef int dimension[7];     // m l t ...
dimension const mass          = {1,0,0,0,0,0,0};
dimension const length        = {0,1,0,0,0,0,0};
dimension const time          = {0,0,1,0,0,0,0};
...
dimension const acceleration = {0,1,-2,0,0,0,0};
```

Dimensional Analysis - Representations

At runtime we might ...

```
typedef int dimension[7];     // m l t ...
dimension const mass          = {1,0,0,0,0,0,0};
dimension const length        = {0,1,0,0,0,0,0};
dimension const time          = {0,0,1,0,0,0,0};
...
dimension const acceleration = {0,1,-2,0,0,0,0};
```

Dimensional Analysis - Representations

We can use compile-time constructs to acheive the same

```
// dimensions                    m l t ...
typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length;
typedef mpl::vector_c<int,0,0,1,0,0,0,0> time;
...

typedef mpl::vector_c<int,0,2, 0,0,0,0,0> area;
typedef mpl::vector_c<int,0,1,-2,0,0,0,0> acceleration;
```

ciere.com

Dimensional Analysis - Representations

We can use compile-time constructs to acheive the same

```
// dimensions                  m l t ...
typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length;
typedef mpl::vector_c<int,0,0,1,0,0,0,0> time;
...

typedef mpl::vector_c<int,0,2, 0,0,0,0,0> area;
typedef mpl::vector_c<int,0,1,-2,0,0,0,0> acceleration;
```

Dimensional Analysis - Representations

We can use compile-time constructs to acheive the same

```
// dimensions                 m l t ...
typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length;
typedef mpl::vector_c<int,0,0,1,0,0,0,0> time;
...

typedef mpl::vector_c<int,0,2, 0,0,0,0,0> area;
typedef mpl::vector_c<int,0,1,-2,0,0,0,0> acceleration;
```

Dimensional Analysis - Representations

We can use compile-time constructs to acheive the same

```
// dimensions                     m l t ...
typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length;
typedef mpl::vector_c<int,0,0,1,0,0,0,0> time;
...

typedef mpl::vector_c<int,0,2, 0,0,0,0,0> area;
typedef mpl::vector_c<int,0,1,-2,0,0,0,0> acceleration;
```

ciere.com

Dimensional Analysis - Representations

We can use compile-time constructs to acheive the same

```
// dimensions                  m l t ...
typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length;
typedef mpl::vector_c<int,0,0,1,0,0,0,0> time;
...

typedef mpl::vector_c<int,0,2, 0,0,0,0,0> area;
typedef mpl::vector_c<int,0,1,-2,0,0,0,0> acceleration;
```

ciere.com

Dimensional Analysis - Representations

We can use compile-time constructs to acheive the same

```
// dimensions                    m l t ...
typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length;
typedef mpl::vector_c<int,0,0,1,0,0,0,0> time;
...

typedef mpl::vector_c<int,0,2, 0,0,0,0,0> area;
typedef mpl::vector_c<int,0,1,-2,0,0,0,0> acceleration;
```

ciere.com

Dimensional Analysis - Wrapper for Quantities

```
template <class T, class Dimensions>
struct quantity
{
    explicit quantity(T x)
      : value_(x)
    {}

    T value() const { return value_; }

  private:
    T value_;
};

quantity<double,length>  d( 1.0 );
quantity<double,time>    t( 3.4 );
d = t;
```

ciere.com

Dimensional Analysis - Wrapper for Quantities

```
template <class T, class Dimensions>
struct quantity
{
    explicit quantity(T x)
      : value_(x)
    {}

    T value() const { return value_; }

  private:
    T value_;
};

quantity<double,length>  d( 1.0 );
quantity<double,time>    t( 3.4 );
d = t;
```

Dimensional Analysis - Wrapper for Quantities

```
template <class T, class Dimensions>
struct quantity
{
    explicit quantity(T x)
      : value_(x)
    {}

    T value() const { return value_; }

  private:
    T value_;
};

quantity<double,length>  d( 1.0 );
quantity<double,time>    t( 3.4 );
d = t;
```

Output - clang

```
units_adhoc.cpp:36:6: error: no viable overloaded '='
    d = t;
    ~ ^ ~
```

Dimensional Analysis - Addition

```
template <class T, class D>
quantity<T,D>
operator+(quantity<T,D> x, quantity<T,D> y)
{
  return quantity<T,D>(x.value() + y.value());
}

template <class T, class D>
quantity<T,D>
operator-(quantity<T,D> x, quantity<T,D> y)
{
  return quantity<T,D>(x.value() - y.value());
}
```

Dimensional Analysis - Addition

```
quantity<double,si::time>      t1( 1.0 );
quantity<double,si::time>      t2( 3.4 );
quantity<double,si::length>    l1( 2.8 );

t1 = t2 + l1;

std::cout << "t1: " << t1.value() << std::endl;
```

Output - clang

```
units_adhoc.cpp:84:12: error: invalid operands to binary
expression ('quantity<double, si::time>'
and 'quantity<double, si::length>')
   t1 = t2 + l1;
        ~~ ^ ~~
```

Dimensional Analysis - Addition

```
quantity<double,si::time>      t1( 1.0 );
quantity<double,si::time>      t2( 3.4 );
quantity<double,si::length>    l1( 2.8 );

t1 = t2 + l1;

std::cout << "t1: " << t1.value() << std::endl;
```

Output - clang

```
units_adhoc.cpp:84:12: error: invalid operands to binary
expression ('quantity<double, si::time>'
and 'quantity<double, si::length>')
   t1 = t2 + l1;
        ~~ ^ ~~
```

ciere.com

Dimensional Analysis - Addition

```
quantity<double,si::time>     t1( 1.0 );
quantity<double,si::time>     t2( 3.4 );
quantity<double,si::time>     t3( 2.8 );

t1 = t2 + t3;

std::cout << "t1: " << t1.value() << std::endl;
```

Example

```
t1: 6.2
```

Dimensional Analysis - Addition

```
quantity<double,si::time>     t1( 1.0 );
quantity<double,si::time>     t2( 3.4 );
quantity<double,si::time>     t3( 2.8 );

t1 = t2 + t3;

std::cout << "t1: " << t1.value() << std::endl;
```

Example

```
t1: 6.2
```

Dimensional Analysis - The Price

```
double add_test(double x, double y, double z)
{
   quantity<double,si::time>     t1( x );
   quantity<double,si::time>     t2( y );
   quantity<double,si::time>     t3( z );

   t1 = t2 + t3;
   return t1.value();
}

double add_double(double x, double y, double z)
{
   double     d1( x );
   double     d2( y );
   double     d3( z );

   d1 = d2 + d3;
   return d1;
}
```

Dimensional Analysis - The Price : g++ -O2

```
**** double
**** add_double(double x,
****            double y,
****            double z)
**** {
               .loc 1 93 0
               .cfi_startproc
               .cfi_personality 0x3,
                      __gxx_personality_v0
               .LVL2:
               .loc 1 93 0
 0020 660F28C1  movapd %xmm1, %xmm0
               .LVL3:
 0024 F20F58C2  addsd %xmm2, %xmm0
****   double     d1(x);
****   double     d2(y);
****   double     d3(z);
****
****   d1 = d2 + d3;
****   return d1;
**** }
               .loc 1 100 0
 0028 C3        ret
               .cfi_endproc
```

```
**** double
**** add_test(double x,
****          double y,
****          double z)
**** {
               .loc 1 80 0
               .cfi_startproc
               .cfi_personality 0x3,
                      __gxx_personality_v0
               .LVL0:
               .loc 1 80 0
 0010 660F28C1  movapd %xmm1, %xmm0
               .LVL1:
 0014 F20F58C2  addsd %xmm2, %xmm0
****   quantity<double,time> t1(x);
****   quantity<double,time> t2(y);
****   quantity<double,time> t3(z);
****
****   t1 = t2 + t3;
****   return t1.value();
**** }
               .loc 1 87 0
 0018 C3        ret
               .cfi_endproc
```

Dimensional Analysis - Multiply

```
template <class T, class D1, class D2>
quantity< T,
          typename mpl::transform< D1,D2,
                                   mpl::plus<_1,_2> >::type
>
operator*( quantity<T,D1> x, quantity<T,D2> y )
{
   typedef typename
           mpl::transform<D1,D2,
                          mpl::plus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() * y.value() );
}
```

ciere.com

Dimensional Analysis - Multiply

```
template <class T, class D1, class D2>
quantity< T,
          typename mpl::transform< D1,D2,
                                   mpl::plus<_1,_2> >::type
>
operator*( quantity<T,D1> x, quantity<T,D2> y )
{
   typedef typename
           mpl::transform<D1,D2,
                          mpl::plus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() * y.value() );
}
```

ciere.com

Dimensional Analysis - Multiply

```
template <class T, class D1, class D2>
quantity< T,
          typename mpl::transform< D1,D2,
                                   mpl::plus<_1,_2> >::type
>
operator*( quantity<T,D1> x, quantity<T,D2> y )
{
   typedef typename
           mpl::transform<D1,D2,
                          mpl::plus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() * y.value() );
}
```

Dimensional Analysis - Multiply

```
template <class T, class D1, class D2>
quantity< T,
          typename mpl::transform< D1,D2,
                                   mpl::plus<_1,_2> >::type
>
operator*( quantity<T,D1> x, quantity<T,D2> y )
{
   typedef typename
           mpl::transform<D1,D2,
                          mpl::plus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() * y.value() );
}
```

Dimensional Analysis - Multiply

```
template <class T, class D1, class D2>
quantity< T,
          typename mpl::transform< D1,D2,
                                   mpl::plus<_1,_2> >::type
>
operator*( quantity<T,D1> x, quantity<T,D2> y )
{
   typedef typename
           mpl::transform<D1,D2,
                          mpl::plus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() * y.value() );
}
```

ciere.com

Dimensional Analysis - Multiply

```
template <class T, class D1, class D2>
quantity< T,
          typename mpl::transform< D1,D2,
                                   mpl::plus<_1,_2> >::type
>
operator*( quantity<T,D1> x, quantity<T,D2> y )
{
   typedef typename
           mpl::transform<D1,D2,
                          mpl::plus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() * y.value() );
}
```

Dimensional Analysis - Multiply

```
template <class T, class D1, class D2>
quantity< T,
          typename mpl::transform< D1,D2,
                                    mpl::plus<_1,_2> >::type
>
operator*( quantity<T,D1> x, quantity<T,D2> y )
{
   typedef typename
           mpl::transform<D1,D2,
                          mpl::plus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() * y.value() );
}
```

ciere.com

Dimensional Analysis - Multiply

```
template <class T, class D1, class D2>
quantity< T,
          typename mpl::transform< D1,D2,
                                   mpl::plus<_1,_2> >::type
>
operator*( quantity<T,D1> x, quantity<T,D2> y )
{
   typedef typename
           mpl::transform<D1,D2,
                          mpl::plus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() * y.value() );
}
```

Dimensional Analysis - Multiply

```
quantity<double,si::length>   d2( 2.4 );
quantity<double,si::length>   d3( 1.7 );
quantity<double,si::area>     a( 1.0 );

a = d2 * d3;
```

Output - clang

```
units_adhoc.cpp:153:6: error: no viable overloaded '='
   a = d2 * d3;
   ~ ^ ~~~~~~~
```

ciere.com

Dimensional Analysis - Multiply

```
quantity<double,si::length>   d2( 2.4 );
quantity<double,si::length>   d3( 1.7 );
quantity<double,si::area>     a( 1.0 );

a = d2 * d3;
```

Output - clang

```
units_adhoc.cpp:153:6: error: no viable overloaded '='
   a = d2 * d3;
   ~ ^ ~~~~~~~
```

ciere.com

Dimensional Analysis - Multiply

```
template <class T, class Dimensions>
struct quantity
{
   template <class OtherDimensions>
   quantity(quantity<T,OtherDimensions> const& rhs)
      : value_(rhs.value())
   {
      BOOST_STATIC_ASSERT((
         mpl::equal<Dimensions,OtherDimensions>::type::value
      ));
   }

   ...

};
```

ciere.com

Dimensional Analysis - Multiply

```
template <class T, class Dimensions>
struct quantity
{
    template <class OtherDimensions>
    quantity(quantity<T,OtherDimensions> const& rhs)
        : value_(rhs.value())
    {
        BOOST_STATIC_ASSERT((
            mpl::equal<Dimensions,OtherDimensions>::type::value
        ));
    }

    ...

};
```

ciere.com

Dimensional Analysis - Multiply

```
template <class T, class Dimensions>
struct quantity
{
   template <class OtherDimensions>
   quantity(quantity<T,OtherDimensions> const& rhs)
      : value_(rhs.value())
   {
      BOOST_STATIC_ASSERT((
         mpl::equal<Dimensions,OtherDimensions>::type::value
      ));
   }

   ...

};
```

Dimensional Analysis - Multiply

```
template <class T, class Dimensions>
struct quantity
{
   template <class OtherDimensions>
   quantity(quantity<T,OtherDimensions> const& rhs)
      : value_(rhs.value())
   {
      BOOST_STATIC_ASSERT((
         mpl::equal<Dimensions,OtherDimensions>::type::value
      ));
   }

   ...

};
```

ciere.com

Dimensional Analysis - Multiply

```
quantity<double,si::length>   d2( 2.4 );
quantity<double,si::length>   d3( 1.7 );
quantity<double,si::area>     a( 1.0 );

a = d2 * d3;
quantity<double,si::length> d1 = d2 * d3;
```

Output - clang

```
units_adhoc.cpp:35:7: error: implicit instantiation of
      undefined template
      'boost::STATIC_ASSERTION_FAILURE<0>'
      BOOST_STATIC_ASSERT((
      ^
...
   quantity<double,si::length> d1 = d2 * d3;
```

Dimensional Analysis - Multiply

```
quantity<double,si::length>   d2( 2.4 );
quantity<double,si::length>   d3( 1.7 );
quantity<double,si::area>     a( 1.0 );

a = d2 * d3;
quantity<double,si::length> d1 = d2 * d3;
```

Output - clang

```
units_adhoc.cpp:35:7: error: implicit instantiation of
      undefined template
      'boost::STATIC_ASSERTION_FAILURE<0>'
      BOOST_STATIC_ASSERT((
      ^
...
   quantity<double,si::length> d1 = d2 * d3;
```

Dimensional Analysis - Check out Boost.Units

- No runtime execution cost
- Errors reported at compile time
- Scaled Units
- Conversion Factors

Explore

Find out more :

- Boost.MPL - Meta Programming Library
- Boost.Fusion - Algorithms on Tuples!

http://ciere.com/cppnow12/