Solving World Problems with Fusion

Professional C++ Training

ciere consulting

Michael Caisse

ciere.com

Part I

Motivation

Outline

- Distance

Tuple Points

```
typedef std::tuple<int,int> point_2d_t;

point_2d_t a(2,2);
point_2d_t b(4,4);
double d = distance(a,b);
```

```
typedef std::tuple<int,int,int> point_3d_t;

point_3d_t a(2,2,2);
point_3d_t b(4,4,4);
double d = distance(a,b);
```

Euclidean distance

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

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

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

Euclidean distance

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

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

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

Distance with Tuples

```
template< typename P1, typename P2 >
double distance(P1 p1, P2 p2)
{
    static_assert(    tuple_size<P1>::value
                   == tuple_size<P2>::value
                  , "error : point dimensions must match." );

    double accumulated =
       pythagoras<P1,P2,tuple_size<P1>::value>::apply(p1,p2);

    return sqrt(accumulated);
}
```

Distance with Tuples

```
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;
    }
};
```

Tuple Points

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

```
typedef std::tuple<int,int> point_2d_t;

point_2d_t a(2,2);
mypoint    b{4,4};
double d = distance( a
                   , forward_as_tuple(b.x,b.y) );
```

Tuple Points - Fail

```
class secret_point
{
public:
    secret_point(double x, double y) : x_(x), y_(y) {}

    double get_x() const     { return x_; }
    void   set_x(double d)   { x_=d; }

    double get_y() const     { return y_; }
    void   set_y(double d)   { y_=d; }

private:
    double x_, y_;
};


secret_point a(2,2);
mypoint      b{4,4};
double d = distance( ?????
                   , forward_as_tuple(b.x,b.y) );
```

Euclidean distance

$$\sum_{i=1}^{n}(q_i - p_i)^2$$

- Fold, accumulate, reduce
- Convolution, zip

Euclidean distance

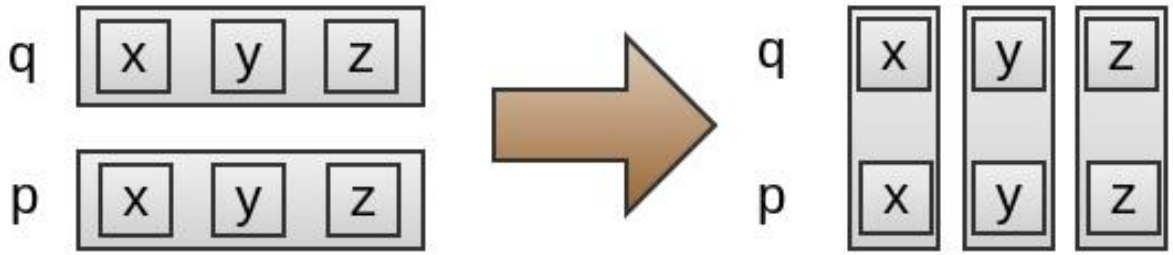
$$\sum_{i=1}^{n}(q_i - p_i)^2$$

- **Fold, accumulate, reduce**
- Convolution, zip

Euclidean distance

$$\sum_{i=1}^{n}(q_i - p_i)^2$$

- Fold, accumulate, reduce
- **Convolution, zip**



`std::tuple` Interface

`template< class... Types >`
`class tuple;`
(constructor)
`operator=`
`swap`

`make_tuple`
`tie`
`forward_as_tuple`
`std::get`
==,!=,<,<=,>,>=
`swap`
`tuple_size`
`tuple_element`

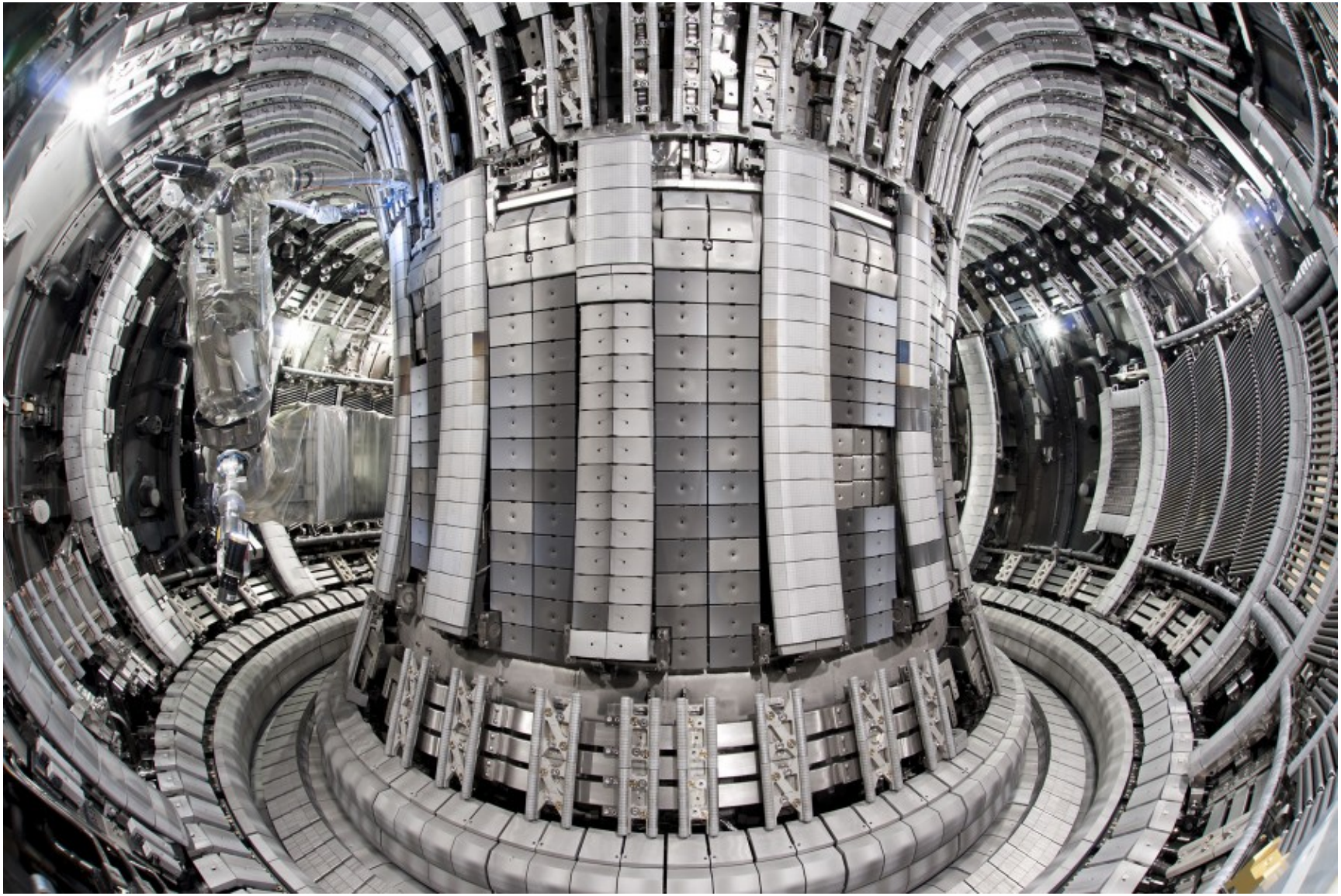
`std::tuple` Interface

`template< class... Types >`
`class tuple;`
(constructor)
`operator=`
`swap`

`make_tuple`
`tie`
`forward_as_tuple`
`std::get`
==,!=,<,<=,>,>=
`swap`
`tuple_size`
`tuple_element`

- Iterators?
- Ranges?
- Algorithms?

Fusion



Fusion

Provides

- Iterators
- Ranges
- Algorithms

Also...

- Variety of Containers
- Light Weight Views
- Functional Features
- Sequence Adaption
- Extension Mechanism

Fusion

Provides

- Iterators
- Ranges
- Algorithms

Also...

- Variety of Containers
- Light Weight Views
- Functional Features
- Sequence Adaption
- Extension Mechanism

Fusion

Fusion brings together Compile Time and Run Time for working with heterogenous collections.

It is the *fusion* of compile time Meta Programming and runtime programming.

Tuple Points - Fusion

```
typedef std::tuple<int,int> point_2d_t;

point_2d_t a(2,2);
point_2d_t b(4,4);
double d = distance(a,b);
```

```
typedef std::tuple<int,int,int> point_3d_t;

point_3d_t a(2,2,2);
point_3d_t b(4,4,4);
double d = distance(a,b);
```

Distance with Tuples - Fusion

```
template< typename P1, typename P2 >
double distance(P1 p1, P2 p2)
{
    static_assert(    result_of::size<P1>::value
                   == result_of::size<P2>::value
                  , "error : point dimensions must match." );

    typedef vector<P1&,P2&> zip_t;

    double accumulated = fold( zip_view<zip_t>( zip_t(p1,p2) )
                             , 0
                             , pythagoras() );

    return sqrt(accumulated);
}
```

Distance with Tuples - Fusion

```
struct pythagoras
{
    typedef double result_type;

    template<typename T>
    double operator()(double acc, T const & axis) const
    {
        double d = at_c<0>(axis) - at_c<1>(axis);
        return acc + d*d;
    }
};
```

Distance with Fusion - User Defined Types

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

```
typedef std::tuple<int,int> point_2d_t;

point_2d_t a(2,2);
mypoint    b{4,4};
double d = distance(a,b);
```

Distance with Fusion - User Defined Types

```
class secret_point
{
public:
    secret_point(double x, double y) : x_(x), y_(y)   {}

    double get_x() const     { return x_; }
    void   set_x(double d)   { x_=d; }

    double get_y() const     { return y_; }
    void   set_y(double d)   { y_=d; }

private:
    double x_, y_;
};
```

```
secret_point a(2,2);
mypoint      b{4,4};
double d = distance(a,b);
```

Fusion - Adapting Structs

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

```
BOOST_FUSION_ADAPT_STRUCT(
    mypoint,
    (double, x)
    (double, y)
)
```

Fusion - Adapting Abstract Data Types

```
class secret_point
{
public:
    secret_point(double x, double y) : x_(x), y_(y)   {}

    double get_x() const     { return x_; }
    void   set_x(double d)   { x_=d; }

    double get_y() const     { return y_; }
    void   set_y(double d)   { y_=d; }

private:
    double x_, y_;
};


BOOST_FUSION_ADAPT_ADT(
    secret_point,
    (double, double, obj.get_x(), obj.set_x(val) )
    (double, double, obj.get_y(), obj.set_y(val) )
)
```

Part II

Overview

Outline

- Sequence
- Iterators
- Container
- Adapted Types
- Views
- Algorithms
- Functional

Concepts

The concepts for Sequence

Forward Sequence

Run Time			Compile Time
`begin(s)`	Forward Iterator		`begin<S>::type`
`end(s)`	Forward Iterator		`end<S>::type`
`size(s)`	int size		`size<S>::type`
`empty(s)`	bool		`empty<S>::type`
`front(s)`	Any type		`front<S>::type`
`front(s) = o`	Any type		

Bidirectional Sequence

Adds to Forward Sequence:

Run Time			Compile Time
`begin(s)`	Bidirectional Iterator		`begin<S>::type`
`end(s)`	Bidirectional Iterator		`end<S>::type`
`back(s)`	Any type		`back<S>::type`
`back(s) = o`	Any type		

Random Access Sequence

Adds to Bidirectional Sequence:

Run Time			Compile Time
`begin(s)`	Random Access Iterator		`begin<S>::type`
`end(s)`	Random Access Iterator		`end<S>::type`
`at_c<N>(s)`	Any type		
`at_c<N>(s)=o`	Any type		
`at<M>(s)`	Any type		`at<S,M>::type`
`at<M>(s)=o`	Any type		`at<S,M>::type`

Associative Sequence

Run Time			Compile Time
`has_key<K>(s)`	bool		`has_key<S,K>::type`
`at_key<K>(s)`	Any type		`at_key<S,K>::type`
`at_key<K>(s) = o`	Any type		
			`value_at_key<S,K>::type`

Sequence Intrinsics

Run Time		Compile Time
`begin(s)`		`begin<S>::type`
`end(s)`		`end<S>::type`
`empty(s)`		`empty<S>::type`
`front(s)`		`front<S>::type`
`back(s)`		`back<S>::type`
`size(s)`		`size<S>::type`
`at<M>(s)`		`at<S,M>::type`
`at_c<N>(s)`		`at_c<S,N>::type`
		`value_at<S,M>::type`
		`value_at_c<S,N>::type`
`has_key<K>(s)`		`has_key<S,K>::type`
`at_key<K>(s)`		`at_key<S,K>::type`
		`value_at_key<S,K>::type`
`swap(s1,s2)`		`swap<S,M>::type`

Sequence Intrinsics - Runtime

Run Time	
`begin(s)`	iterator to first element
`end(s)`	iterator to one past end
`empty(s)`	true if an empty sequence
`front(s)`	first element in sequence
`back(s)`	last element in sequence
`size(s)`	number of elements in sequence
`at<M>(s)`	M'th element in sequence
`at_c<N>(s)`	N'th element in sequence
`has_key<K>(s)`	true if the sequence contains Key
`at_key<K>(s)`	element at Key
`swap(s1,s2)`	calls swap for each element between s1 and s2

Sequence Intrinsics

Compile Time	
`begin<S>::type`	returns type of iterator to first element
`end<S>::type`	returns type of iterator to last element
`empty<S>::type`	returns `mpl::true_` if empty
`front<S>::type`	type when dereferencing iter to first element
`back<S>::type`	type when dereferencing iter to last element
`size<S>::type`	`mpl` integral constant of sequence size
`at<S,M>::type`	type returned from RT `at`
`at_c<S,N>::type`	type returned from RT `at_c`
`value_at<S,M>::type`	type of the element at given index
`value_at_c<S,N>::type`	type of the element at given index
`has_key<S,K>::type`	returns `mpl::true_` if contains Key
`at_key<S,K>::type`	returns type from RT `at_key`
`value_at_key<S,K>::type`	returns type of element stored at Key
`swap<S,M>::type`	always returns `void`

Outline

- Sequence
- Iterators
- Container
- Adapted Types
- Views
- Algorithms
- Functional

Iterator Intrinsics

Run Time		Compile Time
`deref(i)`		`deref<I>::type`
		`value_of<I>::type`
`next(i)`		`next<I>::type`
`prior(i)`		`prior<I>::type`
`operator==`		`equal_to<I,J>::type`
`distance(i1,i2)`		`distance<I1,I2>::type`
`advance<M>(i)`		`advance<I,M>::type`
`advance_c<N>(i)`		`advance_c<I,N>::type`
		`key_of<I>::type`
`deref_data(i)`		`deref_data<I>::type`
		`value_of_data<I>::type`

Also `operator*`, `operator!=`,

Outline

- Sequence
- Iterators
- Container
- Adapted Types
- Views
- Algorithms
- Functional

The Containers

Type	Model
`vector`	Random Access Sequence
`cons`	Forward Sequence
`list`	Forward Sequence
`deque`	Bidirectional Sequence
`set`	Associative Sequence
`map`	Associative Sequence

Generating Containers

Function
`make_list`
`make_cons`
`make_vector`
`make_deque`
`make_set`
`make_map`
`list_tie`
`vector_tie`
`deque_tie`
`map_tie`

Generating Containers

Function
make_list
make_cons
make_vector
make_deque
make_set
make_map
list_tie
vector_tie
deque_tie
map_tie

```
make_list( "C++Now!"
         , 42
         , true);
```

Generating Containers

Function
make_list
make_cons
make_vector
make_deque
make_set
make_map
list_tie
vector_tie
deque_tie
map_tie

```
int i = 42;
make_list( "C++Now!"
         , std::ref(i)
         , true);
```

Generating Containers

Function
make_list
make_cons
make_vector
make_deque
make_set
make_map
list_tie
vector_tie
deque_tie
map_tie

```
make_cons
   ( "C++Now!"
   , make_cons
        ( 42
        , make_cons(true)
        )
   );
```

Generating Containers

Function
make_list
make_cons
make_vector
make_deque
make_set
make_map
list_tie
vector_tie
deque_tie
map_tie

```
make_set(42, true);
```

Generating Containers

Function
make_list
make_cons
make_vector
make_deque
make_set
make_map
list_tie
vector_tie
deque_tie
map_tie

```
make_map< uint8_t
        , bool
        , std::string >
    ( "C++Now!"
    , 42
    , true );
```

Generating Containers

Function
make_list
make_cons
make_vector
make_deque
make_set
make_map
list_tie
vector_tie
deque_tie
map_tie

```
struct amazing_conference;
struct amazing_number;
struct valid_key;

std::string conf("C++Now!");
uint8_t value = 42;
bool valid;

auto my_map =
  map_tie< amazing_conference
         , amazing_number
         , valid_key >
       ( conf
       , value
       , valid );

at_key<valid_key>(my_map) = true;
```

Generating Containers

Also, the Meta Function versions.

Function
`make_list`
`make_cons`
`make_vector`
`make_deque`
`make_set`
`make_map`
`list_tie`
`vector_tie`
`deque_tie`
`map_tie`

Converting Containers

Function
`as_list`
`as_vector`
`as_deque`
`as_set`
`as_map`

Converting Containers

Function
as_list
as_vector
as_deque
as_set
as_map

```
auto my_vec
       = make_vector(42,"gorp",false);
auto my_list = as_list(my_vec);
```

Converting Containers

Function
as_list
as_vector
as_deque
as_set
as_map

```
auto my_vec = make_vector(42,false);
auto my_set = as_set(my_vec);
```

Converting Containers

Function
as_list
as_vector
as_deque
as_set
as_map

```
as_map( make_vector(
    make_pair<std::string>(42)
  , make_pair<uint8_t>("fish") ) );
```

Converting Containers

Also, the Meta Function versions.

Function
`as_list`
`as_vector`
`as_deque`
`as_set`
`as_map`

Outline

- Sequence
- Iterators
- Container
- Adapted Types
- Views
- Algorithms
- Functional

Adapted Types

Array
`std::pair`
MPL Sequence
`boost::array`
`boost::tuple`
`std::tuple`
structures
arbitrary data type

```
<boost/fusion/adapted/array.hpp>
<boost/fusion/adapted/std_pair.hpp>
<boost/fusion/adapted/mpl.hpp>
<boost/fusion/adapted/boost_array.hpp>
<boost/fusion/adapted/boost_tuple.hpp>
<boost/fusion/adapted/std_tuple.hpp>
```

Adapted Types

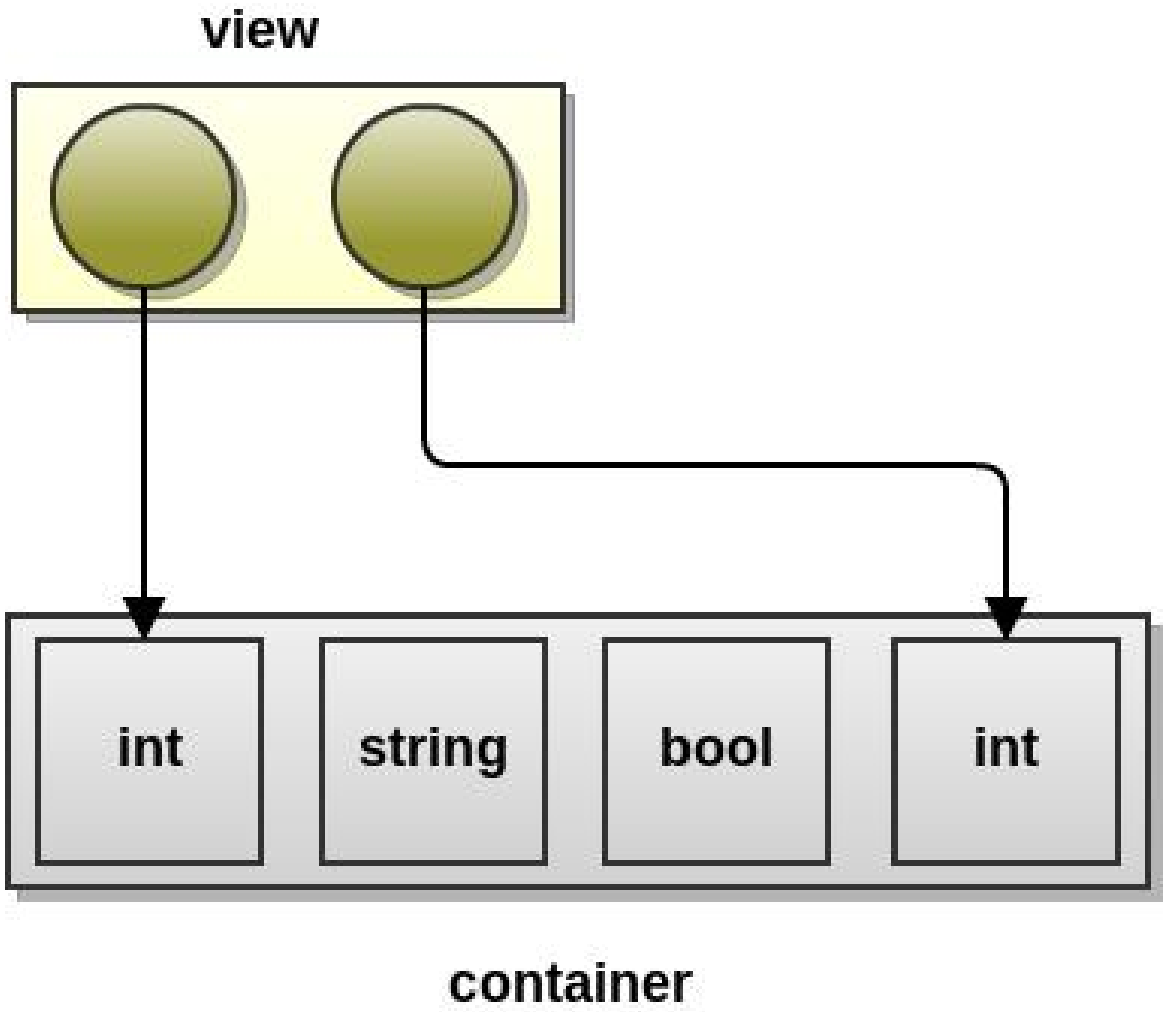
Array
std::pair
MPL Sequence
boost::array
boost::tuple
std::tuple
structures
arbitrary data type

```
<boost/fusion/adapted/array.hpp>
<boost/fusion/adapted/std_pair.hpp>
<boost/fusion/adapted/mpl.hpp>
<boost/fusion/adapted/boost_array.hpp>
<boost/fusion/adapted/boost_tuple.hpp>
<boost/fusion/adapted/std_tuple.hpp>
```

Outline

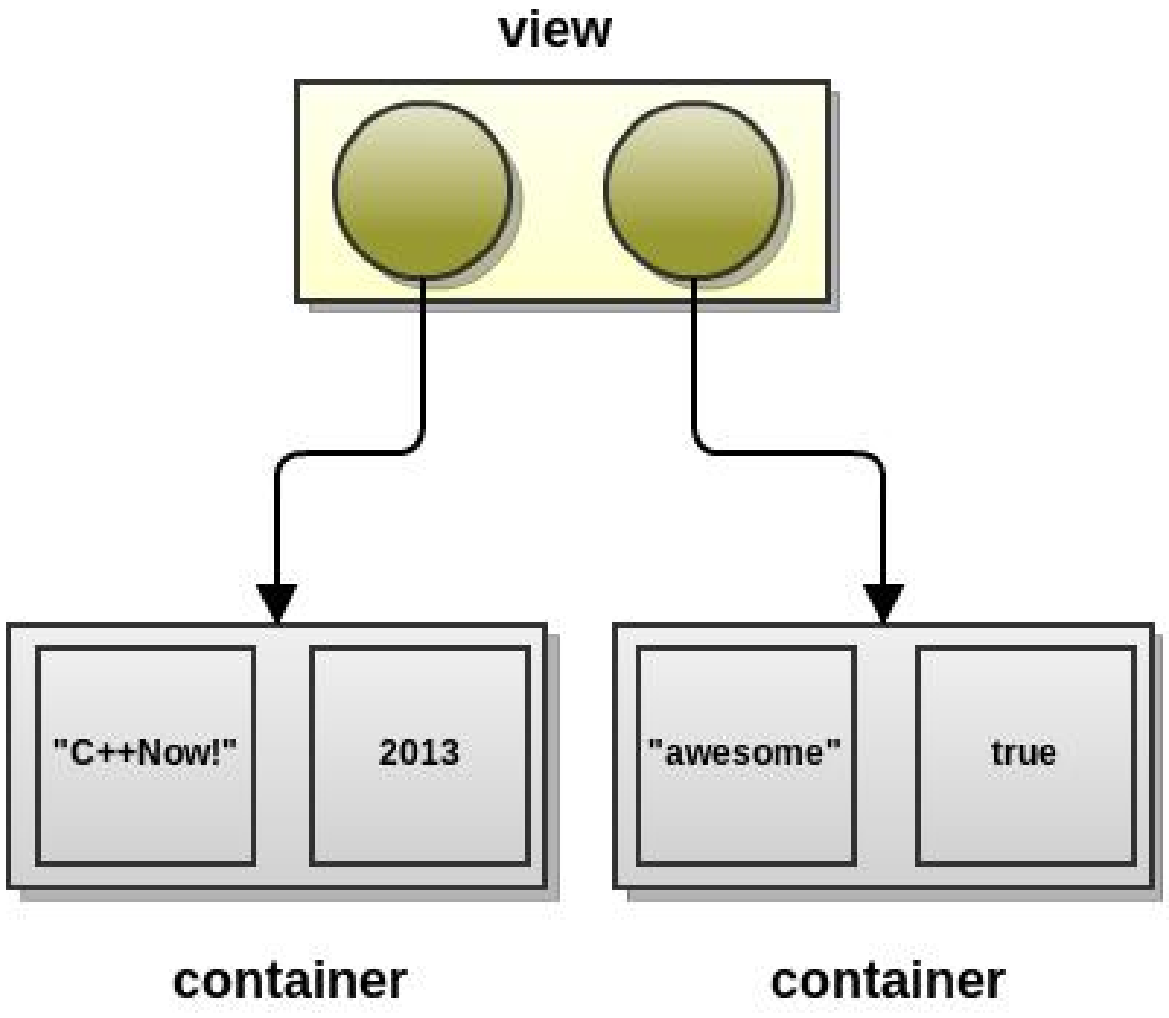
- Sequence
- Iterators
- Container
- Adapted Types
- Views
- Algorithms
- Functional

Alternative Representation



Alternative Representation

```
auto vec = make_vector("C++Now!", 2013);
auto deq = make_deque("awesome", true);

auto view = join(vec,deq);
```



The Views

Type	Model
`single_view`	Random Access Sequence
`filter_view`	Forward / Associative
`iterator_view`	Forward / Bidirectional / Random / Associative
`joint_view`	Forward / Associative
`zip_view`	Forward Sequence
`transform_view`	Forward / Bidirectional / Random
`reverse_view`	Forward / Bidirectional / Random / Associative
`n_view`	Random Access Sequence
`repetitive_view`	Forward Sequence

Outline

- Sequence
- Iterators
- Container
- Adapted Types
- Views
- Algorithms
- Functional

Algorithms

- Are Lazy
- Are not sequence-type preserving
- Return Views

- Auxiliary
- Iteration
- Query
- Transformation

Algorithms

- Are Lazy
- Are not sequence-type preserving
- Return Views

- Auxiliary
- Iteration
- Query
- Transformation

Algorithms

Auxiliary

- `copy`

Iteration

- `fold`
- `reverse_fold`
- `iter_fold`
- `reverse_iter_fold`
- `accumulate`
- `for_each`

Query

- `any`
- `all`
- `none`
- `find`
- `find_if`
- `count`
- `count_if`

Algorithms

Transformations

- `filter`
- `filter_if`
- `transform`
- `replace`
- `replace_if`
- `remove`
- `remove_if`
- `reverse`
- `clear`
- `erase`
- `erase_key`
- `insert`
- `insert_range`
- `join`
- `zip`
- `pop_back`
- `pop_front`
- `push_back`
- `push_front`

Outline

- Sequence
- Iterators
- Container
- Adapted Types
- Views
- Algorithms
- Functional

Overview

Fused

```
void foo(int,std::string,double);

void fused_foo( Sequence );
```

Fused

```
void foo(double,double);

int a[] = {1,3,5};
vector<int,float,double> v(42,1.5,7.8);

transform( zip(a,v)
         , make_fused(&foo) );
```

Invoke

```
void foo(double,int);

vector<float,int> v(7.8,42);

invoke(foo, v);
```

Part III

Power

Outline

- Serialize Example
- Invoke Example
- Signal Group Example

serialize method

```
template< typename T >
void serialize(T v)
{
    fusion::for_each( v, (serial_out()) );
}
```

Serialize `for_each` functor

```
struct serial_out
{
   template< typename T >
   void operator()( T & v ) const
   {
      simple::serialize<T>::write(v);
   }
};
```

Serialize - a few overloads

```
namespace simple
{
   template<typename T> struct serialize{};

   template<>    struct serialize<int32_t>
   {
      static void write(int32_t v) { cout << v; }
   };

   template<>    struct serialize<uint32_t>
   {
      static void write(uint32_t v) { cout << v; }
   };

   template<>    struct serialize<std::string>
   {
      static void write(std::string v)
      {
         std::cout << static_cast<uint32_t>(v.size()) << v ;
      }
   };
}
```

Serialize `for_each` functor updated

```
struct serial_out
{
   template< typename T >   void operator()
      ( T & v
      , typename
        std::enable_if<!is_sequence<T>::value>::type* = 0
      ) const
   { detail::serialize<T>::write(v); }

   template< typename T >   void operator()
      ( T & v
      , typename
        std::enable_if<is_sequence<T>::value>::type* = 0
      ) const
   { serialize(v); }

   template< typename T >
   void operator()( std::vector<T> & v ) const
   {
      detail::serialize<uint32_t>::write(v.size());
      std::for_each(v.begin(),v.end(),*this);
   }
};
```

Outline

- Serialize Example
- Invoke Example
- Signal Group Example

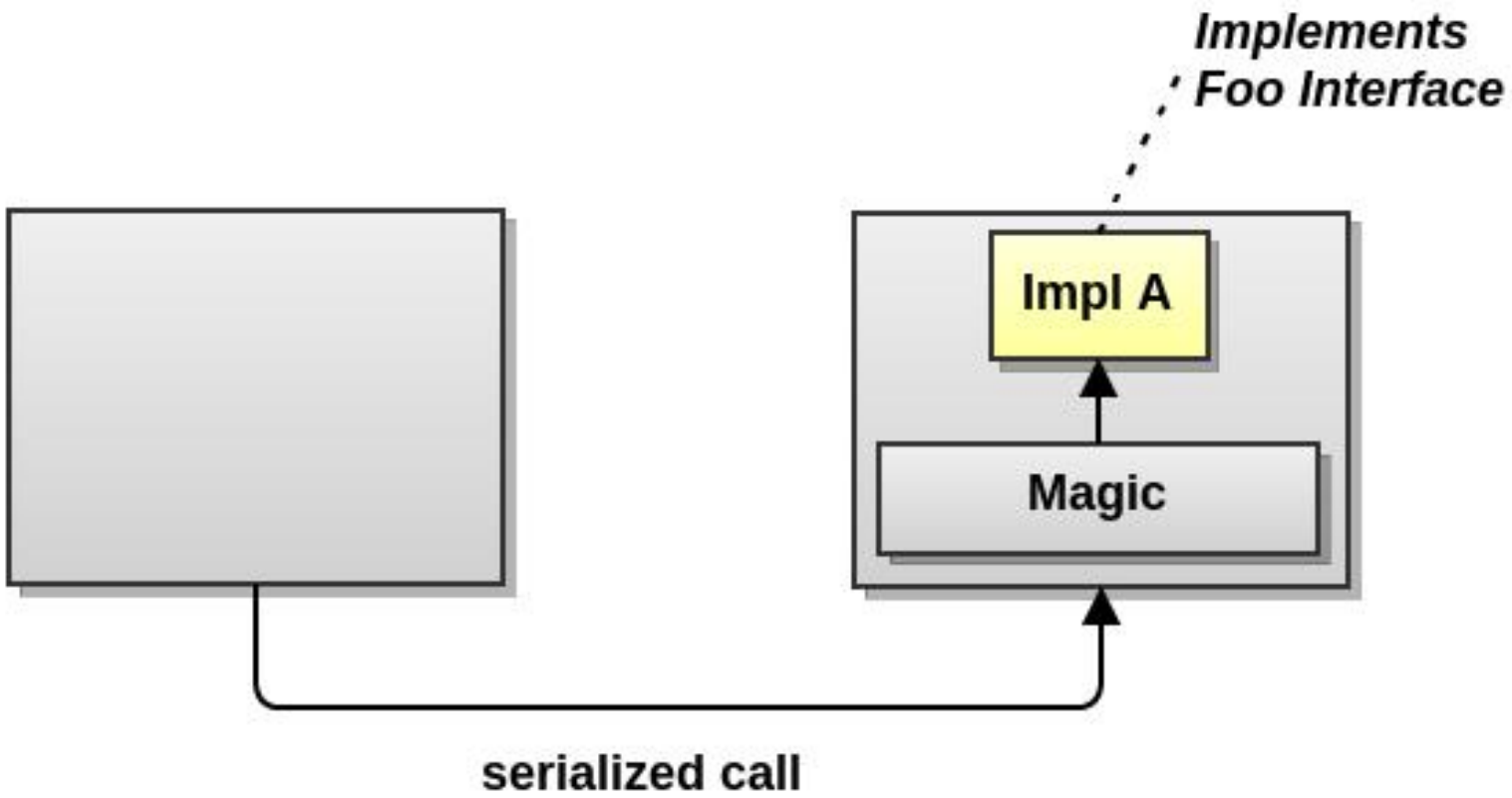
Simple Invoke

```
void do_something(int i, std::string s)
{
    std::cout << i << " " << s << "\n";
}


vector<int,std::string> v(42,"cppnow!");

invoke( do_something
      , v);
```

Simple Invoke

```
struct foo
{
    std::string s;
    int i;
    int j;
    double d;
};

BOOST_FUSION_ADAPT_STRUCT(
    foo,
    (int          , i)
    (std::string , s)
)


foo v{"cppnow!", 42, 8, 1234.5};

invoke( do_something
      , v);
```

Invoke and Map



Invoke and Map

The `foo` interface:

```
interface foo
{
   void stop();
   void start();
   void set_power(int);
   int  get_power();
};
```

Invoke with Map - Types represent methods

```
namespace method
{
    struct stop         {};
    struct start        {};
    struct set_power    { int percent; };
    struct get_power    {};
}


BOOST_FUSION_ADAPT_STRUCT( method::stop,       )

BOOST_FUSION_ADAPT_STRUCT( method::start,      )

BOOST_FUSION_ADAPT_STRUCT( method::get_power,  )

BOOST_FUSION_ADAPT_STRUCT( method::set_power,
                           (int, percent)      )
```

Invoke with Map - Interface description

```
namespace method
{
    struct stop         {};
    struct start        {};
    struct set_power    { int percent; };
    struct get_power    {};
}


struct foo_interface
{
    typedef
    map< pair< method::stop      , std::function<void()>     >
       , pair< method::start     , std::function<void()>     >
       , pair< method::set_power , std::function<void(int)> >
       , pair< method::get_power , std::function<int()>      >
       > call_map_t;
};
```

Invoke with Map - Interface Provider

```
struct foo_provider : rpc_base<foo_interface>
{

   virtual void stop() = 0;
   virtual void start() = 0;
   virtual void set_power(int p) = 0;
   virtual int  get_power() = 0;
};
```

Invoke with Map - RPC Base

```
template<typename Interface>
struct rpc_base
{
    template<typename Method, typename F>
    void tie(F f)
    {

    }

    template<typename Method>
    void make_call(Method args)
    {

    }

    typename Interface::call_map_t call_map;
};
```

Invoke with Map - RPC Base `tie`

```
template<typename Interface>
struct rpc_base
{
    template<typename Method, typename F>
    void tie(F f)
    {
        at_key<Method>(call_map) = f;
    }


    typename Interface::call_map_t call_map;
};
```

Invoke with Map - RPC Base `make_call`

```
template<typename Interface>
struct rpc_base
{
   template<typename Method>
   void make_call(Method args)
   {
      typedef typename
      result_of::value_at_key< typename Interface::call_map_t
                             , Method >::type function_t;

      function_t method = at_key<Method>(call_map);
      if(method)
      {
         invoke(method,args);
      }
   }

   typename Interface::call_map_t call_map;
};
```

Invoke with Map - Interface Provider

```
struct foo_provider : rpc_base<foo_interface>
{
    foo_provider()
    {
        typedef foo_provider inter;
        tie<method::stop>      (bind(&inter::stop, this));
        tie<method::start>     (bind(&inter::start, this));
        tie<method::set_power>(bind(&inter::set_power, this, _1)
        tie<method::get_power>(bind(&inter::get_power, this));
    }

    virtual void stop() = 0;
    virtual void start() = 0;
    virtual void set_power(int p) = 0;
    virtual int  get_power() = 0;
};
```

Invoke with Map - Interface Implementations

```
struct foo : foo_provider
{
   void stop()           { std::cout << "stop\n";    }
   void start()          { std::cout << "stop\n";    }
   void set_power(int p){ std::cout << "power " << p << "\n";
   int  get_power()      { std::cout << "get power\n"; return 0
};

struct gorp : foo_provider
{
   void stop()           { std::cout << "g stop\n";    }
   void start()          { std::cout << "g start\n";    }
   void set_power(int p){ std::cout << "g power " << p << "\n"
   int  get_power()      { std::cout << "g get_power\n"; return
};
```

Invoke with Map

```
// user ....
foo f;

// infrastructure ....
f.make_call(method::set_power{42});


// user ....
gorp g;

// infrastructure ....
g.make_call(method::start());
```

ciere.com

Invoke with Map - Make it better

```
namespace method
{
    struct stop         {};
    struct start        {};
    struct set_power    { int percent; };
    struct get_power    {};
}


struct foo_interface
{
    typedef
    map< pair< method::stop      , std::function<void()>     >
       , pair< method::start     , std::function<void()>     >
       , pair< method::set_power , std::function<void(int)> >
       , pair< method::get_power , std::function<int()>      >
       > call_map_t;
};
```

Invoke with Map - New Interface Description

```
namespace method
{
    struct stop
    {
        typedef std::function<void()> call_sig;
    };

    struct start
    {
        typedef std::function<void()> call_sig;
    };

    struct set_power
    {
        int percent;
        typedef std::function<void(int)> call_sig;
    };

    struct get_power
    {
        typedef std::function<int()> call_sig;
    };
}
```

Invoke with Map - New Interface Description

```
struct foo_interface
    :  mpl::vector<  method::stop
                  ,  method::start
                  ,  method::set_power
                  ,  method::get_power
                  >
{};
```

Invoke with Map - New Interface Description

```
struct foo_provider : rpc_proxy<foo_interface>
```

Invoke with Map - RPC Proxy

```
template<typename Interface>
struct rpc_proxy
{
    template<typename Method, typename F>
    void tie(F f)
    {
        at_key<Method>(call_map) = f;
    }

    template<typename Method>
    void make_call(Method args)
    {

    }


    call_map_t call_map;
};
```

Invoke with Map - RPC Proxy

```
template <typename T>
struct call_sig
{
   typedef typename T::call_sig type;
};


template<typename Interface>
struct rpc_proxy
{
   typedef typename
      mpl::transform< Interface
                    , result_of::make_pair< mpl::_1
                                          , call_sig<mpl::_1>
                                          >::type
                    >::type  call_map_def_t;

   typedef typename
      result_of::as_map<call_map_def_t>::type call_map_t;

   call_map_t call_map;
};
```

Invoke with Map - RPC Proxy

```
template <typename T>
struct call_sig
{
   typedef typename T::call_sig type;
};


template<typename Interface>
struct rpc_proxy
{
   typedef typename
      mpl::transform< Interface
                    , result_of::make_pair< mpl::_1
                                          , call_sig<mpl::_1>
                                          >::type
                    >::type  call_map_def_t;

   typedef typename
      result_of::as_map<call_map_def_t>::type call_map_t;

   call_map_t call_map;
};
```

Invoke with Map - RPC Proxy

```
template<typename Interface>
struct rpc_proxy
{

    template<typename Method>
    void make_call(Method args)
    {
        auto method = at_key<Method>(call_map);
        if(method)
        {
            invoke(method,args);
        }
    }

    call_map_t call_map;
};
```

Invoke with Map - Better

```
struct gorp : foo_provider
{
   void stop()           { std::cout << "g stop\n";    }
   void start()          { std::cout << "g start\n";    }
   void set_power(int p){ std::cout << "g power " << p << "\n"
   int  get_power()      { std::cout << "g get_power\n"; return
};
```

```
gorp g;

// ...
g.make_call(method::start());
```

Invoke with Map - Better

```
void free_set_power(int p)
{
    std::cout << "free set power: " << p << "\n";
}
```

```
rpc_proxy<foo_interface> foo_proxy;
foo_proxy.tie<method::set_power>(free_set_power);

// ...
foo_proxy.make_call(method::set_power{182});
```

Outline

- Serialize Example
- Invoke Example
- Signal Group Example

Simple Group

```
signal_combiner< mpl::vector< group_1
                            , group_2
                            , group_3 > >  sc;


sc.notify_group<group_2>();
sc.notify_signal<sig::a>();
```

Simple Group

```
namespace sig
{
   struct a  {};
   struct b  {};
   struct c  {};
   struct d  {};
   struct e  {};
}

struct group_1 :
   mpl::vector< sig::a
              , sig::c >
{};

struct group_2 :
   mpl::vector< sig::b
              , sig::d >
{};

struct group_3 :
   mpl::vector< sig::e >
{};
```

Simple Group

```
template< typename Groups >
struct signal_combiner
{
   template<typename Signal, typename F>
   void add_listner(F f)
   {
      fusion::at_key<Signal>(sig_map) = f;
   }


   typedef typename
      mpl::transform< typename flatten_group<Groups>::type
                    , result_of::make_pair< mpl::_1
                                          , std::function<void
                                          >::type
                    >::type  sig_map_t;

   sig_map_t sig_map;
};
```

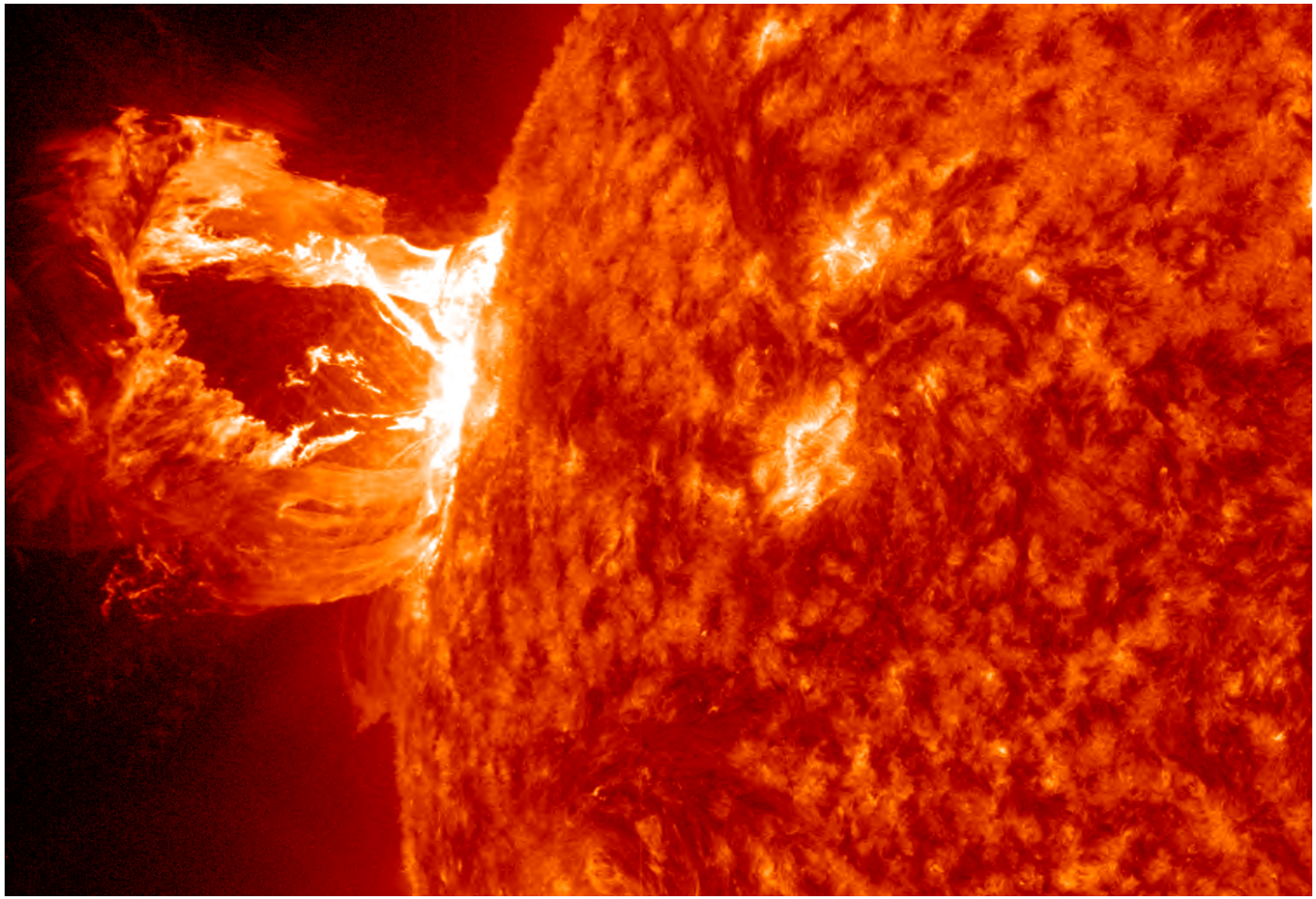
Simple Group

```
template< typename Groups >
struct signal_combiner
{
    template<typename Signal, typename F>
    void add_listner(F f)
    {
        fusion::at_key<Signal>(sig_map) = f;
    }

    template<typename Group>
    void notify_group()
    {
        mpl::for_each<Group>(*this);
    }
    template<typename Signal>
    void operator()(Signal&)
    {
        auto f = fusion::at_key<Signal>(sig_map);
        f();
    }
    sig_map_t sig_map;
};
```

Simple Group

```
template< typename Groups >
struct signal_combiner
{

   template<typename Signal>
   void notify_signal()
   {
      typedef typename mpl::find_if<  Groups
                                    , mpl::contains< mpl::_1,
                                   >::type group_iter;

      typedef typename mpl::deref<group_iter>::type group_t;

      notify_group<group_t>();
   }

   sig_map_t sig_map;
};
```

Fusion - Scary Powerful




ciere.com