Boost Metaprogramming Concepts and Frameworks

A brief 3-hour (!) Introduction

Dave Abrahams, BoostPro

Agenda

- The What, Why, and How
- Tour of Boost Metaprogramming Library (MPL)
- Using MPL for Dimensional Analysis
- Building Finite State Machines

"Meta:"

- A prefix meaning one level of description higher. If X is some concept then **meta-X is data about, or processes operating on, X**.

  For example, a metasyntax is syntax for specifying syntax, metalanguage is a language used to discuss language, meta-data is data about data, and meta-reasoning is reasoning about reasoning.

```
This is difficult to explain briefly, but
much hacker humour turns on deliberate
confusion between meta-levels.
```

- Source: *The Free On-line Dictionary of Computing, © 1993-2004 Denis Howe*

Metaprogram: Our Definition

- Code that explicitly manipulates or generates program elements such as:
  - Functions
  - Types
  - Classes
  - Variables
  - Constants
  - etc.

Compile-Time Computation

```
binary<1101>::value → 13

template <unsigned long N>
struct binary
{
  static unsigned const value
    = N%10
      + 2 * binary<N/10>::value;
};

template <>
struct binary<0>
{
    static unsigned const value
      = 0;
};
```

```
binary<1101>::value =

   1 + 2 * binary<110>::value = 13

binary<110>::value =

    0 + 2 * binary<11>::value = 6

binary<11>::value =

     1 + 2 * binary<1>::value = 3

binary<1>::value =

     1 + 2 * binary<0>::value = 1

binary<0>::value              = 0
```

Why Metaprogramming?

- Why not runtime computation?

```
int binary(unsigned long n)
{
    return n ? n%10 + binary(n/10) * 2 : 0;
}
```

  - Metaprogram is faster at runtime (duh)
  - Metaprogram interacts deeply with host language
- Why not user analysis?

```
long odd_bits = binary<1010101>::value;
long even_bits = 0xAA;
```

  - Metaprogram is more convenient
  - Metaprogram is more reliable
  - Metaprogram brings us closer to the domain abstraction

It Takes Some Effort

Metaprogramming is Library Authors

Boost Metaprogramming Concepts & Frameworks

The Boost Metaprogramming Library: A Whirlwind Tour

Aleksey Gurtovoy (in the hat)

Why a TMP Library?

- Simplify complicated template system
- Hide low-level hacks
- Smooth over implementation differences
- Readability: reduce the syntactic burden
- Conceptual integrity: like STL, the concepts and idioms shape the way we program.
- Fun: high-level programming is more productive than using template "assembly language"

MPL Metafunction Concept

- A class template
- Operating on types only
- Single return "value" (i.e. a type)
- Called `::type`

```
template <class A1, class A2, …An>
struct identifier
{
    typedef type-expression type;
};
```

- If `::value` appears, it's purely optional.

Metadata Polymorphism?

```
template <template <class,class> class C>
struct foo {};

int obj;
void f(long);

foo<char*>                  // OK
foo<1>                      // integral constant
foo<f>                      // function reference
foo<&f>                     // function pointer
foo<obj>                    // object reference
foo<&obj>                   // object pointer
foo<&pair<int,int>::first>  // pointer to data member
foo<std::vector>            // class template
```

Metadata Polymorphism

Solution: turn everything into a type with some kind of wrapper:

```
template <class T, T x>
struct wrap
{};
```

Works for almost everything…

```
wrap<int,5>                                // OK

wrap<                                      // Also OK
     int std::pair<int,int>::*
   , &std::pair<int,int>::first >
{};
```

…except?

MPL Integral Constant Wrappers

```
template <class T, T n>
struct integral_c
{
   typedef T value_type;
   static T const value = n;

   typedef integral_c<T,n> type; // self
   operator T() const { return n; }
};

template <int n>
struct int_ : integral_c<int,n> {};

template <bool x>
struct bool_ : integral_c<bool,x> {};
  …
```

Type Logic

```
typedef bool_<true> true_;
typedef bool_<false> false_;

template <bool C, class T, class F>
struct if_impl;

template <
   class C, class T, class F
>
struct if_
{
    typename
      if_impl<C::value,T,F>::type
    type;
};
```

```
template <class T, class F>
struct if_impl<true,T,F>
{
    typedef T type;
};

template <class T, class F>
struct if_impl<false,T,F>
{
    typedef F type;
};
```

Type Sequences

```
typedef mpl::vector<int*,long,char[4]> v1;

typedef mpl::deref<
    mpl::begin<v1>::type    // MPL iterators
>::type pointer_to_int;     // *v1.begin()

typedef mpl::at<v1, mpl::int_<2> >::type
  four_chars;

typedef mpl::list<int*, long, char[4]> l1;

typedef mpl::set<int*, long, char[4]> s1;
```

Algorithms

```
template<class Iter, class T>                                  // STL
void replace(Iter first, Iter last, T const& old, T const& new);

template<class Sequence, class OldType, class NewType…> // MPL
struct replace;

template<class InIter, class OutIter, class UnaryOp>     // STL
OutIter transform(InIter first, InIter last, OutIter result,
   UnaryOp op);

template <class Sequence, class UnaryOp…>                // MPL
struct transform;

template <class Iter, class T, class BinaryOp>            // STL
T accumulate(Iter first, Iter last, T init, BinaryOp binary_op);

template<class Sequence, class Init, class BinaryOp>     // MPL
struct fold;  // a.k.a. accumulate
```

Lazy Logic

```
struct add_ptr<class T>
{
    typedef T* type;
};
```

```
typedef add_ptr<int&>::type oops;
```

Lazy Logic

```
struct add_ptr_<class T>
{
    typedef T* type;
};

#include <boost/type_traits/is_reference.hpp>
template <class T>
struct add_ptr
{
    typedef typename mpl::if_<
         boost::is_reference<T>
        , T
        , add_ptr_<T>::type
    >::type type;
};
```

```
typedef add_ptr<int&>::type oops;
```

Eagerly instantiated

Lazy Logic

```
struct add_ptr_<class T>          typedef add_ptr<int&>::type oops;
{
    typedef T* type;
};

#include <boost/type_traits/is_reference.hpp>
template <class T>
struct add_ptr
{
    typedef typename mpl::if_<
            boost::is_reference<T>
          , T
          , add_ptr_<T>
    >::type::type type;
};
```

Lazy Logic

```
struct add_ptr_<class T>
{
    typedef T* type;
};

#include <boost/type_traits/is_reference.hpp>
template <class T>
struct add_ptr
{
    typedef typename mpl::if_<
         boost::is_reference<T>
        , mpl::identity<T>
        , add_ptr_<T>
    >::type::type type;
};
```

```
typedef add_ptr<int&>::type ok;
```

Lazy Logic

```
struct add_ptr_<class T>
{
    typedef T* type;
};

#include <boost/type_traits/is_reference.hpp>
template <class T>
struct add_ptr
{
    typedef typename mpl::eval_if<
         boost::is_reference<T>
        , mpl::identity<T>
        , add_ptr_<T>
    >::type type;
};
```

```
typedef add_ptr<int&>::type ok;
```

Views / Lazy Sequences

```
template <class Sequence, class UnaryOp>
struct transform_view;


template <class Sequence, class Predicate>
struct filter_view;


template <class SequenceOfSequences>
struct zip_view;


template <class T, T Start, T Finish>
struct range_c;
```

Example: Unit-Based Computation

- Typechecking for scientists
- Newtons Law:
  - Breakdown: $F = \boldsymbol{m}a$

$$a = \frac{\Delta v}{\Delta \boldsymbol{t}}, \qquad v = \frac{\Delta \boldsymbol{l}}{\Delta \boldsymbol{t}}$$

  - Units of v: $l \,/\, t$ e.g. miles per hour
  - Units of a: $(l \,/\, t) \,/\, t \;=\; l \,/\, t^2$ e.g. feet per sec^2
  - Units of F: $ml \,/\, t^2$ e.g. kg m/s^2

Boost.Units

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

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

    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;
}
```

The Challenge

- Addition/Subtraction
  boring – can't combine different units

  `f(m + l);` // error!

- Multiplication/Division
  interesting – produces new types

  `f(m * l);` // OK – a type representing *ml*

- Challenge: compute the new type

MPL Integral Sequence Wrapper

- Convenient shorthand for type sequence of integral constant wrappers

```
mpl::list_c<int, i1, i2, …in>

≈ mpl::list<
      mpl::int_<i1>, mpl::int_<i1>
    , ...mpl::int_<in>
  >
```

Representing Units

```
typedef list_c<int,1,0,0,0,0,0,0> mass;
typedef list_c<int,0,1,0,0,0,0,0> length;
typedef list_c<int,0,0,1,0,0,0,0> time;
typedef list_c<int,0,0,0,1,0,0,0> charge;
typedef list_c<int,0,0,0,0,1,0,0> temperature;
typedef list_c<int,0,0,0,0,0,1,0> intensity;
typedef list_c<int,0,0,0,0,0,0,1>
   amount_of_substance;
typedef list_c<int,0,0,0,0,0,0,0> scalar; // e.g. pi
typedef list_c<int,0,1,-2,0,0,0,0> acceleration;
```

Representing Quantities

- Bind metadata to a runtime value

```
template <class T, class Units>
class quantity
{
 public:
    explicit quantity(T x)
      : m_value(x) {}

    T value() const
     { return m_value; }

 private:
    T m_value;
};

quantity<float, time> t(99.7);
```

Addition/Subtraction

- Just combine “like quantities”

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

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

Multiplication/Division

- Must compute return units

```
template <class T, class U1, class U2>
quantity<T, ?? >
operator*( quantity<T,U1> x, quantity<T,U2> y );

template <class T, class U1, class U2>
quantity<T, ?? >
operator/( quantity<T,U1> x, quantity<T,U2> y );
```

- Add/subtract corresponding powers:

$$a = v/t = (l/t)/t$$

$$= l^{1} \cdot t^{-1} \cdot t^{-1} = l^{1} \cdot t^{-2}$$

Multiplication/Division

- Use MPL's transform algorithm:

```
template <class Seq1, class Seq2, class BinaryOp>
struct transform;        → Result Sequence
```

- Similar to STL transform:

```
template <
  class InIter1, class InIter2,
  class OutIter, class BinaryOp
>
void transform(
    InIter1 start1, InIter1 finish1, InIter2 start2
  , OutIter, BinaryOp
)
```

What is the BinaryOp?

- MPL gives us plus/minus

```
mpl::plus<mpl::int_<2>, mpl::int_<3> >::type
➔ mpl::int_<5>

mpl::minus<mpl::int_<2>, mpl::int_<3> >::type
➔ mpl::int_<-1>
```

- Are we done now?

```
template <class T, class U1, class U2>
quantity<
     T
   , typename mpl::transform<U1,U2,mpl::plus>::type
 >
 operator*(quantity<T,U1> x, quantity<T,U2> y);
```

ERROR

- Templates are not polymorphic metadata (types)!

Metafunction Classes

- Applying familiar type-wrapper idiom:

```
struct plus_f
{
  template <class T1, class T2>       // nested metafunction
  struct apply
  {
    typedef typename mpl::plus<T1,T2>::type type;
  };
};

template <class T, class U1, class U2>
quantity<
   T, typename mpl::transform<U1,U2,plus_f>::type  // OK
>
operator*(quantity<T,U1> x, quantity<T,U2> y);
```

- Definition: MPL Metafunction Class

  a class with a nested metafunction named "apply"

Division – 3 Simplifications

- #1: "Metafunction Forwarding"

```
struct minus_f
{
    template <class T1, class T2>
    struct apply : mpl::minus<T1,T2> {};
};
```

- Replaces

```
      typedef typename … ::type type;
```

  with:

```
      :
```

- Powerful syntactic trick; use it often… but wrapping metafunctions still boring (no fun)

Simplification #2: Placeholders

```
template <class T, class U1, class U2>
quantity<
    T
  , typename mpl::transform<
      U1,U2,mpl::minus<_1,_2>
     >::type
>
operator/(quantity<T,U1> x, quantity<T,U2> y)
{
    return quantity<
        T
      , typename mpl::transform<
            U1,U2,mpl::minus<_1,_2>
        >::type
    >(
      x.value() / y.value()
    );
}
```

- Not boring! Fun!

Simplification #3: Refactoring

```
template <class U1, class U2>
divide_units
  : mpl::transform<U1,U2,mpl::minus<_1,_2> >
{};

template <class T, class U1, class U2>
quantity<T, typename divide_units<U1,U2>::type>
operator/(quantity<T,U1> x, quantity<T,U2> y)
{
    return quantity<
      T, typename divide_units<U1,U2>::type
    >(
        x.value() / y.value()
     );
}
```

Higher-Order (Meta)Functions

- Functions which operate on or return other functions, e.g. transform

- “Higher-order functional programming”
  – Fancy title for a simple concept
    …but well-earned!

- Possible because metafunctions are polymorphic

Writing a Higher-Order Metafunction

- The Task:

$$\text{twice}(f, x) \quad := \quad f\,(\,f\,(x))$$

- Implementation for metafunction class F:

```
template <class F, class X>
struct twice
{
 private:
    typedef typename F::template apply<X>::type once;
 public:
    typedef
      typename F::template apply<once>::type
    type;
};
```

Writing a Higher-Order Metafunction

- Simplified using forwarding:

```
template <class F, class X>
struct twice
   : F::template apply<
         typename F::template apply<X>::type
     > {};
```

- Refactored:

```
template <class UnaryMFC, class Arg>
struct apply1
   : UnaryMFC::template apply<Arg> {};

template <class F, class X>
struct twice
   : apply1<F, typename apply1<F,X>::type> {};
```

Handling Placeholders

- Add a number to itself **twice**

```
template <class X>
struct times4
   : twice< mpl::plus<_1,_1>, X > {};

typedef times4<int_<4> >::type sixteen; // error!
```

- Why doesn't it work?

```
template <class T1, class T2>
struct plus
{
     typedef ... type;
};
```

- No nested `apply<…>` – not a metafunction class

The Lambda Metafunction

- **`plus<_1,_1>`** is called a "lambda expression"
- `lambda`<*lambda-expr*>`::type` ➔ Metafunction Class

```
template <class X>
struct times4
   : twice<
          typename mpl::lambda<
               mpl::plus<_1,_1>
          >::type, X
      >
{};
```

- Note: `lambda<…>` passes metafunction classes through untouched

The Lambda Calculus

- Branch of mathematical logic
- Deals with the application of functions to their arguments
- Developed in the late '30s and early '40s, by…

Alonzo Church

Internal Lambda

- MPL algorithms apply lambda so you don't have to
- MPL's flexible version of **`apply1`**:

```
template <class UnaryOp, class Arg>
struct apply1
   : lambda<UnaryOp>::type
      ::template apply<Arg>
{};
```

- `mpl::`**`apply`**`<F, A1, A2,` …An> is the n-ary version
- To handle both lambda expressions and metafunction classes as arguments, invoke them with **`apply`**N or **`apply`**

The Power of Lambda

- Transform binary → unary metafunction

```
mpl::plus<_1,_1>
```

- Also via currying (arg. binding)

```
mpl::plus<_1, mpl::int_<42> >
```

- Functional Composition:

```
mpl::times<
    mpl::plus<_1,_2>
  , mpl::minus<_1,_2>
>
```

Boost Metaprogramming Concepts & Frameworks

Building a Domain-Specific Embedded Language (DSEL)

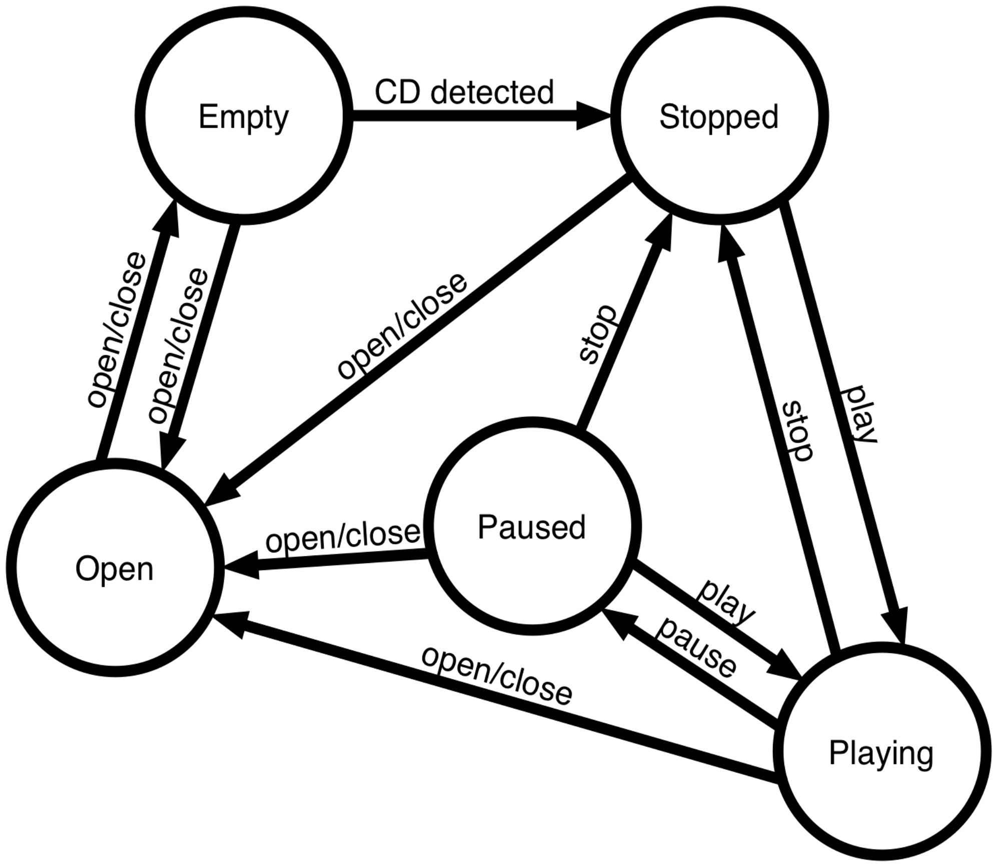
Finite State Machines (FSMs)

- Uses:
  - Embedded system controllers
  - Parsing (YACC parsers are stacks of FSMs)
  - Pattern Matching
  - Hardware Verification
  - Natural Language Processing
  - Any other stateful process
- Not Turing Complete
- Maybe that’s a good thing?

1) Understand Domain Abstraction

- **States** e.g. *locked, unlocked, ready*
- **Transitions** define legal paths from state to state, often with an associated **action**.
- **Events** trigger transitions, e.g. *coin-inserted*, tokens (in a parser). May be internally-generated.

2) Review Standard Notations



Simple CD Player

State Transition Table (STT)

Current State	Event	Next State	Transition Action
Stopped	*play*	*Playing*	start playback
Stopped	*open/close*	*Open*	open drawer
Open	*open/close*	*Empty*	close drawer; collect CD information
Empty	*open/close*	*Open*	open drawer
Empty	*cd detected*	*Stopped*	store CD information
Playing	*stop*	*Stopped*	stop playback
Playing	*pause*	*Paused*	pause playback
Playing	*open/close*	*Open*	stop playback; open drawer
Paused	*play*	*Playing*	resume playback
Paused	*stop*	*Stopped*	stop playback
Paused	*open/close*	*Open*	stop playback; open drawer

3) Choose an *Embedded* Notation

```
     // start     event        finish      action
fsm  [ stopped,  play,         playing,    start_playback  ]
     [ playing,  pause,        paused,     pause_playback  ]...

fsm  [ stopped|  play    |  playing    |  start_playback  ]
     [ playing|  pause   |  paused     |  pause_playback  ]...

fsm= +---------+---------------+----------+-----------------+
     ! stopped | on<play>()    | playing  | start_playback  |
     +---------+---------------+----------+-----------------+
     ! playing | on<pause>()   | paused   | pause_playback  |
     +---------+---------------+----------+-----------------+
     ...end;

 //   +---------+-------+----------+--------------------+
 row<  stopped , play   , playing , &p::start_playback  >,
 //   +---------+-------+----------+--------------------+
 row<  playing , pause  , paused  , &p::pause_playback  >
 //   +---------+-------+----------+--------------------+
```

4) Consider Usage of Product

```
int main()
{
  player p;                   // FSM object

  p.process_event(drawer()); // user opens CD player
  p.process_event(drawer()); // inserts CD and
  closes

  p.process_event(            // CD is detected
    cd_detected(
      "louie, louie"
     , std::vector<std::clock_t>( ...track lengths... )
    )
  );
  p.process_event(play());   // etc.
}
```

Nail Down Representation of Events

```
struct play {};
struct open_close {};
struct pause {};

struct cd_detected
{
    cd_detected(
      std::string name,
      std::vector<std::clock_t> const& track_lengths
     )
       : name(name), track_lengths(track_lengths)
     {}

    std::string name;
    std::vector<std::clock_t> track_lengths
};
```

FSM Declaration

```
class player              // Curiously Recurring Template Pattern
  : public state_machine<player>
{
   friend class state_machine<player>;
   enum states { Empty, Open, Stopped, Playing, Paused,
                 initial_state = Empty };

   void start_playback(play const&);    // transition actions
   void open_drawer(drawer const&);
   void close_drawer(drawer const&);
   void store_cd_info(cd_detected const&);
   void stop_playback(stop const&);
   void pause_playback(pause const&);
   void resume_playback(play const&);
   void stop_and_open(drawer const&);

   typedef mpl::vector11<...STT goes here...> transition_table;
};
```

FSM and its use (whole enchilada)

```
class player : public state_machine<player>   // player.hpp
{
 private:
    // the list of FSM states
    enum states {
        Empty, Open, Stopped, Playing, Paused
      , initial_state = Empty
    };

    void start_playback(play const&);
    void open_drawer(open_close const&);
    void close_drawer(open_close const&);
    void store_cd_info(cd_detected const&);
    void stop_playback(stop const&);
    void pause_playback(pause const&);
    void resume_playback(play const&);
    void stop_and_open(open_close const&);


    friend class state_machine<player>;
    typedef player p; // makes transition table cleaner

    // transition table
    struct transition_table : mpl::vector11<

    //     Start     Event         Next      Action
    //  +---------+-------------+---------+---------------------+
    row < Stopped , play        , Playing , &p::start_playback  >,
    row < Stopped , open_close  , Open    , &p::open_drawer     >,
    //  +---------+-------------+---------+---------------------+
    row < Open    , open_close  , Empty   , &p::close_drawer    >,
    //  +---------+-------------+---------+---------------------+
    row < Empty   , open_close  , Open    , &p::open_drawer     >,
    row < Empty   , cd_detected , Stopped , &p::store_cd_info   >,
    //  +---------+-------------+---------+---------------------+
    row < Playing , stop        , Stopped , &p::stop_playback   >,
    row < Playing , pause       , Paused  , &p::pause_playback  >,
    row < Playing , open_close  , Open    , &p::stop_and_open   >,
    //  +---------+-------------+---------+---------------------+
    row < Paused  , play        , Playing , &p::resume_playback >,
    row < Paused  , stop        , Stopped , &p::stop_playback   >,
    row < Paused  , open_close  , Open    , &p::stop_and_open   >
    //  +---------+-------------+---------+---------------------+

    > {};
};
```

```
// player.cpp

void player::start_playback(play const&){}
void player::open_drawer(open_close const&){}
void player::close_drawer(open_close const&){}
void player::store_cd_info(cd_detected const&){}
void player::stop_playback(stop const&){}
void player::pause_playback(pause const&){}
void player::resume_playback(play const&){}
void player::stop_and_open(open_close const&){}



// main.cpp

int main()
{
    player p;

    p.process_event(open_close()); // user opens CD player
    p.process_event(open_close()); // inserts CD and closes
    p.process_event(               // CD is detected
        cd_detected(
          "louie, louie"
        , std::vector<std::clock_t>( /* track lengths */ )
        )
    );
    p.process_event(play());       // etc.
    p.process_event(pause());
    p.process_event(play());
    p.process_event(stop());
    return 0;
}
```

5) Implementing the Framework

- i.e., we need to define the `state_machine<…>` class template
- Note that the definition of our FSM, player, contains **no details** about how the state machine is implemented
- We can choose among various implementation strategies.

Curiously Recurring Template Pattern

```
template <class FSM>
struct state_machine
{
    template <class FSM,
        int FromState, class Event, int ToState
      , void (FSM::*action)(Event const&) >
    struct row
    { };
        static int const from_state = FromState;
        static int const to_state = ToState;
        typedef Event event;
        typedef FSM fsm_t;

        static void execute(FSM& fsm, Event const& e)
        { (fsm.*action)(e); }

};
```

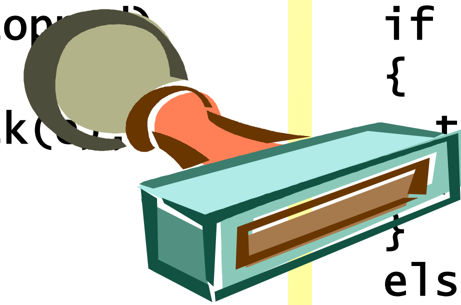
Dispatching Events (By Hand)

```
template <class FSM>
void state_machine<FSM>::process_event(play const& e)
{
   switch (this->state)
   {
    case Stopped:
        this->start_playback(e);
        this->m_state = playing;
        break;
    case Paused:
        this->resume_playback(e);
        this->m_state = playing;
        break;
    default:
        this->no_transition(e);
   }
}
```

Breakdown for Code Generation

```
void process_event(play const& e)
{
  this->state = case_Stopped(e);
}


int case_Stopped(play const& e)
{
  if (this->state == Stopped)
  {
    this->start_playback(e);
    return Playing;
  }
  else
  {
    return this->case_Paused(e);
  }
}
```



```
int case_default(play const& e)
{
  return this->no_transition(
    this->state, e);
}

int case_Paused(play const& e)
{
  if (this->state == Paused)
  {
    this->resume_playback(e);
    return Playing;
  }
  else
  {
    return this->case_default(e);
  }
}
```

Generating the Dispatch Functions

```
int
case_Stopped(play const& e)
{
  if (this->state == Stopped)
  {
    this->start_playback(e);
    return Playing;
  }
  else
  {
    return
       this->case_Paused(e);
  }
}
```

```
template <int State, class E>
int case_(E const& e)
{
   if (fsm.state == State)
   {
     this->??????(e);
     return ?????;
   }
   else
   {
     return
       this->case_<???>(e);
   }
}
```

Generating the Dispatch Functions

```
template<class Row>
struct event_dispatcher
{
  typedef typename Row::fsm_t fsm_t;
  typedef typename Row::event event;

  static int dispatch(fsm_t& fsm, int state, event const& e)
  {
    if (state == Row::from_state)
    {
      Row::execute(fsm, e);
      return Row::to_state;
    }
    else ????;
  }
};
```

“No Transition” Handler

```
struct default_event_dispatcher
{
  template<class FSM, class Event>
  static int dispatch(FSM& m, int state, Event const& e)
  {
     return m.call_no_transition(state, e);
  }
};

template <class FSM>
class state_machine
{
  template <class Event>
  int call_no_transition(int state, Event const& e)
  {
    return static_cast<FSM*>(this)->no_transition(state, e);
  }
```

Must Assemble This Dispatcher Type

```
typedef player p;

event_dispatcher<
   row<Stopped, play, Playing, &p::start_playback>,

   event_dispatcher<
      row<Paused, play, Playing, &p::resume_playback>,

      default_event_dispatcher
   >

>
```

Generating the Dispatcher

```
template <class Row, class Event>
struct has_event
  : boost::is_same<typename Row::event, Event>
{};

template<class RowSequence, class Event>
struct generate_dispatcher
  : mpl::accumulate<                    // a.k.a. fold

      // Select all rows with transitions on Event
      mpl::filter_view<RowSequence, has_event<_1,Event> >,

      // Terminating case
      default_event_dispatcher,

      // Accumulator
      event_dispatcher<_2,_1>
    >
{};
```

Finally, the Event Processor

```
template<class Event>
int process_event(Event const& event)
{
    typedef typename generate_dispatcher<
        typename FSM::transition_table, Event
    >::type dispatcher;

    this->state = dispatcher::dispatch(
        static_cast<FSM&>(*this),
        this->state,
        event
    );

    return this->state;
}
```

Final Details of state_machine<>

```
public:
   template <class Event>
   int no_transition(int state, Event const& e)
   {
       assert(false);
       return state;
   }

protected:
   state_machine()
     : state(FSM::initial_state)
   {}

private:
   int state;
```

What Have We Achieved?

- As expressive as custom (text-based) state machine compiler
- Better scalability, maintainability, type safety, expressiveness, and efficiency than State pattern.
- Efficiency comparable to hand-coded FSM
- Fully Declarative: procedural logic fully replaceable

Boost Metaprogramming Concepts & Frameworks

Other Metaprogramming Paradigms in C++

Tuple – The Quantum Sequence

```
typedef boost::tuple<
    int, char const*, bool
> t;

t s(3, “hello”, true);
```

- heterogeneous sequence of values (wave):

  `3, “hello”, true`

- also a sequence of types (particle):

  `int, char const*, bool`

Fusion/MPL Interoperability

- Every fusion tuple is a valid MPL sequence
- fusion::as_vector(*mpl-sequence-type*()) ➔ tuple

Fusion: Tuple Metaprogramming

- Primitives:
  - begin(s), end(s) → iterators
  - next(p), prior(q) → adjacent iterators
  - Some standard STL iterator operations:

    *p   p == q   p != q
- Algorithms:
  - for_each(s, unary_function) → void
  - transform(s, unary_function) → transform_view
  - push_back(s, value) → ??
  - fold(s, init, binary_function) → ??

Why Fusion Algorithms Yield Views

```
fusion::tuple<std::string, std::string>  x( “hello”,
  “world” );

fusion::for_each(
    fusion::push_back(            ← copy two strings, one int, one vector?
        fusion::push_back(            ← copy two strings, one int?
            fusion::push_back(x, 1) ← copy two strings?
          , std::vector<int>(5)
        )
      , 12
    )
  , some_function
);
```

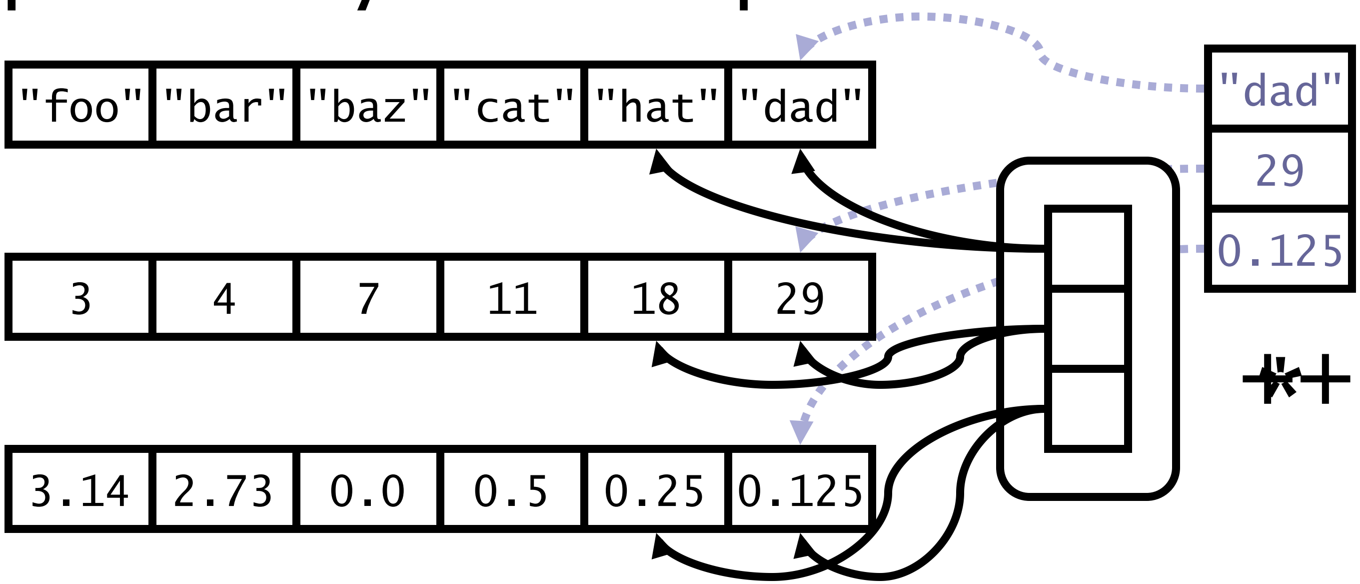
How to Get a Real Tuple

```
fusion::tuple<std::string, std::string>
    x( “hello”, “world” );

fusion::as_vector(
    fusion::push_back(
        fusion::push_back(
            fusion::push_back(x, 1)
          , std::vector<int>(5)
        )
      , 12
    )
)
```

Example: zip_iterator<IterTuple>

- Stores a tuple of iterators into different sequences
- Traverses the sequences in parallel
- operator* yields a tuple of references



Why zip_iterator?

Consider std::transform:

```
std::transform(start, finish, result, op);
std::transform(start, finish, start2, result, op);

std::transform(start, finish, start2, start3, result, op);
std::transform(start, finish, start2, start3, start4, result, op);
std::transform(start, finish, start2, start3, start4, start5, result, op);
std::transform(start, finish, start2, start3, start4, start5, start6, result, op);
std::transform(start, finish, start2, start3, start4, start5, start6, start7, result, op);
std::transform(start, finish, start2, start3, start4, start5, start6, start7, start8, result, op);
```

At some point, you want to generalize back to:

```
std::transform(zip_start, zip_finish, result, op);
```

From zip_iterator's Implementation

```
struct increment
{
    template<typename T>
    void operator()(T& x) const
    { ++x; }
};

template <class IteratorTuple>
zip_iterator<IteratorTuple>::operator++()
{
    fusion::for_each(this->m_iterators, increment());
};
```

From zip_iterator's Implementation

```
template <class IteratorTuple>
zip_iterator<IteratorTuple>::operator++()
{
    using namespace boost::phoenix;
    fusion::for_each( this->m_iterators, ++_1 );
};
```

Implementing Dereference

```
struct deref
{
  template <class> struct result;

  // return type computation
  template<class F, class Iterator>
  struct result<F(Iterator)>
  {
    typedef typename
    std::iterator_traits<
        typename remove_cv<
            typename remove_reference<
                Iterator
            >::type
        >::type
    >::reference  type;
  };

  // runtime function call
  template<typename Iter>
  typename result<deref(Iter)>::type
  operator()(Iter const& it) const
  { return *it; }
};
```

```
template <class IteratorTuple>
struct zip_iterator
{
  typedef typename
  mpl::transform<
      IteratorTuple
    , deref::result<deref(mpl::_1)>
  >::type reference;

  reference operator*() const
  {
    return fusion::transform(
         this->m_iterators
       , deref()
    );
  }
   …
   IteratorTuple m_iterators;
};
```

Fusion Tuple Metaprogramming

- Merges pure type manipulation and runtime computation
- Makes manipulation of heterogeneous collections easy
- Interoperates seamlessly with MPL
- A floor wax *and* a dessert topping.

Preprocessor Metaprogramming

- The preprocessor is a text generator

```
#define FOO(x)   foo x
FOO(bar)    ➔    foo bar
```

- You can program it

- What if you use it to generate C++ program text?

Preprocessor Abstractions

- *preprocessing token*
- 2 kinds of macros
  - *object-like*: `#define X y`
  - *function-like* `#define FOO(x) foo x`
- *macro argument*
  - Sequence of *preprocessing tokens*
  - All parens are balanced
  - No commas outside of parens

Problem: Generate Overload Set

```
tuple<> make_tuple()
{ return tuple<>(); }

template <class A0>
tuple<A0> make_tuple(A0 x0)
{ return tuple<A0>(x0); }

template <class A0, class A1>
tuple<A0,A1> make_tuple(A0 x0, A1 x1)
{ return tuple<A0,A1>(x0,x1); }

template <class A0, class A1, class A3>
tuple<A0,A1> make_tuple(A0 x0, A1 x1, A3 x3)
{ return tuple<A0,A1,A3>(x0,x1,x3); }

template <class A0, class A1, class A3, class A4>
tuple<A0,A1> make_tuple(A0 x0, A1 x1, A3 x3, A4 x4)
{ return tuple<A0,A1,A3,A4>(x0,x1,x3,x4); }
…
```

Boost.Preprocessor Solution

```
tuple<> make_tuple()
{ return tuple<>(); }

#define FUSION_make_tuple(z, n, unused)                 \
template < BOOST_PP_ENUM_PARAMS(n, class A) >           \
tuple< BOOST_PP_ENUM_PARAMS(n, A) >                     \
make_tuple( BOOST_PP_ENUM_BINARY_PARAMS(n, A, x ) )     \
{                                                       \
    return tuple< BOOST_PP_ENUM_PARAMS(n, A) >(         \
        BOOST_PP_ENUM_PARAMS(n, x)                      \
    );                                                  \
}

BOOST_PP_REPEAT_FROM_TO(
   1, FUSION_MAX_SIZE, FUSION_make_tuple, ~)

#undef FUSION_make_tuple
```

Boost.Preprocessor Solution

```
tuple<> make_tuple()
{ return tuple<>(); }

#define FUSION_make_tuple(z, n, unused)                          \
template < class A0, class A1, class A2, class A3 >  \
tuple< BOOST_PP_ENUM_PARAMS(n, A) >                   \
make_tuple( BOOST_PP_ENUM_BINARY_PARAMS(n, A, x ) )  \
{                                                     \
    return tuple< BOOST_PP_ENUM_PARAMS(n, A) >(       \
        BOOST_PP_ENUM_PARAMS(n, x)                    \
    );                                                \
}

BOOST_PP_REPEAT_FROM_TO(
   1, FUSION_MAX_SIZE, FUSION_make_tuple, ~)

#undef FUSION_make_tuple
```

Boost.Preprocessor Solution

```
tuple<> make_tuple()
{ return tuple<>(); }

#define FUSION_make_tuple(z, n, unused)                \
template < class A0, class A1, class A2, class A3 >  \
tuple< A0, A1, A2, A3 >                                \
make_tuple( BOOST_PP_ENUM_BINARY_PARAMS(n, A, x ) )  \
{                                                      \
    return tuple< BOOST_PP_ENUM_PARAMS(n, A) >(       \
        BOOST_PP_ENUM_PARAMS(n, x)                     \
    );                                                 \
}

BOOST_PP_REPEAT_FROM_TO(
  1, FUSION_MAX_SIZE, FUSION_make_tuple, ~)

#undef FUSION_make_tuple
```

Boost.Preprocessor Solution

```
tuple<> make_tuple()
{ return tuple<>(); }

#define FUSION_make_tuple(z, n, unused)                  \
template < class A0, class A1, class A2, class A3 >  \
tuple< A0, A1, A2, A3 >                               \
make_tuple( A0 x0, A1 x1, A2 x2, A3 x3 )             \
{                                                     \
    return tuple< BOOST_PP_ENUM_PARAMS(n, A) >(       \
        BOOST_PP_ENUM_PARAMS(n, x)                    \
    );                                                \
}

BOOST_PP_REPEAT_FROM_TO(
  1, FUSION_MAX_SIZE, FUSION_make_tuple, ~)

#undef FUSION_make_tuple
```

Boost.Preprocessor Solution

```
tuple<> make_tuple()
{ return tuple<>(); }

#define FUSION_make_tuple(z, n, unused)                      \
template < class A0, class A1, class A2, class A3 >  \
tuple< A0, A1, A2, A3 >                                \
make_tuple( A0 x0, A1 x1, A2 x2, A3 x3 )              \
{                                                       \
    return tuple< A0, A1, A2, A3 >(                   \
        x0, x1, x2, x3                                 \
    );                                                  \
}

BOOST_PP_REPEAT_FROM_TO(
  1, FUSION_MAX_SIZE, FUSION_make_tuple, ~)

#undef FUSION_make_tuple
```

What About Debuggability?

- A macro expansion never contains line breaks – “horizontal repetition”
- All `make_tuple` overloads ultimately generated by a single `BOOST_PP_REPEAT`
- Resulting code all generated on that line

  template <class A0> tuple<A0> make_tuple(A0 x0) { return tuple<A0>(x0); } template <class A0, class A1> …
- Stepping through make_tuple in debugger

```
BOOST_PP_REPEAT_FROM_TO(
   1, FUSION_MAX_SIZE, FUSION_make_tuple, ~)
```

Vertical Repetition

```
#ifndef FUSION_MAKE_TUPLE_HPP    // #include guard
#  define FUSION_MAKE_TUPLE_HPP

tuple<> make_tuple() { return tuple<>(); }

#define BOOST_PP_ITERATION_LIMITS ( 1, FUSION_MAX_SIZE - 1)
#define BOOST_PP_FILENAME_1  "make_tuple_overload.hpp"
#include BOOST_PP_ITERATE()

#endif // FUSION_MAKE_TUPLE_HPP
```

```
// make_tuple_overload.hpp
#define n BOOST_PP_ITERATION()
template < BOOST_PP_ENUM_PARAMS(n, class A) >
tuple< BOOST_PP_ENUM_PARAMS(n, A) >
make_tuple( BOOST_PP_ENUM_BINARY_PARAMS(n, A, x ) )
{
    return tuple< BOOST_PP_ENUM_PARAMS(n, A) >(
        BOOST_PP_ENUM_PARAMS(n, x)
    );
}
```

Self-Iteration

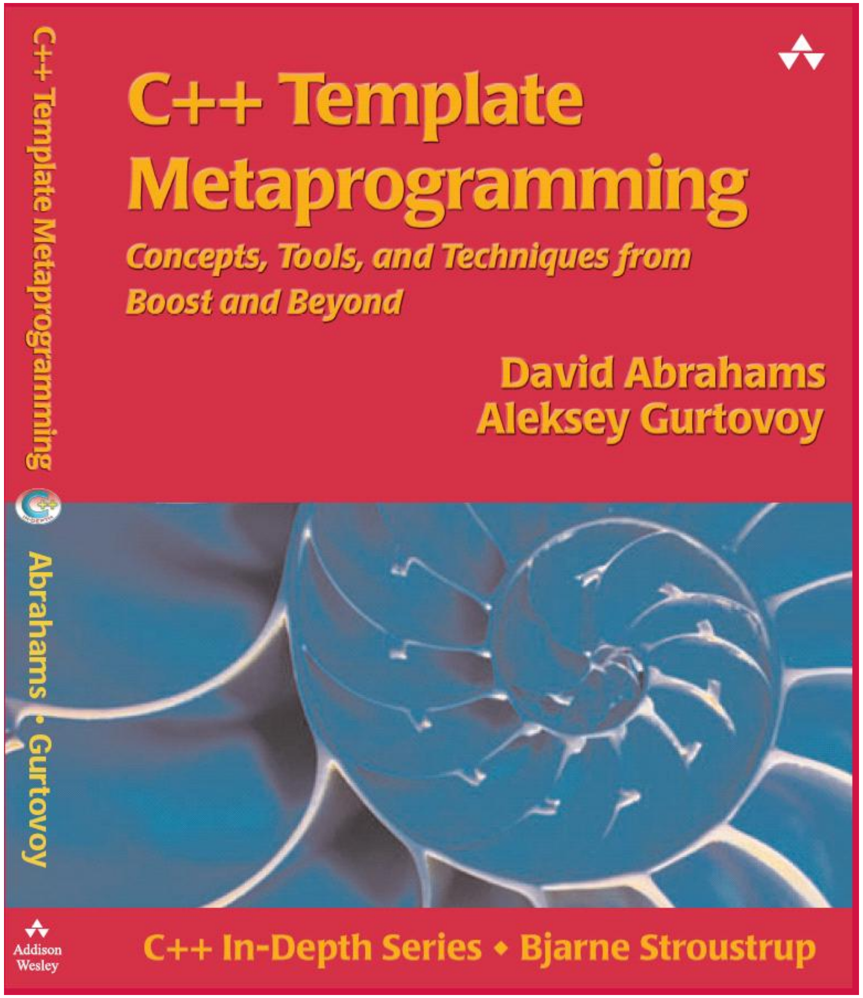
```
#ifndef BOOST_PP_IS_ITERATING
#  ifndef FUSION_MAKE_TUPLE_HPP
#    define FUSION_MAKE_TUPLE_HPP
tuple<> make_tuple() { return tuple<>(); }

#define BOOST_PP_ITERATION_LIMITS( 1, FUSION_MAX_SIZE - 1)
#define BOOST_PP_FILENAME_1  ”make_tuple.hpp"
#include BOOST_PP_ITERATE()
#  endif // FUSION_MAKE_TUPLE_HPP
#else
#  define n BOOST_PP_ITERATION()
template < BOOST_PP_ENUM_PARAMS(n, class A) >
tuple< BOOST_PP_ENUM_PARAMS(n, A) >
make_tuple( BOOST_PP_ENUM_BINARY_PARAMS(n, A, x ) )
{
    return tuple< BOOST_PP_ENUM_PARAMS(n, A) >(
        BOOST_PP_ENUM_PARAMS(n, x)
    );
}
#endif
```

Other Boost.Preprocessor Facilities

- Sequences: `(foo bar)(baz)(1 + 2)`
- Sequence Algorithms:
```
BOOST_PP_SEQ_FOR_EACH( f, data, seq )
BOOST_PP_SEQ_FOLD_LEFT( f, data, seq )
```
- Token Arithmetic: **`BOOST_PP_ADD`**`(3, 4)` ➔ 7
- Token Logic: **`BOOST_PP_AND`**`( 1, 0 )` ➔ 0
- Like MPL, Boost.Preprocessor is a complete programming system.

Further Reading



- Erwin Unruh, Prime number computation. ANSI X3J16-94-0075/ISO WG21-462. 1994.
- Todd Veldhuizen. Using C++ Template Metaprograms, *C++ Report*, SIGS Publications Inc., ISSN 1040-6042, Vol. 7, No. 4, pp. 36-43. May 1995.
- Krzysztof Czarnecki, Ulrich Eisenecker, Metalisp. http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm
- Todd Veldhuizen. *Active Libraries and Universal Languages.* Doctoral Dissertation, Indiana University Computer Science, 17 May 2004. http://www.cs.chalmers.se/~tveldhui/papers/2004/dissertation.pdf
- Todd Veldhuizen. Blitz++. http://www.oonumerics.org/blitz/.