# METAPROGRAMMING IN C++14 (AND BEYOND)

LOUIS DIONNE, ACCU 2017

# A BIT OF HISTORY

# IT ALL STARTED WITH TEMPLATES

```
1 template <typename T>
2 struct vector { /* ... */ };
3
4 int main() {
5   vector<int> ints = {1, 2, 3};
6   vector<string> strings = {"foo", "bar", "baz"};
7 }
8
```

WE SUSPECTED THEY WERE HIDING SOMETHING MORE POWERFUL

IT WASN'T CLEAR UNTIL SOMEONE CAME UP WITH A VERY SPECIAL PROGRAM

# MARCH 1994, SAN DIEGO MEETING

# ERWIN UNRUH COMES UP WITH THIS:

```
1 template <int i> struct D { D(void*); operator int(); };
2
3 template <int p, int i> struct is_prime {
4   enum { prim = (p%i) && is_prime<(i > 2 ? p : 0), i -1> :: prim };
5 };
6
7 template < int i > struct Prime_print {
8   Prime_print<i-1> a;
9   enum { prim = is_prime<i, i-1>::prim };
10   void f() { D<i> d = prim; }
11 };
12
13 struct is_prime<0,0> { enum {prim=1}; };
14 struct is_prime<0,1> { enum {prim=1}; };
15 struct Prime_print<2> { enum {prim = 1}; void f() { D<2> d = prim; } };
16 #ifndef LAST
17 #define LAST 10
18 #endif
19 main () { Prime_print<LAST> a; }
20
```

(source: http://www.erwin-unruh.de/primorig.html)

# IT PRINTS PRIME NUMBERS AT COMPILE-TIME

```
P:\HC\D386_O> hc3 i primes.cpp -DLAST=30

MetaWare High C/C++ Compiler R2.6
(c) Copyright 1987-94, MetaWare Incorporated
E "primes.cpp",L16/C63(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<2>´ [...]
-- Detected during instantiation of Prime_print<30> [...]
E "primes.cpp",L11/C25(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<3>´ [...]
-- Detected during instantiation of Prime_print<30> [...]
E "primes.cpp",L11/C25(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<5>´ [...]
-- Detected during instantiation of Prime_print<30> [...]
E "primes.cpp",L11/C25(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<7>´ [...]
-- Detected during instantiation of Prime_print<30> [...]
E "primes.cpp",L11/C25(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<11>´ [...]
-- Detected during instantiation of Prime_print<30> [...]
E "primes.cpp",L11/C25(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<13>´ [...]

[...]

```

# FAST FORWARD TO 2001

# ANDREI ALEXANDRESCU PUBLISHES MODERN C++ DESIGN

# INTRODUCES THE LOKI LIBRARY, WHICH INCLUDES Typelist

```
1 template <class T, class U>
2 struct Typelist {
3    typedef T Head;
4    typedef U Tail;
5 };
6
7 using Types = LOKI_TYPELIST_4(int, char, float, void);
8 using Second = Loki::TL::TypeAt<Types, 2>::Result;
9 // -> float
10
```

# SEVERAL ALGORITHMS ON `Typelist` ARE PROVIDED

```
1 using Types = LOKI_TYPELIST_6(int, char, float, char, void, float);
2
3 using NoChar = Loki::TL::EraseAll<Types, char>::Result;
4 // -> LOKI_TYPELIST_4(int, float, void, float)
5
6 using Uniqued = Loki::TL::NoDuplicates<Types>::Result;
7 // -> LOKI_TYPELIST_4(int, char, float, void)
8
9 using Reversed = Loki::TL::Reverse<Types>::Result;
10 // -> LOKI_TYPELIST_6(float, void, char, float, char, int)
11
12 // etc...
13
```

# THE NOTION OF COMPILE-TIME ALGORITHMS AND DATA STRUCTURES STARTS TO EMERGE

# 2004

# D. ABRAHAMS AND A. GURTOVOY PUBLISH THE MPL BOOK

The book is actually called
*C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*

IT MAKES A THOROUGH TREATMENT OF METAPROGRAMMING THROUGH THE BOOST MPL LIBRARY

# THE LIBRARY CONTAINS SEVERAL META DATA STRUCTURES

- `boost::mpl::vector`
- `boost::mpl::list`
- `boost::mpl::map`
- `boost::mpl::set`
- `boost::mpl::string`

# IT ALSO PROVIDES SEVERAL GENERIC ALGORITHMS WORKING ON META-ITERATORS, LIKE THE STL

- `boost::mpl::equal`
- `boost::mpl::transform`
- `boost::mpl::remove_if`
- `boost::mpl::sort`
- `boost::mpl::partition`
- etc...

# FOR EXAMPLE

```
1 using Types = mpl::vector<int, void, char, long, void>;
2
3 using NoVoid = mpl::remove_if<Types, std::is_void<mpl::_1>>::type;
4 // -> mpl::vector<int, char, long>
5
6 using Ptrs = mpl::transform<Types, std::add_pointer<mpl::_1>>::type;
7 // -> mpl::vector<int*, void*, char*, long*, void*>
8
```

2008

J. DE GUZMAN, D. MARSDEN AND T. SCHWINGER RELEASE THE BOOST FUSION LIBRARY

MPL ALLOWS MANIPULATING TYPES (AT COMPILE-TIME)

FUSION ALLOWS MANIPULATING OBJECTS (AT COMPILE-TIME)

# LIKE MPL, IT PROVIDES DATA STRUCTURES

- `boost::fusion::vector`
- `boost::fusion::list`
- `boost::fusion::set`
- `boost::fusion::map`

# AND ALGORITHMS

- `boost::fusion::remove_if`
- `boost::fusion::find_if`
- `boost::fusion::count_if`
- `boost::fusion::transform`
- `boost::fusion::reverse`
- etc...

# FOR EXAMPLE

```
// vector
auto vector = fusion::make_vector(1, 2.2f, "hello"s, 3.4, 'x');
auto no_floats = fusion::remove_if<
                        std::is_floating_point<mpl::_>>(vector);

assert(no_floats == fusion::make_vector(1, "hello"s, 'x'));

```

```
// map
struct a; struct b; struct c;
auto map = fusion::make_map<a, b, c>(1, 'x', "hello"s);

assert(fusion::at_key<a>(map) == 1);
assert(fusion::at_key<b>(map) == 'x');
assert(fusion::at_key<c>(map) == "hello"s);

```

BOOSTCON 2010

MATT CALABRESE AND ZACH LAINE PRESENT INSTANTIATIONS MUST GO

THEY INTRODUCE A WAY OF METAPROGRAMMING WITHOUT ANGLY BRACKETS

THE IDEA IS KINDA SHOT DOWN AND NOBODY FOLLOWS UP

...UNTIL HANA

# BASIC IDEA:

# REPRESENT COMPILE-TIME ENTITIES AS OBJECTS, NOT TYPES

# HANA PROVIDES

- data structures like Boost.Fusion
- algorithms like Boost.Fusion
- a way to represent types as values

# ALL YOU NEED FROM MPL AND FUSION IN A SINGLE LIBRARY

# DATA STRUCTURES

- `boost::hana::tuple`
- `boost::hana::map`
- `boost::hana::set`

# ALGORITHMS

- `boost::hana::remove_if`
- `boost::hana::find_if`
- `boost::hana::count_if`
- `boost::hana::transform`
- `boost::hana::reverse`
- etc…

# UTILITIES

- `boost::hana::type`
- `boost::hana::integral_constant`
- `boost::hana::string`

# MPL

```
using Types = mpl::vector<int, void, char, long, void>;

using NoVoid = mpl::remove_if<Types, std::is_void<mpl::_1>>::type;
// -> mpl::vector<int, char, long>

using Ptrs = mpl::transform<Types, std::add_pointer<mpl::_1>>::type;
// -> mpl::vector<int*, void*, char*, long*, void*>

```

# HANA

```
auto Types = hana::tuple_t<int, void, char, long, void>;

auto NoVoid = hana::remove_if(Types, hana::traits::is_void);
// -> hana::tuple_t<int, char, long>

auto Ptrs = hana::transform(Types, hana::traits::add_pointer);
// -> hana::tuple_t<int*, void*, char*, long*, void*>

```

# FUSION

```
// vector
auto vector = fusion::make_vector(1, 2.2f, "hello"s, 3.4, 'x');
auto no_floats = fusion::remove_if<
                      std::is_floating_point<mpl::_>>(vector);

assert(no_floats == fusion::make_vector(1, "hello"s, 'x'));

```

# HANA

```
// tuple
auto tuple = hana::make_tuple(1, 2.2f, "hello"s, 3.4, 'x');
auto no_floats = hana::remove_if(tuple, [](auto const& t) {
  return hana::traits::is_floating_point(hana::typeid_(t));
});

assert(no_floats == hana::make_tuple(1, "hello"s, 'x'));

```

# FUSION

```
// map
struct a; struct b; struct c;
auto map = fusion::make_map<a, b, c>(1, 'x', "hello"s);

assert(fusion::at_key<a>(map) == 1);
assert(fusion::at_key<b>(map) == 'x');
assert(fusion::at_key<c>(map) == "hello"s);

```

# HANA

```
// map
struct a; struct b; struct c;
auto map = hana::make_map(
  hana::make_pair(hana::type<a>{}, 1),
  hana::make_pair(hana::type<b>{}, 'x'),
  hana::make_pair(hana::type<c>{}, "hello"s)
);

assert(map[hana::type<a>{}] == 1);
 assert(map[hana::type<b>{}] == 'x');
 assert(map[hana::type<c>{}] == "hello"s);

```

# HOW CAN I ACTUALLY USE THIS?

# EXAMPLE: PARSER COMBINATORS

```
1  int main() {
2    auto parser = combine_parsers(
3      lit('(') , parse<int>()           ,
4      lit(',') , parse<std::string>()   ,
5      lit(',') , parse<double>()        ,
6      lit(')')
7    );
8
9    std::istringstream text{"(1, foo, 3.3)"};
10    hana::tuple<int, std::string, double> data = parser(text);
11
12    assert(data == hana::make_tuple(1, "foo", 3.3));
13 }
14
```

# PRIMER: COMPILE-TIME TYPE INFORMATION

```
1 constexpr auto Int = hana::typeid_(3);
2 // -> type<int>
3
4 constexpr auto IntPtr = hana::traits::add_pointer(Int);
5 // -> type<int*>
6
7 static_assert(IntPtr == hana::type<int*>{});
8
```

# HOW THAT WORKS

```
1  template <typename T>
2  struct type { };
3
4  template <typename T>
5  constexpr type<T*> add_pointer(type<T>)
6  { return {}; }
7
8  template <typename T, typename U>
9  constexpr std::false_type operator==(type<T>, type<U>)
10 { return {}; }
11
12 template <typename T>
13 constexpr std::true_type operator==(type<T>, type<T>)
14 { return {}; }
15
```

# BASIC PARSER

```
1 template <typename T>
2 struct parser {
3   T operator()(std::istream& in) const {
4     T result;
5     in >> result;
6     return result;
7   }
8 };
9
10 template <typename T>
11 parser<T> parse() { return {}; }
12
```

# LITERAL PARSER

```
1 struct void_ { };
2
3 struct literal_parser {
4   char c;
5   void_ operator()(std::istream& in) const {
6     in.ignore(1, c);
7     return {};
8   }
9 };
10
11 literal_parser lit(char c) { return {c}; }
12
```

# COMBINING PARSERS

```
1 template <typename ...Parsers>
2 auto combine_parsers(Parsers const& ...parsers) {
3   return [=](std::istream& in) {
4     auto all = hana::make_tuple(parsers(in)...);
5     auto result = hana::remove_if(all, [](auto const& result) {
6       return hana::typeid_(result) == hana::type<void_>{};
7     });
8     return result;
9   };
10 }
11
```

# EXAMPLE: DIMENSIONAL ANALYSIS

```
1 double m = 10.3;  // mass in kg
2 double d = 3.6;   // distance in meters
3 double t = 2.4;   // time delta in seconds
4 double v = d / t; // speed in m/s
5 double a = ...;   // acceleration in m/s²
6
7 double force = m * v; // What's wrong?
8
```

# SOLUTION: ATTACH UNITS TO QUANTITIES

```
1 quantity<mass>          m{10.3};
2 quantity<length>        d{3.6};
3 quantity<time_>         t{2.4};
4 quantity<velocity>      v{d / t};
5 quantity<acceleration> a{3.9};
6 quantity<force>         f{m * v}; // Compiler error!
7 quantity<force>         f{m * a}; // Works as expected
8
```

# PRIMER: COMPILE-TIME INTEGERS

```
1 auto three = 1 + 2;
2 // -> int
3 static_assert(three == 3);
4
5 auto three = 1_c + 2_c;
6 // -> integral_constant<int, 3>
7 static_assert(three == 3_c);
8
```

# HOW THAT WORKS

```
1 template <typename T, T v>
2 struct integral_constant {
3   static constexpr T value = v;
4   constexpr operator T() const noexcept { return value; }
5   // etc...
6 };
7
8 template <char ...c>
9 constexpr auto operator"" _c() {
10   constexpr int n = parse<c...>();
11   return integral_constant<int, n>{};
12 }
13
14 template <typename T, T x, T y>
15 constexpr auto operator+(integral_constant<T, x>,
16                          integral_constant<T, y>)
17 { return integral_constant<T, x + y>{}; }
18
```

# REPRESENTING QUANTITIES

```
1 template <typename Dimensions>
2 struct quantity {
3   double value_;
4   explicit quantity(double v) : value_(v) { }
5   explicit operator double() const { return value_; }
6 };
7
```

# REPRESENTING DIMENSIONS

```
1 // Note: tuple_c<int, x...> is tuple<integral_constant<int, x>...>
2
3 // base dimensions                                M  L  T  I  K  J  N
4 using mass        = decltype(hana::tuple_c<int, 1, 0, 0, 0, 0, 0, 0>);
5 using length      = decltype(hana::tuple_c<int, 0, 1, 0, 0, 0, 0, 0>);
6 using time_       = decltype(hana::tuple_c<int, 0, 0, 1, 0, 0, 0, 0>);
7 using charge      = decltype(hana::tuple_c<int, 0, 0, 0, 1, 0, 0, 0>);
8 using temperature = decltype(hana::tuple_c<int, 0, 0, 0, 0, 1, 0, 0>);
9 using intensity   = decltype(hana::tuple_c<int, 0, 0, 0, 0, 0, 1, 0>);
10 using amount      = decltype(hana::tuple_c<int, 0, 0, 0, 0, 0, 0, 1>);
11
12 // composite dimensions
13 using velocity     = decltype(hana::tuple_c<int, 0, 1, -1, 0, 0, 0, 0>); // M/
14 using acceleration = decltype(hana::tuple_c<int, 0, 1, -2, 0, 0, 0, 0>); // M/
15 using force        = decltype(hana::tuple_c<int, 1, 1, -2, 0, 0, 0, 0>); // MI
16
```

# CATCHING ERRORS

```
1  template <typename OtherDimensions>
2  explicit quantity(quantity<OtherDimensions> other)
3    : value_(other.value_)
4  {
5    static_assert(Dimensions{} == OtherDimensions{},
6      "Constructing quantities with incompatible dimensions!");
7  }
8
```

# COMPOSING DIMENSIONS

```
1  template <typename D1, typename D2>
2  auto operator*(quantity<D1> a, quantity<D2> b) {
3    using D = decltype(hana::zip_with(std::plus<>{}, D1{}, D2{}));
4    return quantity<D>{static_cast<double>(a) * static_cast<double>(b)};
5  }
6
7  template <typename D1, typename D2>
8  auto operator/(quantity<D1> a, quantity<D2> b) {
9    using D = decltype(hana::zip_with(std::minus<>{}, D1{}, D2{}));
10    return quantity<D>{static_cast<double>(a) / static_cast<double>(b)};
11 }
12
```

# EXAMPLE: A SIMPLE EVENT SYSTEM

```
1  int main() {
2    event_system events{{"foo", "bar", "baz"}};
3
4    events.on("foo", []() { std::cout << "foo triggered!" << '\n'; });
5    events.on("foo", []() { std::cout << "foo again!"     << '\n'; });
6    events.on("bar", []() { std::cout << "bar triggered!" << '\n'; });
7    events.on("baz", []() { std::cout << "baz triggered!" << '\n'; });
8
9    events.trigger("foo");
10    events.trigger("baz");
11    // events.trigger("unknown"); // WOOPS! Runtime error!
12 }
13
```

# WHAT IF

- All events are known at compile-time
- We always know what event to trigger at compile-time

# COULD WE DO BETTER?

```
1  int main() {
2    auto events = make_event_system("foo"_s, "bar"_s, "baz"_s);
3
4    events.on("foo"_s, []() { std::cout << "foo triggered!" << '\n'; });
5    events.on("foo"_s, []() { std::cout << "foo again!" << '\n'; });
6    events.on("bar"_s, []() { std::cout << "bar triggered!" << '\n'; });
7    events.on("baz"_s, []() { std::cout << "baz triggered!" << '\n'; });
8    // events.on("unknown"_s, []() { }); // compiler error!
9
10    events.trigger("foo"_s); // no overhead for event lookup
11    events.trigger("baz"_s);
12    // events.trigger("unknown"_s); // compiler error!
13 }
14
```

# PRIMER: COMPILE-TIME STRINGS

```
1 auto Hello_world = "hello"_s + " world"_s;
2 static_assert(Hello_world == "hello world"_s);
3
```

# HOW THAT WORKS

```
1  template <char ...c> struct string { };
2
3  template <typename CharT, CharT ...c>
4  constexpr string<c...> operator"" _s() { return {}; }
5
6  template <char ...c1, char ...c2>
7  constexpr auto operator==(string<c1...>, string<c2...>) {
8    return std::is_same<string<c1...>, string<c2...>>{};
9  }
10
11 template <char ...c1, char ...c2>
12 constexpr auto operator+(string<c1...>, string<c2...>) {
13   return string<c1..., c2...>{};
14 }
15
```

# RUNTIME

```
struct event_system {
  using Callback = std::function<void()>;
  std::unordered_map<std::string, std::vector<Callback>> map_;

```

# COMPILE-TIME

```
template <typename ...Events>
struct event_system {
  using Callback = std::function<void()>;
  hana::map<hana::pair<Events, std::vector<Callback>>...> map_;

```

# RUNTIME

```
explicit event_system(std::initializer_list<std::string> events) {
  for (auto const& event : events)
    map_.insert({event, {}});
}

```

# COMPILE-TIME

```
template <typename ...Events>
event_system<Events...> make_event_system(Events ...events) {
  return {};
}

```

# RUNTIME

```
template <typename F>
void on(std::string const& event, F callback) {
  auto callbacks = map_.find(event);
  assert(callbacks != map_.end() &&
    "trying to add a callback to an unknown event");

  callbacks->second.push_back(callback);
}

```

# COMPILE-TIME

```
template <typename Event, typename F>
void on(Event e, F callback) {
  auto is_known_event = hana::contains(map_, e);
  static_assert(is_known_event,
    "trying to add a callback to an unknown event");

  map_[e].push_back(callback);
}

```

# RUNTIME

```
void trigger(std::string const& event) const {
  auto callbacks = map_.find(event);
  assert(callbacks != map_.end() &&
    "trying to trigger an unknown event");

  for (auto& callback : callbacks->second)
    callback();
}

```

# COMPILE-TIME

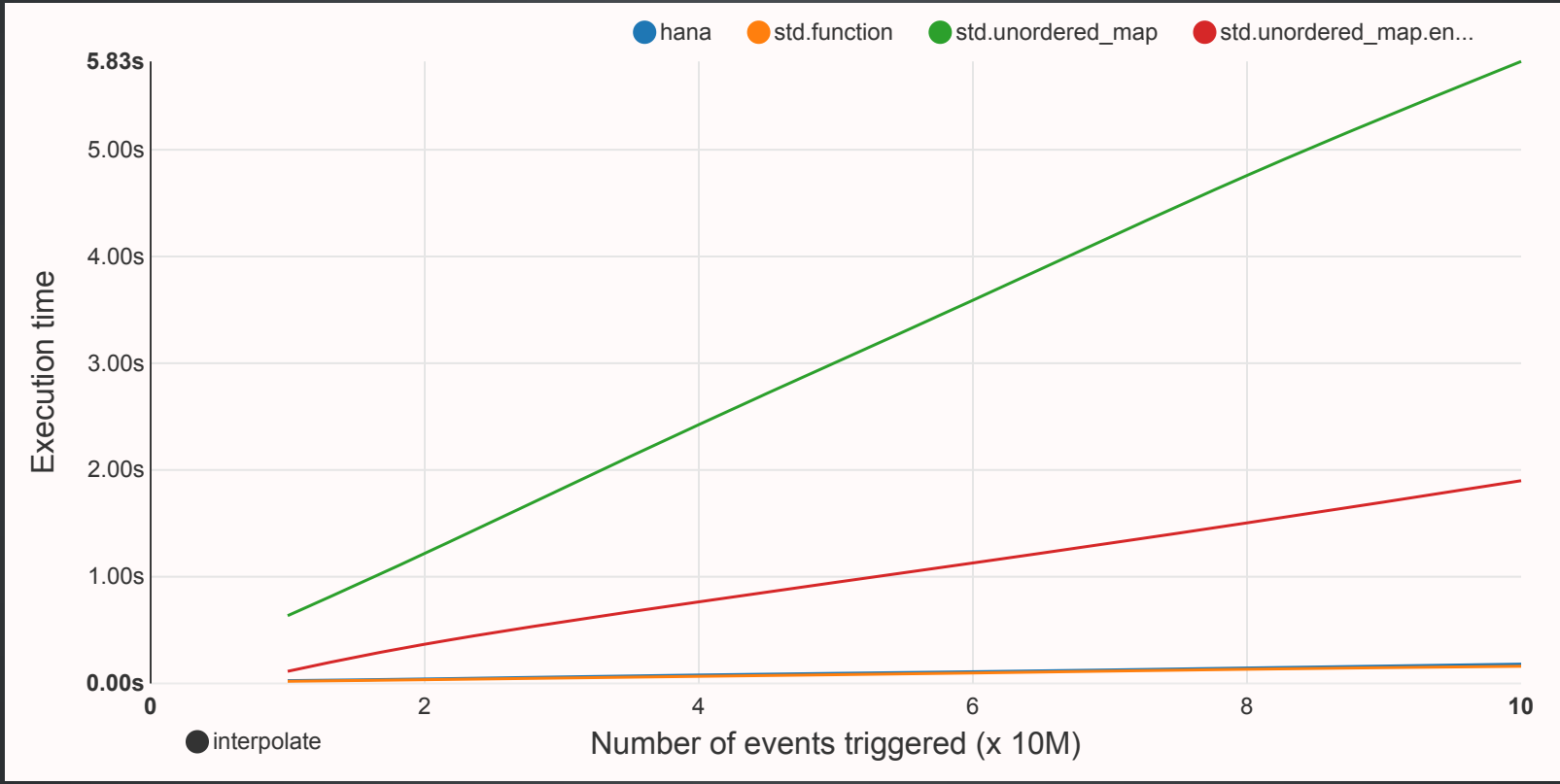
```
template <typename Event>
void trigger(Event e) const {
  auto is_known_event = hana::contains(map_, e);
  static_assert(is_known_event,
    "trying to trigger an unknown event");

  for (auto& callback : map_[e])
    callback();
}

```

# BUT DOES IT ACTUALLY MATTER?

# COMPILED WITH `-O3 -flto`



# WHAT IF THE EVENT TO TRIGGER CAN BE DECIDED AT RUNTIME?

```
1 int main() {
2   auto events = make_event_system("foo"_s, "bar"_s, "baz"_s);
3
4   events.on("foo"_s, []() { std::cout << "foo triggered!" << '\n'; });
5   events.on("foo"_s, []() { std::cout << "foo again!" << '\n'; });
6   events.on("bar"_s, []() { std::cout << "bar triggered!" << '\n'; });
7   events.on("baz"_s, []() { std::cout << "baz triggered!" << '\n'; });
8
9   std::string e = read_from_stdin();
10   events.trigger(e);
11 }
12
```

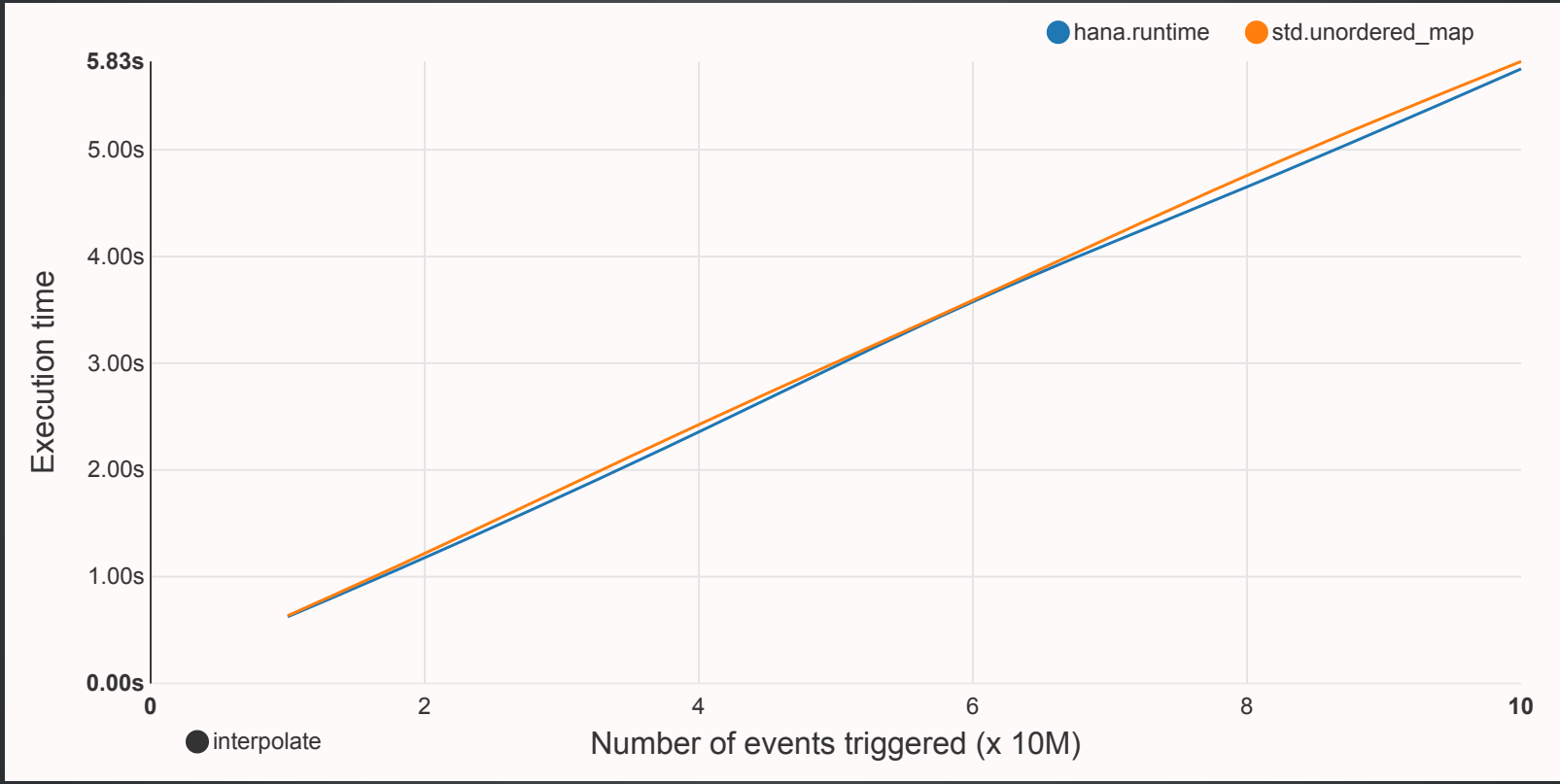
# FIRST, MAINTAIN A DYNAMIC MAP

```
1 std::unordered_map<std::string, std::vector<Callback>* const> dynamic_;
2
3 event_system() {
4   hana::for_each(hana::keys(map_), [&](auto event) {
5     dynamic_.insert({event.c_str(), &map_[event]});
6   });
7 }
8
```

# THEN, OVERLOAD `trigger`!

```
1 void trigger(std::string const& e) const {
2   auto callbacks = dynamic_.find(e);
3   assert(callbacks != dynamic_.end() &&
4     "trying to trigger an unknown event");
5
6   for (auto& callback : *callbacks->second)
7     callback();
8 }
9
```

# AND WHAT ABOUT PERFORMANCE?



# HANA SHINES WHEN COMBINING COMPILE-TIME AND RUNTIME

# EXAMPLE: GENERATING JSON USING LIMITED REFLECTION

```
1 Person joe{"Joe", 30};
2 std::cout << to_json(hana::make_tuple(1, 'c', joe));
3
```

## Output:

```
1 [1, "c", {"name" : "Joe", "age" : 30}]
2
```

# DEFINE YOUR TYPE LIKE THIS

```
1 struct Person {
2   BOOST_HANA_DEFINE_STRUCT(Person,
3     (std::string, name),
4     (int, age)
5   );
6 };
7
```

(non-intrusive version is `BOOST_HANA_ADAPT_STRUCT`)

# HANDLE BASE TYPES

```
1 std::string quote(std::string s) { return "\"" + s + "\""; }
2
3 template <typename T>
4 auto to_json(T const& x) -> decltype(std::to_string(x)) {
5   return std::to_string(x);
6 }
7
8 std::string to_json(char c) { return quote({c}); }
9 std::string to_json(std::string s) { return quote(s); }
10
```

# HANDLE Sequences

```
1 template <typename Xs>
2   std::enable_if_t<hana::is_a<hana::Sequence, Xs>(),
3 std::string> to_json(Xs const& xs) {
4   auto json = hana::transform(xs, [](auto const& x) {
5     return to_json(x);
6   });
7
8   return "[" + join(std::move(json), ", ") + "]";
9 }
10
```

# HANDLE Structs

```
1 template <typename T>
2   std::enable_if_t<hana::is_a<hana::Struct, T>(),
3 std::string> to_json(T const& x) {
4   auto json = hana::transform(keys(x), [&](auto name) {
5     auto const& member = hana::at_key(x, name);
6     return quote(name.c_str()) + " : " + to_json(member);
7   });
8
9   return "{" + join(std::move(json), ", ") + "}";
10 }
11
```

# THE FUTURE

# HOW WOULD WE WANT METAPROGRAMMING TO LOOK LIKE?

# CONSIDER SERIALIZATION TO JSON

```
1 struct point { float x, y, z; };
2 struct triangle { point a, b, c; };
3
4 struct tetrahedron {
5   triangle base;
6   point apex;
7 };
8
9 int main() {
10    tetrahedron t{
11      {{0.f,0.f,0.f}, {1.f,0.f,0.f}, {0.f,0.f,1.f}},
12      {0.f,1.f,0.f}
13    };
14
15    to_json(std::cout, t);
16 }
17
```

# SHOULD OUTPUT

```
{
  "base": {
    "a": {"x": 0, "y": 0, "z": 0},
    "b": {"x": 1, "y": 0, "z": 0},
    "c": {"x": 0, "y": 0, "z": 1}
  },
  "apex": {"x": 0, "y": 1, "z": 0}
}

```

# HOW TO WRITE THIS `to_json`?

# EASY WITH REFLECTION AND TUPLE FOR-LOOPS

## SYNTAX TBD

```
template <typename T>
std::ostream& to_json(std::ostream& out, T const& v) {
  if constexpr (std::meta::Record(reflexpr(T))) {
    out << "{";
    constexpr auto members = reflexpr(T).members();
    for constexpr (int i = 0; i != members.size(); ++i) {
      if (i > 0) out << ", ";
      out << '"' << members[i].name() << "\": ";
      to_json(out, v.*members[i].pointer());
     }
     out << '}';
  } else {
    out << v;
  }
  return out;
}
```

# THE FUTURE OF TYPE-LEVEL COMPUTATIONS?

```
1  constexpr std::vector<std::meta::type>
2  sort_by_alignment(std::vector<std::meta::type> types) {
3    std::sort(v.begin(), v.end(), [](std::meta::type t,
4                                     std::meta::type u) {
5      return t.alignment() < u.alignment();
6    });
7    return v;
8  }
9
10 constexpr std::vector<std::meta::type> types{
11   reflexpr(Foo), reflexpr(Bar), reflexpr(Baz)
12 };
13
14 constexpr std::vector<std:meta::type> sorted = sort_by_alignment(types);
15
16 std::tuple<typename(sorted)...> tuple{...};
17
```

# STEPS TO GET THERE

# EXPAND CONSTEXPR EVALUATION TO ALLOW SOME ALLOCATIONS

constexpr variable size sequences (basically `std::vector`)

# CREATE `std::meta::type` (OR EQUIVALENT)

- Basically compile-time RTTI
- Pointer to an AST node inside the compiler

# CONVERT FROM `std::meta::type` TO C++ TYPE

Allows influencing types in our program based on the result of type-level computations

# UNPACK A CONSTEXPR SEQUENCE INTO A PARAMETER PACK

- Gets us `std::tuple<typename(sorted)...>`
- Technically not needed (could expand using `std::index_sequence`), but probably desirable

# METAPROGRAMMING IS POWERFUL

WE NEED MORE METAPROGRAMMING

BUT LESS *TEMPLATE* METAPROGRAMMING

# LET'S EMBRACE THIS REALITY


https://a9.com/careers

# BONUS

- Go interfaces
- Haskell typeclasses
- Rust traits
- C++0x concept maps

NAME THEM HOWEVER YOU'D LIKE

```
1 struct Square {
2   void draw(std::ostream& out) const { out << "Square"; }
3 };
4
5 struct Circle {
6   void draw(std::ostream& out) const { out << "Circle"; }
7 };
8
9 void f(drawable const& d) {
10   d.draw(std::cout);
11 }
12
13 f(Square{}); // prints "Square"
14 f(Circle{}); // prints "Circle"
15
```

# DEFINE THE INTERFACE

```
1 struct Drawable : decltype(dyno::requires(
2   "draw"_s = dyno::function<void (dyno::T const&, std::ostream&)>
3 )) { };
4
```

# DEFINE HOW THE INTERFACE IS FULFILLED

```
template <typename T>
auto dyno::concept_map_for<Drawable, T> = dyno::make_concept_map(
  "draw"_s = [](T const& self, std::ostream& out) { self.draw(out); }
);

```

# DEFINE AN WRAPPER FOR THINGS THAT SATISFY THE CONCEPT

```
1  struct drawable {
2    template <typename T>
3    drawable(T x) : poly_{x} { }
4
5    void draw(std::ostream& out) const
6    { poly_.virtual_("draw"_s)(poly_, out); }
7
8  private:
9    dyno::poly<Drawable> poly_;
10 };
11
```

# HOW DOES IT WORK?

IT'S SIMPLE

NO, NOT REALLY

# BUT BITS OF IT ARE

# HOW WE CREATE THE VTABLE

```
struct Drawable : decltype(dyno::requires(
  "draw"_s = dyno::function<void (dyno::T const&, std::ostream&)>
)) { };

```

```
template <typename ...Name, typename ...Signature>
auto requires(hana::pair<Name, hana::type<Signature>> ...f)
  -> hana::map<hana::pair<Name, Signature>...>
;

using VTable = decltype(requires(
  hana::make_pair(
    "draw"_s,
    hana::type<void (void const*, std::ostream&)>{}
  )
));

```

# HOW WE FILL IT

```
template <typename T>
auto dyno::concept_map_for<Drawable, T> = dyno::make_concept_map(
  "draw"_s = [](T const& self, std::ostream& out) { self.draw(out); }
);

```

```
template <typename ...Name, typename ...Function>
auto make_concept_map(hana::pair<Name, Function> ...f) {
  return hana::make_map(f...);
}

template <typename T>
auto functions = make_concept_map(
  hana::make_pair(
    "draw"_s,
     [](T const& self, std::ostream& out) { self.draw(out); }
   )
);

```

# HOW WE BIND THE TWO TOGETHER

```
1 struct drawable {
2   template <typename T>
3   drawable(T t) : vtable_{functions<T>}, ... { }
4
5   ...
6
7 private:
8   VTable vtable_;
9   ...
10 };
11
```