FUN WITH BOOST.HANA

LOUIS DIONNE, C++NOW 2017

QUICK PRIMER ON HANA

HANA PROVIDES

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

DATA STRUCTURES

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

ALGORITHMS

- `hana::remove_if`
- `hana::find_if`
- `hana::count_if`
- `hana::transform`
- `hana::reverse`
- etc...

UTILITIES

- `hana::type`
- `hana::integral_constant`
- `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, [](auto t) {
  return hana::traits::is_void(t);
});
// -> hana::tuple_t<int, char, long>

auto Ptrs = hana::transform(Types, [](auto t) {
  return hana::traits::add_pointer(t);
});
// -> 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);
```

NOW, LET'S HAVE FUN

EXAMPLE: PARSER COMBINATORS

```
int main() {
  auto parser = combine_parsers(
    lit('(') , parse<int>()           ,
    lit(',') , parse<std::string>()   ,
    lit(',') , parse<double>()        ,
    lit(')')
  );

  std::istringstream text{"(1, foo, 3.3)"};
  hana::tuple<int, std::string, double> data = parser(text);

  assert(data == hana::make_tuple(1, "foo", 3.3));
}
```

PRIMER: COMPILE-TIME TYPE INFORMATION

```
constexpr auto Int = hana::typeid_(3);
// -> type<int>

constexpr auto IntPtr = hana::traits::add_pointer(Int);
// -> type<int*>

static_assert(IntPtr == hana::type<int*>{});
```

HOW THAT WORKS

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

template <typename T>
constexpr type<T*> add_pointer(type<T>)
{ return {}; }

template <typename T, typename U>
constexpr std::false_type operator==(type<T>, type<U>)
{ return {}; }

template <typename T>
constexpr std::true_type operator==(type<T>, type<T>)
{ return {}; }
```

BASIC PARSER

```
template <typename T>
struct parser {
  T operator()(std::istream& in) const {
    T result;
    in >> result;
    return result;
  }
};

template <typename T>
parser<T> parse() { return {}; }
```

LITERAL PARSER

```
struct void_ { };

struct literal_parser {
  char c;
  void_ operator()(std::istream& in) const {
    in.ignore(1, c);
    return {};
  }
};

literal_parser lit(char c) { return {c}; }
```

COMBINING PARSERS

```
template <typename ...Parsers>
auto combine_parsers(Parsers const& ...parsers) {
  return [=](std::istream& in) {
    hana::tuple<decltype(parsers(in))...> all{parsers(in)...};
    auto result = hana::remove_if(all, [](auto const& result) {
      return hana::typeid_(result) == hana::type<void_>{};
    });
    return result;
  };
}
```

EXAMPLE: DIMENSIONAL ANALYSIS

```
double m = 10.3;  // mass in kg
double d = 3.6;   // distance in meters
double t = 2.4;   // time delta in seconds
double v = d / t; // speed in m/s
double a = ...;   // acceleration in m/s²

double force = m * v; // What's wrong?
```

SOLUTION: ATTACH UNITS TO QUANTITIES

```
quantity<mass>          m{10.3};
quantity<length>        d{3.6};
quantity<time_>         t{2.4};
quantity<velocity>      v{d / t};
quantity<acceleration> a{3.9};
quantity<force>         f{m * v}; // Compiler error!
quantity<force>         f{m * a}; // Works as expected
```

PRIMER: COMPILE-TIME INTEGERS

```
constexpr auto three = 1 + 2;
// -> int
static_assert(three == 3);

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

HOW THAT WORKS

```
template <typename T, T v>
struct integral_constant {
  static constexpr T value = v;
  constexpr operator T() const noexcept { return value; }
  // etc...
};

template <char ...c>
constexpr auto operator"" _c() {
  constexpr int n = parse<c...>();
  return integral_constant<int, n>{};
}

template <typename T, T x, T y>
constexpr auto operator+(integral_constant<T, x>,
                         integral_constant<T, y>)
{ return integral_constant<T, x + y>{}; }
```

REPRESENTING QUANTITIES

```
template <typename Dimensions>
struct quantity {
  double value_;
  explicit quantity(double v) : value_(v) { }
  explicit operator double() const { return value_; }
};
```

REPRESENTING DIMENSIONS

```
// Note: tuple_c<int, x...> is tuple<integral_constant<int, x>...>

// base dimensions                                  M  L  T  I  K  J  N
using mass        = decltype(hana::tuple_c<int, 1, 0, 0, 0, 0, 0, 0>);
using length      = decltype(hana::tuple_c<int, 0, 1, 0, 0, 0, 0, 0>);
using time_       = decltype(hana::tuple_c<int, 0, 0, 1, 0, 0, 0, 0>);
using charge      = decltype(hana::tuple_c<int, 0, 0, 0, 1, 0, 0, 0>);
using temperature = decltype(hana::tuple_c<int, 0, 0, 0, 0, 1, 0, 0>);
using intensity   = decltype(hana::tuple_c<int, 0, 0, 0, 0, 0, 1, 0>);
using amount      = decltype(hana::tuple_c<int, 0, 0, 0, 0, 0, 0, 1>);

// composite dimensions
using velocity     = decltype(hana::tuple_c<int, 0, 1, -1, 0, 0, 0, 0>); // M/T
using acceleration = decltype(hana::tuple_c<int, 0, 1, -2, 0, 0, 0, 0>); // M/T^2
using force        = decltype(hana::tuple_c<int, 1, 1, -2, 0, 0, 0, 0>); // ML/T
```

CATCHING ERRORS

```
template <typename OtherDimensions>
explicit quantity(quantity<OtherDimensions> other)
  : value_(other.value_)
{
  static_assert(Dimensions{} == OtherDimensions{},
    "Constructing quantities with incompatible dimensions!");
}
```

COMPOSING DIMENSIONS

```
template <typename D1, typename D2>
auto operator*(quantity<D1> a, quantity<D2> b) {
  using D = decltype(hana::zip_with(std::plus<>{}, D1{}, D2{}));
  return quantity<D>{static_cast<double>(a) * static_cast<double>(b)};
}

template <typename D1, typename D2>
auto operator/(quantity<D1> a, quantity<D2> b) {
  using D = decltype(hana::zip_with(std::minus<>{}, D1{}, D2{}));
  return quantity<D>{static_cast<double>(a) / static_cast<double>(b)};
}
```

EXAMPLE: A SIMPLE EVENT SYSTEM

```
int main() {
  auto events = make_event_system(
    "foo"_e = function<void(string)>,
    "bar"_e = function<void(int)>,
    "baz"_e = function<void(double)>
  );

  events.on("foo"_e, [](string s) { cout << "foo with '" << s << "'!\n"; });
  events.on("foo"_e, [](string s) { cout << "foo with '" << s << "' again!\n"; });
  events.on("bar"_e, [](int i) { cout << "bar with '" << i << "'!\n"; });
  events.on("baz"_e, [](double d) { cout << "baz with '" << d << "'!\n"; });
  // events.on("unknown"_e, []() { }); // compiler error!

  events.trigger("foo"_e, "hello"); // no overhead for event lookup
  events.trigger("bar"_e, 4);
  events.trigger("baz"_e, 3.3);
  // events.trigger("unknown"_e); // compiler error!
}
```

ASSUME

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

PRIMER: COMPILE-TIME STRINGS

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

HOW THAT WORKS

```
template <char ...c> struct string { };

template <typename CharT, CharT ...c>
constexpr string<c...> operator"" _s() { return {}; }

template <char ...c1, char ...c2>
constexpr auto operator==(string<c1...>, string<c2...>) {
  return std::is_same<string<c1...>, string<c2...>>{};
}

template <char ...c1, char ...c2>
constexpr auto operator+(string<c1...>, string<c2...>) {
  return string<c1..., c2...>{};
}
```

OUR DSL

```
  auto events = make_event_system(
    "foo"_e = function<void(string)>,
    "bar"_e = function<void(int)>,
    "baz"_e = function<void(double)>
  );
```

```
template <typename Signature>
constexpr hana::basic_type<Signature> function{};

template <char ...c>
struct event {
  template <typename F>
  constexpr auto operator=(F f) const {
    return hana::make_pair(*this, f);
  }
};

template <typename CharT, CharT ...c>
constexpr event<c...> operator""_e() {
  return {};
}
```

STORING EVENTS

```
template <typename ...Events>
struct event_system;

template <typename ...Events, typename ...Signatures>
struct event_system<hana::pair<Events, hana::basic_type<Signatures>>...> {
  hana::map<
    hana::pair<Events, std::vector<std::function<Signatures>>>...
  > map_;
```

CONSTRUCTING THE SYSTEM

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

REGISTERING EVENTS

```
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);
}
```

TRIGGERING EVENTS

```
template <typename Event, typename ...Args>
void trigger(Event e, Args ...a) 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(a...);
}
```

EXAMPLE: RUST TRAITS

```
struct Circle {
  x: f64,
  y: f64,
  radius: f64,
}

trait HasArea {
  fn area(&self) -> f64;
}

impl HasArea for Circle {
  fn area(&self) -> f64 {
    std::f64::consts::PI * (self.radius * self.radius)
  }
}
```

USAGE

```
fn print_area<T: HasArea>(shape: T) {
  println!("This shape has an area of {}", shape.area());
}

fn main() {
  let c = Circle {
    x: 0.0f64,
    y: 0.0f64,
    radius: 1.0f64,
  };

  print_area(c);
}
```

NOTE THIS IS RELATED TO

- Go interfaces
- Haskell typeclasses
- C++0x concept maps
- Dynamic/Virtual concepts
- Type erasure

LET'S DO THAT IN C++

```
struct Circle {
  double x, y, radius;
};

struct HasArea : decltype(trait(
  "area"_s = function<double (self const&)>
)) { };

template <>
auto impl<HasArea, Circle> = make_impl(
  "area"_s = [](Circle const& self) -> double {
    return 3.1415 * (self.radius * self.radius);
  }
);
```

USAGE

```
void print_area(poly<HasArea> shape) {
  std::cout << "This shape has an area of " << (shape->*"area"_s)();
}

int main() {
  Circle c = {
    /*x:*/0.0,
    /*y:*/0.0,
    /*radius:*/1.0
  };

  print_area(c);
}
```

MOTIVATION: INHERITANCE SUCKS

- Intrusive
- Incompatible with value semantics
- Tightly coupled with dynamic storage
- Slow (hard to adapt to specific use case)

FULL LIBRARY: DYNO

(https://github.com/ldionne/dyno)

HOW DOES IT WORK?

IT'S EASY

JUST KIDDING

LET'S TAKE IT STEP BY STEP

```
struct HasArea : decltype(trait(
  "area"_s = function<double (self const&)>
)) { };
```

```
template <typename Signature>
constexpr hana::basic_type<Signature> function{};

struct self;

template <char ...c>
struct string {
  template <typename Signature>
  constexpr auto operator=(Signature sig) const {
    return hana::make_pair(*this, sig);
  }
};

template <typename CharT, CharT ...c>
constexpr string<c...> operator""_s() {
  return {};
}
```

IN OTHER WORDS

```
struct HasArea : decltype(trait(
  "area"_s = function<double (self const&)>
)) { };
```

is equivalent to

```
struct HasArea : decltype(trait(
  hana::pair<
    string<'a', 'r', 'e', 'a'>,
    hana::basic_type<double (self const&)>
  >{}
)) { };
```

WHAT ABOUT `trait`?

```
struct HasArea : decltype(trait(
  "area"_s = function<double (self const&)>
)) { };
```

```
template <typename ...Methods>
struct trait_t {
  hana::tuple<Methods...> methods;
};

template <typename ...Methods>
constexpr trait_t<Methods...> trait(Methods ...) {
  return {};
}
```

IN OTHER WORDS

```
struct HasArea : decltype(trait(
  "area"_s = function<double (self const&)>
)) { };
```

is equivalent to

```
struct HasArea : trait_t<
  hana::pair<
    string<'a', 'r', 'e', 'a'>,
    hana::basic_type<double (self const&)>
  >
> { };
```

WE HAVE A DESCRIPTION OF THE TRAIT

```
struct Circle {
  double x, y, radius;
};

struct HasArea : decltype(trait(
  "area"_s = function<double (self const&)>
)) { };

template <>
auto impl<HasArea, Circle> = make_impl(
  "area"_s = [](Circle const& self) -> double {
    return 3.1415 * (self.radius * self.radius);
  }
);
```

NOW WE NEED THE IMPLEMENTATION OF THE TRAIT

```
struct Circle {
  double x, y, radius;
};

struct HasArea : decltype(trait(
  "area"_s = function<double (self const&)>
)) { };

template <>
auto impl<HasArea, Circle> = make_impl(
  "area"_s = [](Circle const& self) -> double {
    return 3.1415 * (self.radius * self.radius);
  }
);
```

THIS IS EASY!

```
template <>
auto impl<HasArea, Circle> = make_impl(
  "area"_s = [](Circle const& self) -> double {
    return 3.1415 * (self.radius * self.radius);
  }
);
```

```
template <typename ...Name, typename ...Method>
auto make_impl(hana::pair<Name, Method> ...m) {
  return hana::make_map(m...);
}

template <typename Trait, typename T>
auto impl = make_impl();
```

REMEMBER

```
template <char ...c>
struct string {
  template <typename Signature>
  constexpr auto operator=(Signature sig) const {
    return hana::make_pair(*this, sig);
  }
};
```

IN OTHER WORDS

```
template <>
auto impl<HasArea, Circle> = make_impl(
  "area"_s = [](Circle const& self) -> double {
    return 3.1415 * (self.radius * self.radius);
  }
);
```

is equivalent to

```
template <>
auto impl<HasArea, Circle> = hana::make_map(
  hana::make_pair(
    "area"_s,
    [](Circle const& self) { ... }
  )
);
```

WE HAVE THE INTERFACE AND THE IMPLEMENTATION

```
struct Circle {
  double x, y, radius;
};

struct HasArea : decltype(trait(
  "area"_s = function<double (self const&)>
)) { };

template <>
auto impl<HasArea, Circle> = make_impl(
  "area"_s = [](Circle const& self) -> double {
    return 3.1415 * (self.radius * self.radius);
  }
);
```

WE MUST BIND THE TWO TOGETHER

```
void print_area(poly<HasArea> shape) {
  std::cout << "This shape has an area of " << (shape->*"area"_s)();
}

int main() {
  Circle c = {
    /*x:*/0.0,
    /*y:*/0.0,
    /*radius:*/1.0
  };

  print_area(c);
}
```

THIS IS WHERE THE COMPUTATIONAL PART OF HANA IS USEFUL

CAREFUL: IT GOES DOWNHILL FROM HERE

DIVING DEEPER INTO `poly`

```
template <typename Trait>
struct poly {
  template <typename T>
  poly(T t)
    : self_{new T{t}}
    , vtable_{impl<Trait, T>} // <= interesting stuff here
  { }

  template <typename F>
  auto operator->*(F f) const {
    return [=](auto ...args) {
      return vtable_[f](self_, args...);
    };
  }

private:
  void* self_; // simplification
  vtable<Trait> vtable_;
};
```

CREATING OUR OWN VTABLE

```
template <typename Trait>
class vtable {
  using Map = typename decltype(vtable_layout(Trait{}))::type;
  Map map_;

public:
  template <typename Impl>
  explicit vtable(Impl impl)
    : map_{erase_impl(Trait{}, impl)}
  { }

  template <typename F>
  auto operator[](F f) const {
    return map_[f];
  }
};
```

Step 1: Determine the vtable layout

```
template <typename Trait>
auto vtable_layout(Trait t) {
  auto erased = hana::transform(t.methods,
    hana::fuse([](auto name, auto sig) {
      using Signature = typename decltype(sig)::type;

      // 'double (T const&)' -> 'double (void const*)'
      using Erased = erase_signature_t<Signature>;

      return hana::type<hana::pair<decltype(name), Erased*>>{};
    }));

  // 'erased' is a 'tuple<type<pair<Name, Signature*>>...>'
  // we return a 'type<map<pair<Name, Signature*>...>>'
  return hana::unpack(erased, hana::template_<hana::map>);
}
```

Step 2: Erase incoming `impls`

```
template <typename Trait, typename Impl>
auto erase_impl(Trait t, Impl impl) {
  auto erased = hana::transform(t.methods,
    hana::fuse([&](auto name, auto sig) {
      using Signature = typename decltype(sig)::type;
      return hana::make_pair(
        name, erase_function<Signature>(impl[name])
      );
    }));

  // 'erased' is a 'tuple<pair<Name, Signature*>>'
  return hana::to_map(erased);
}
```

SUPPORTING REFINEMENTS

```
template <typename Reference>
struct Iterator : decltype(trait(
  CopyConstructible{},
  CopyAssignable{},
  Destructible{},
  Swappable{},
  "increment"_s = function<void (self&)>,
  "dereference"_s = function<Reference (self&)>
)) { };
```

Step 1: Take refined traits into account for vtable layout

```
template <typename Trait>
auto vtable_layout(Trait t) {
  auto erased = hana::transform(all_methods(t),
//                              ^^^^^^^^^^^^^^^
    hana::fuse([](auto name, auto sig) {
      ...
    }));

  ...
  return hana::unpack(erased, hana::template_<hana::map>);
}
```

Step 1.1: Implementing `all_methods`

```
template <typename ...Methods>
struct trait_t : trait_base { ... };
//               ^^^^^^^^^^

template <typename X>
constexpr auto is_trait(X const&) {
  return std::is_base_of<trait_base, X>{};
}

template <typename Trait>
auto all_methods(Trait t) {
  auto expanded = hana::transform(t.methods, [](auto m) {
    if constexpr (is_trait(m)) {
      return all_methods(m);
    } else {
      return hana::make_tuple(m);
    }
  });
  return hana::flatten(expanded);
}
```

Step 2: Take refined `impls` into account when filling the vtable

```
template <typename Trait>
struct poly {
  template <typename T>
  poly(T t)
    : self_{new T{t}}
    , vtable_{
        complete_impl<T>(Trait{}, impl<Trait, T>)
//      ^^^^^^^^^^^^^^^^^^^^^^^^^
      }
  { }
};
```

Step 2.1: Implementing `impl` completion

```
// For each refined trait, recursively complete the impl
// for that trait and merge that into the current impl.
template <typename T, typename Trait, typename Impl>
auto complete_impl(Trait t, Impl impl) {
  auto refined_traits = hana::filter(t.methods, is_trait);
  auto merged = hana::fold_left(refined_traits, impl,
    [](auto full_impl, auto refined) {
      using Refined = typename decltype(refined)::type;
      auto refined_impl = impl<Refined, T>;
      auto complete = complete_impl<T>(refined, refined_impl);
      return hana::union_(complete, full_impl);
    });

  return merged;
}
```

CONGRATULATIONS, YOU JUST DID A COMPILER'S JOB

AND THE RESULTS?

```
template <typename Signature>
struct Callable;

template <typename R, typename ...Args>
struct Callable<R(Args...)> : decltype(trait(
  "call"_s = function<R (self const&, Args...)>
)) { };

template <typename R, typename ...Args, typename F>
auto const impl<Callable<R(Args...)>, F> = make_impl(
  "call"_s = [](F const& f, Args ...args) -> R {
    return f(std::forward<Args>(args)...);
  }
);
```

std::function!

```
template <typename Signature>
struct std_function;

template <typename R, typename ...Args>
struct std_function<R(Args...)> {
  template <typename F = R(Args...)>
  std_function(F&& f) : poly_{std::forward<F>(f)} { }

  R operator()(Args ...args) const {
    return (poly_->*"call"_s)(std::forward<Args>(args)...);
  }

private:
  poly<Callable<R(Args...)>> poly_;
};
```

EXAMPLE: GENERATING JSON USING LIMITED REFLECTION

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

Output:

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

DEFINE YOUR TYPE LIKE THIS

```
struct Person {
  BOOST_HANA_DEFINE_STRUCT(Person,
    (std::string, name),
    (int, age)
  );
};
```

(non-intrusive version is `BOOST_HANA_ADAPT_STRUCT`)

HANDLE BASE TYPES

```
std::string quote(std::string s) { return "\"" + s + "\""; }

template <typename T>
auto to_json(T const& x) -> decltype(std::to_string(x)) {
  return std::to_string(x);
}

std::string to_json(char c) { return quote({c}); }
std::string to_json(std::string s) { return quote(s); }
```

HANDLE Sequences

```
template <typename Xs>
  std::enable_if_t<hana::is_a<hana::Sequence, Xs>(),
std::string> to_json(Xs const& xs) {
  auto json = hana::transform(xs, [](auto const& x) {
    return to_json(x);
  });

  return "[" + join(std::move(json), ", ") + "]";
}
```

HANDLE Structs

```
template <typename T>
  std::enable_if_t<hana::is_a<hana::Struct, T>(),
std::string> to_json(T const& x) {
  auto json = hana::transform(keys(x), [&](auto name) {
    auto const& member = hana::at_key(x, name);
    return quote(name.c_str()) + " : " + to_json(member);
  });

  return "{" + join(std::move(json), ", ") + "}";
}
```

THE FUTURE

HOW WOULD WE WANT METAPROGRAMMING TO LOOK LIKE?

CONSIDER SERIALIZATION TO JSON

```
struct point { float x, y, z; };
struct triangle { point a, b, c; };

struct tetrahedron {
  triangle base;
  point apex;
};

int main() {
  tetrahedron t{
    {{0.f,0.f,0.f}, {1.f,0.f,0.f}, {0.f,0.f,1.f}},
    {0.f,1.f,0.f}
  };

  to_json(std::cout, t);
}
```

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?

```
constexpr std::vector<std::meta::type>
sort_by_alignment(std::vector<std::meta::type> types) {
  std::sort(v.begin(), v.end(), [](std::meta::type t,
                                   std::meta::type u) {
    return t.alignment() < u.alignment();
  });
  return v;
}

constexpr std::vector<std::meta::type> types{
  reflexpr(Foo), reflexpr(Bar), reflexpr(Baz)
};

constexpr std::vector<std:meta::type> sorted = sort_by_alignment(types);

std::tuple<typename(sorted)...> tuple{...};
```

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

IT'S MUCH MORE PALATABLE THAN IT USED TO BE

AND IT'S ONLY GETTING BETTER

https://a9.com/careers