METAPROGRAMMING FOR DUMMIES

LOUIS DIONNE, C++NOW 2016

WHY DO WE NEED THIS TALK?

BECAUSE METAPROGRAMMING IS HARD?

OR BECAUSE IT'S TAUGHT THE WRONG WAY?

I THINK IT'S THE LATTER

I'LL DO MY BEST TO BREAK THE MOLD

BUT PLEASE FORGET WHAT YOU ALREADY KNOW

WHY YOU NEED METAPROGRAMMING

GENERATING JSON

```
struct Person {
    std::string name;
    int age;
};

int main() {
    Person joe{"Joe", 30};
    std::cout << to_json(joe);
}
```

IN PYTHON, WE COULD WRITE

```
def to_json(obj):
    members = []
    for member in obj.__dict__:
        members.append('"' + member + '" : ' +
                       str(getattr(obj, member)))
    return '{' + ', '.join(members) + '}'

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

print to_json(Person("John", 30))
```

IN C++, NOT SO

```
template <typename T>
std::string to_json(T const& obj) {
    std::vector<std::string> members;
    for (std::string const& member : obj.__members__()) // ???
        members.push_back('"' + member + "\" : " +
                                  to_json(obj.__get__(member))); // ???

    return "{" + boost::algorithm::join(members, ", ") + "}";
}
```

REGISTERING CLASSES

```
class A { };
class B { };
class C { };

template <typename T>
void register_() {
    std::cout << "Registering " << typeid(T).name()
              << ", which is of size " << sizeof(T) << std::endl;
    static_assert(sizeof(T) <= 1000, "");
}

int main() {
    register_<A>();
    register_<B>();
    register_<C>();
}
```

HOW TO ALLOW REGISTERING MANY AT ONCE?

CLASSIC SOLUTION

```
template <typename ...Ts>
struct register_;

template <>
struct register_<> {
    void operator()() const { }
};

template <typename T, typename ...Ts>
struct register_<T, Ts...> {
    void operator()() const {
        std::cout << "Registering " << typeid(T).name()
                  << ", which is of size " << sizeof(T) << std::endl;
        static_assert(sizeof(T) <= 1000, "");

        register_<Ts...>{}();
    }
};

int main() {
    register_<A, B, C>{}();
}
```

DRAWBACKS

- Not straightforward
- Not reusable

IN PYTHON, WE CAN WRITE

```
class A: pass
class B: pass
class C: pass

def register(classes):
    for c in classes:
        print "Registering " + c.__name__

register([A, B, C])
```

WHAT WE WOULD LIKE TO WRITE IN C++

```
void register_(std::vector<...> const& classes) {
    for (auto c : classes) {
        std::cout << "Registering " << c.name()
                  << ", which is of size " << sizeof(c) << std::endl;
        static_assert(sizeof(c) <= 1000, "");
    }
}

int main() {
    std::vector<...> classes{A, B, C};
    register_(classes);
}
```

OPTIMIZING ALIGNMENT

```
template <typename T, typename U, typename V>
struct Triple {
    T first;
    U second;
    V third;
};

struct A { char data; };
struct B { int data; };
struct C { double data; };

// What's the most packed layout?
```

WHAT WE WOULD LIKE TO WRITE

```
Type packed_triple(std::vector<Type> types) {
    std::sort(types.begin(), types.end(), [](auto const& a,
                                             auto const& b) {
        return a.alignment() > b.alignment();
    });

    return Type{Triple<types[0], types[1], types[2]>};
}

int main() {
    std::vector<Type> types{A, B, C};
    Packed = packed_triple(types); // get the type
    Packed triple{...}; // define an object of that type
}
```

INTRODUCING HETEROGENEITY

EVER TRIED RETURNING MULTIPLE VALUES FROM A FUNCTION?

```
double, double, double
from_spherical(double r, double theta, double phi) {
    double x = r * std::sin(theta) * std::cos(phi);
    double y = r * std::sin(phi) * std::sin(phi);
    double z = r * std::cos(theta);
    return (x, y, z);
}

double (x, y, z) = from_spherical(3, 16.5, 25.5);
```

USE std::tuple

```
std::tuple<double, double, double>
from_spherical(double r, double theta, double phi) {
    double x = r * std::sin(theta) * std::cos(phi);
    double y = r * std::sin(phi) * std::sin(phi);
    double z = r * std::cos(theta);
    return {x, y, z};
}

std::tuple<double, double, double> xyz = from_spherical(3, 16.5, 25.5);
double x = std::get<0>(xyz),
       y = std::get<1>(xyz),
       z = std::get<2>(xyz);
```

SOMEWHAT EQUIVALENT TO

```
struct {
    double _0; double _1; double _2;
} xyz = from_spherical(3, 16.5, 25.5);

double x = xyz._0,
       y = xyz._1,
       z = xyz._2;
```

IN ESSENCE IT'S LIKE A STRUCT

YET IT CAN BE SEEN LIKE A SEQUENCE!

A SEQUENCE THAT CAN HOLD DIFFERENT TYPES

```
struct Fish { };
struct Cat { };
struct Dog { };

std::tuple<Fish, Cat, Dog> my_animals() {
    // ...
    return {};
}

std::tuple<Fish, Cat, Dog> animals = my_animals();
Fish fish = std::get<0>(animals);
Cat  cat  = std::get<1>(animals);
Dog  dog  = std::get<2>(animals);
```

SOMEWHAT EQUIVALENT TO

```
struct {
    Fish _0; Cat _1; Dog _2; // notice the members have different types
} animals = my_animals();

Fish fish = animals._0;
Cat  cat  = animals._1;
Dog  dog  = animals._2;
```

LET'S GO BACK TO CLASS REGISTRATION

```
void register_(std::vector<...> const& classes) {
    for (auto c : classes) {
        std::cout << "Registering " << c.name()
                  << ", which is of size " << sizeof(c) << std::endl;
        static_assert(sizeof(c) <= 1000, "");
    }
}

int main() {
    std::vector<...> classes{A, B, C};
    register_(classes);
}
```

WOULD THAT WORK?

```
void register_(std::vector<std::type_info> const& classes) {
    for (auto c : classes) {
        std::cout << "Registering " << c.name()
                  << ", which is of size " << sizeof(c) << std::endl;
        static_assert(sizeof(c) <= 1000, "");
    }
}

int main() {
    std::vector<std::type_info> classes{typeid(A), typeid(B),
                                        typeid(C)};
    register_(classes);
}
```

SOLUTION: A STATIC `type_info`?

```
template <typename T>
struct static_type_info {
    using type = T;
};

void register_(std::vector<...> const& classes) {
    for (auto c : classes) {
        using T = typename decltype(c)::type;
        std::cout << "Registering " << typeid(T).name()
                  << ", which is of size " << sizeof(T) << std::endl;
        static_assert(sizeof(T) <= 1000, "");
    }
}

int main() {
    std::vector<...> classes{static_type_info<A>{},
                             static_type_info<B>{},
                             static_type_info<C>{}};
    register_(classes);
}
```

SOLUTION: `std::tuple`!

```
template <typename ...TypeInfos>
void register_(std::tuple<TypeInfos...> const& classes) {
    for (auto c : classes) {
        using T = typename decltype(c)::type;
        std::cout << "Registering " << typeid(T).name()
                  << ", which is of size " << sizeof(T) << std::endl;
        static_assert(sizeof(T) <= 1000, "");
    }
}

int main() {
    std::tuple<static_type_info<A>,
               static_type_info<B>,
               static_type_info<C>> classes;
    register_(classes);
}
```

NOW, WE ONLY NEED TO ITERATE ON A `std::tuple`

FORTUNATELY, THERE'S HANA

```
template <typename ...TypeInfos>
void register_(std::tuple<TypeInfos...> const& classes) {
    // like std::for_each!
    hana::for_each(classes, [](auto c) {
        using T = typename decltype(c)::type;
        std::cout << "Registering " << typeid(T).name()
                  << ", which is of size " << sizeof(T) << std::endl;
        static_assert(sizeof(T) <= 1000, "");
    });
}
```

THIS IS EQUIVALENT TO

```
template <typename ...TypeInfos>
void register_(std::tuple<TypeInfos...> const& classes) {
    auto f = [](auto c) {
        using T = typename decltype(c)::type;
        std::cout << "Registering " << typeid(C).name()
                  << ", which is of size " << sizeof(C) << std::endl;
        static_assert(sizeof(C) <= 1000, "");
    };

    f(std::get<0>(classes));
    f(std::get<1>(classes));
    f(std::get<2>(classes));
    // ...
}
```

HANA PROVIDES THIS `static_type_info`

```
int main() {
    std::tuple<hana::type<A>,
               hana::type<B>,
               hana::type<C>> classes;
    register_(classes);
}
```

HETEROGENEOUS DATA STRUCTURES AND ALGORITHMS

TUPLE

```
struct Cat      { std::string name; };
struct Dog      { std::string name; };
struct Elephant { std::string name; };
struct Fish     { std::string name; };
// ... comparison operators ...

hana::tuple<Cat, Dog, Fish> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}
};

auto animals2 = hana::make_tuple(
  Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}
);
```

INDEXING

```
hana::tuple<Cat, Dog, Fish> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}
};

assert(animals[0_c] == Cat{"Garfield"});
assert(animals[1_c] == Dog{"Beethoven"});
assert(animals[2_c] == Fish{"Nemo"});
```

INSERTING

```
hana::tuple<Cat, Dog, Fish> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}
};

hana::tuple<Cat, Elephant, Dog, Fish> more_animals =
                hana::insert(animals, 1_c, Elephant{"Dumbo"});
assert(more_animals[1_c] == Elephant{"Dumbo"}); // used to be Beethoven
```

REMOVING

```
hana::tuple<Cat, Dog, Fish> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}
};

hana::tuple<Cat, Fish> fewer_animals = hana::remove_at(animals, 1_c);
assert(fewer_animals[1_c] == Fish{"Nemo"}); // used to be Beethoven
```

ITERATING

```
hana::tuple<Cat, Dog, Fish> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}
};

// like std::for_each!
hana::for_each(animals, [](auto a) {
    std::cout << a.name << ' ';
});

// outputs `Garfield Beethoven Nemo`
```

TRANSFORMING

```
hana::tuple<Cat, Dog, Fish> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}
};

// like std::transform!
auto names = hana::transform(animals, [](auto a) {
    return a.name;
});

assert(names == hana::make_tuple(
    "Garfield", "Beethoven", "Nemo"
));
```

FILTERING OUT

```
hana::tuple<Cat, Fish, Dog, Fish> animals{
    Cat{"Garfield"}, Fish{"Jaws"}, Dog{"Beethoven"}, Fish{"Nemo"}
};

// like std::remove_if!
auto mammals = hana::remove_if(animals, [](auto a) {
    return hana::decltype_(a) == hana::type<Fish>{};
});

assert(mammals == hana::make_tuple(Cat{"Garfield"}, Dog{"Beethoven"}));
```

SORTING

```
hana::tuple<hana::type<char[10]>,
            hana::type<char[5]>,
            hana::type<char[1]>> types;

// like std::sort!
auto sorted = hana::sort(types, [](auto t, auto u) {
    return hana::sizeof_(t) < hana::sizeof_(u);
});

assert((sorted == hana::tuple<hana::type<char[1]>,
                              hana::type<char[5]>,
                              hana::type<char[10]>>{}));
```

PARTITIONING

```
hana::tuple<Cat, Dog, Fish, Dog> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}, Dog{"Lassie"}
};

// like std::partition!
auto parts = hana::partition(animals, [](auto a) {
    return hana::decltype_(a) == hana::type<Dog>{};
});

assert(hana::first(parts) ==
    hana::make_tuple(Dog{"Beethoven"}, Dog{"Lassie"})
);
assert(hana::second(parts) ==
    hana::make_tuple(Cat{"Garfield"}, Fish{"Nemo"})
);
```

SEARCHING

```
hana::tuple<Cat, Dog, Fish, Dog> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}, Dog{"Lassie"}
};

// similar to std::find_if
auto beethoven = hana::find_if(animals, [](auto a) {
    return hana::decltype_(a) == hana::type<Dog>{};
});

assert(*beethoven == Dog{"Beethoven"});

auto not_found = hana::find_if(animals, [](auto a) {
    return hana::decltype_(a) == hana::type<int>{};
});

assert(not_found == hana::nothing);
```

COUNTING

```
hana::tuple<Cat, Dog, Fish, Dog> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}, Dog{"Lassie"}
};

// like std::count_if!
auto dogs = hana::count_if(animals, [](auto a) {
    return hana::decltype_(a) == hana::type<Dog>{};
});

assert(dogs == 2u);
```

REVERSING

```
hana::tuple<Cat, Dog, Fish> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}
};

// like std::reverse!
auto reversed = hana::reverse(animals);

assert(reversed == hana::make_tuple(
    Fish{"Nemo"}, Dog{"Beethoven"}, Cat{"Garfield"}
));
```

UNPACKING

```
hana::tuple<Cat, Dog, Fish> animals{
    Cat{"Garfield"}, Dog{"Beethoven"}, Fish{"Nemo"}
};

auto names = hana::unpack(animals, [](auto ...a) {
    return std::vector<std::string>{a.name...};
});

assert((names == std::vector<std::string>{"Garfield",
                                          "Beethoven",
                                          "Nemo"}));
```

Equivalent to

```
auto f = [](auto ...a) {
    return std::vector<std::string>{a.name...};
};

auto names = f(animals[0_c], animals[1_c], animals[2_c]);
```

LET'S GO BACK TO ALIGNMENT OPTIMIZATION

```
template <typename T, typename U, typename V>
struct Triple {
    T first;
    U second;
    V third;
};

struct A { char data; };
struct B { int data; };
struct C { double data; };

// What's the most packed layout?
```

SOLUTION: USE A TUPLE!

```
template <typename Types>
auto packed_triple(Types const& types) {
    auto sorted = hana::sort(types, [](auto t, auto u) {
        return hana::alignof_(t) > hana::alignof_(u);
    });

    auto triple = hana::unpack(sorted, [](auto ...t) {
        using Answer = Triple<typename decltype(t)::type...>;
        return hana::type<Answer>{};
    });

    return triple;
}

int main() {
    hana::tuple<hana::type<A>, hana::type<B>, hana::type<C>> types;
    using Packed = decltype(packed_triple(types))::type;
    Packed triple{/* ... */};
}
```

A MORE TERSE VARIANT

```
template <typename Types>
auto packed_triple(Types const& types) {
    auto sorted = hana::sort(types, [](auto t, auto u) {
        return hana::alignof_(t) > hana::alignof_(u);
    });

    return hana::unpack(sorted, hana::template_<Triple>);
}

int main() {
    auto types = hana::tuple_t<A, B, C>;
    using Packed = decltype(packed_triple(types))::type;
    Packed triple{/* ... */};
}
```

MAP

```
struct Cat      { std::string name; };
struct Dog      { std::string name; };
struct Elephant { std::string name; };
struct Fish     { std::string name; };
// ... comparison operators ...

auto animals = hana::make_map(
    hana::make_pair("Nemo"_s, Fish{"Nemo"}),
    hana::make_pair("Beethoven"_s, Dog{"Beethoven"}),
    hana::make_pair("Garfield"_s, Cat{"Garfield"})
);
```

ACCESSING KEYS

```
assert(animals["Nemo"_s] == Fish{"Nemo"});
assert(animals["Beethoven"_s] == Dog{"Beethoven"});
assert(animals["Garfield"_s] == Cat{"Garfield"});
```

QUERYING

```
static_assert(hana::contains(animals, "Beethoven"_s), "");
static_assert(!hana::contains(animals, "Lassie"_s), "");
```

INSERTING

```
auto more_animals = hana::insert(animals,
    hana::make_pair("Dumbo"_s, Elephant{"Dumbo"}));

assert(more_animals["Dumbo"_s] == Elephant{"Dumbo"});
```

REMOVING

```
auto fewer_animals = hana::erase_key(animals, "Beethoven"_s);
static_assert(!hana::contains(fewer_animals, "Beethoven"_s), "");
```

AN EXAMPLE: AN EVENT SYSTEM

```
auto events = make_event_system("foo"_s, "bar"_s, "baz"_s);

events.on("foo"_s, []() { std::cout << "foo triggered!" << '\n'; });
events.on("foo"_s, []() { std::cout << "foo again!" << '\n'; });
events.on("bar"_s, []() { std::cout << "bar triggered!" << '\n'; });
events.on("baz"_s, []() { std::cout << "baz triggered!" << '\n'; });
// events.on("unknown"_s, []() { }); // compiler error!

events.trigger("foo"_s, "bar"_s);
events.trigger("baz"_s);
// events.trigger("unknown"_s); // compiler error!
```

```
template <typename ...Events>
auto make_event_system(Events ...events) {
    using Callbacks = std::vector<std::function<void()>>;
    using Map = decltype(hana::make_map(
        hana::make_pair(events, Callbacks{})...
    ));

    return event_system<Map>{};
}
```

```
template <typename EventMap>
struct event_system {
    EventMap map_;
```

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

    map_[e].push_back(handler);
}
```

```
    template <typename ...Events>
    void trigger(Events ...events) {
        hana::for_each(hana::make_tuple(events...), [this](auto e) {
            auto is_known_event = hana::contains(map_, e);
            static_assert(is_known_event,
                "trying to trigger an unknown event");

            for (auto& handler : this->map_[e])
                handler();
        });
    }
};
```

APPLICATIONS ARE ENDLESS

WE SAW

- JSON
- Class registration
- Alignment optimization
- Event systems

BUT THERE'S ALSO

- Dimensional analysis
- Serialization
- Automatic paralellization
- Domain specific languages
- Expression templates

HANA IS IN BOOST

USE IT!

HELP US IMPROVE IT

THANK YOU

http://ldionne.com
http://github.com/ldionne
http://github.com/boostorg/hana