facebook

Meta Techniques
Heterogeneous Polymorphism and Fast Prototyping

Marcelo Juchem <marcelo@fb.com>

CppCon 2014

September 10, 2014

Agenda

Motivation & expectations

What is this?

```
using supported_data_types = lib::type_list<
  data_type<
    str::list, std::vector<std::string>,
    constructor<>,
    operation<str::at, method::at::member_function, std::string(std::size_t)>,
    operation<str::insert, method::emplace_back::member_function, void(std::string)>,
    operation<str::size, method::size::member_function, std::size_t()>
  >,
  data_type<
    str::string, std::string,
    constructor<std::string>,
    operation<str::get, method::c_str::member_function, std::string()>,
    operation<str::substr, method::substr::member_function, std::string(std::size_t, std::size_t)>,
    operation<str::append, method::append::member_function, void(std::string)>,
    operation<str::size, method::size::member_function, std::size_t()>
  >
>;
```

What is this?

```
using supported_data_types = lib::type_list<
  data_type<
    str::list, std::vector<std::string>,
    constructor<>,
    operation<str::at, method::at::member_function, std::string(std::size_t)>,
    operation<str::insert, method::emplace_back::member_function, void(std::string)>,
    operation<str::size, method::size::member_function, std::size_t()>
  >,
  data_type<
    str::string, std::string,
    constructor<std::string>,
    operation<str::get, method::c_str::member_function, std::string()>,
    operation<str::substr, method::substr::member_function, std::string(std::size_t, std::size_t)>,
    operation<str::append, method::append::member_function, void(std::string)>,
    operation<str::size, method::size::member_function, std::size_t()>
  >
>;
```

What just happened?

```
using supported_data_types = lib::type_list<
  data_type<
    str::list, std::vector<std::string>,
    constructor<>,
    operation<str::at, method::at::member_function, std::string(std::size_t)>,
    operation<str::insert, method::emplace_back::member_function, void(std::string)>,
    operation<str::size, method::size::member_function, std::size_t()>
  >,
  data_type<
    str::string, std::string,
    constructor<std::string>,
    operation<str::get, method::c_str::member_function, std::string()>,
    operation<str::substr, method::substr::member_function, std::string(std::size_t, std::size_t)>,
    operation<str::append, method::append::member_function, void(std::string)>,
    operation<str::size, method::size::member_function, std::size_t()>
  >,
  data_type<
    str::map, std::unordered_map<std::string, std::string>,
    constructor<>,
    operation<str::get, method::operator_square_bracket::member_function, std::string(std::string)>,
    operation<str::insert, method::emplace::member_function, void(std::string, std::string)>,
    operation<str::size, method::size::member_function, std::size_t()>
  >
>;
```

"Imagine the following scenario. You come from a design meeting with a couple of printed diagrams, scribbled with your annotations. Okay, the event type passed between these objects is not char anymore; it's int . You change one line of code. The smart pointers to Widget are too slow; they should go unchecked. You change one line of code. The object factory needs to support the new Gadget class just added by another department. ***You change one line of code.***

You have changed the design. *Compile. Link. Done."*

Andrei Alexandrescu, Modern C++ Design

Live demo

Motivation

Some problems...

- Translating ideas to code is hard
  - Hard to reason when information is scattered
  - Logic replication when many moving parts
- High level engines often inefficient
  - Runtime cost
  - Flexibility at the expense of simplicity
  - “String maps processors”

Motivation

What if we could…

- Describe the software as compile-time metadata
- Manipulate compile-time metadata efficiently
  - With well known data structures
- Automatically translate metadata to code
- Go from runtime to TMP

- … and give this power to the non-initiated

Expectations

What NOT to expect from this talk

- A Template Meta-Programming (TMP) tutorial
- Panacea
  - Not everything is a nail
- Extensive code listings
  - Assume there is a library which works as advertised
  - More on that later

Expectations

When you do have a nail…

- Techniques for fast prototyping of code
  - Less boilerplate
  - DRY
  - Loose coupling
  - Catch bugs at compile-time
  - Increased performance
  - Reduced footprint

Goal

- Working prototype for a database that:
  - Stores arbitrary data structures rather than only tables
  - Supports different operations depending on the data type used
  - Uses **less than** 400 lines of code
  - Allows one to trivially add or remove data types and operations

Enriching our toolkit

Variables

Runtime

```
SomeType x = SomeValue;

auto y = SomeValue;
```

TMP

?

Variables

Runtime

```
SomeType x = SomeValue;

auto y = SomeValue;
```

TMP

```
using x = SomeType;

typedef SomeType y;

struct z { /* implementation */ };

template <typename T> struct SomeTemplate { ... }; // not a type but a type template

using w = SomeTemplate<SomeOtherType>; // now it is a type, after template instantiation
```

Scalars

Runtime

```
int x = 10;

double y = 5.6;
```

TMP

?

Scalars

Runtime

```
int x = 10;

double y = 5.6;
```

TMP

```
using x = std::integral_constant<int, 10>;

using y = std::ratio<56, 10>;
```

Lists

Runtime

```
std::vector<int> z{1, 2, 3};

int x[] = {1, 2, 3};

std::array<int, 3> y{{1, 2, 3}};

std::list<int> w{1, 2, 3};
```

TMP

?

Lists

Runtime

```
std::vector<int> z{1, 2, 3};
```

TMP

```
using x = lib::type_list<T1, T2, T3, T4>;

using y = lib::type_list<int, double, void>;

using z = lib::type_list<
  std::integral_constant<int, 1>,
  std::integral_constant<int, 2>,
  std::integral_constant<int, 3>
>;
```

List (some operations)

```
using x = lib::type_list<T1, T2, T3, T4>;

using y = x::slice<1, 3>; // lib::type_list<T2, T3>

using z = x::size; // 4

using w = x::at<x::size / 2>; // T2

// calls the visitor for each of the types in the list
x::foreach([](auto tag) {
  // tag is of type lib::type_tag<T, Index>
  std::cout << "visited: " << typeid(decltype(tag)::type).name()
    << " at " << decltype(tag)::value;
});

// prints "visited: T4 at 3"
bool found = x::visit(3, [](auto tag) {
  // tag is of type lib::type_tag<T, Index>
  std::cout << "visited: " << typeid(decltype(tag)::type).name()
    << " at " << decltype(tag)::value;
});

using v = x::apply<std::tuple>; // std::tuple<T1, T2, T3, T4>
```

List (some operations)

```
template <int X> using val = std::integral_constant<int, X>;
using g = lib::type_list<val<4>, val<2>, val<3>, val<7>, val<1>, val<6>, val<5>>;

template <typename T> using square = val<T::value * T::value>;

using h = g::transform<square>; // lib::type_list<val<1>, val<4>, val<9>, val<16>,
val<25>, val<36>, val<49>>;
```

List (some operations)

```
template <int X> using val = std::integral_constant<int, X>;
using g = lib::type_list<val<4>, val<2>, val<3>, val<7>, val<1>, val<6>, val<5>>;

template <typename T> using square = val<T::value * T::value>;

using h = g::transform<square>; // lib::type_list<val<1>, val<4>, val<9>, val<16>,
val<25>, val<36>, val<49>>;

using m = g::sort<>; // lib::type_list<val<1>, val<2>, val<3>, val<4>, val<5>, val<6>,
val<7>>;
```

List (some operations)

```
template <int X> using val = std::integral_constant<int, X>;
using g = lib::type_list<val<4>, val<2>, val<3>, val<7>, val<1>, val<6>, val<5>>;

template <typename T> using square = val<T::value * T::value>;

using h = g::transform<square>; // lib::type_list<val<1>, val<4>, val<9>, val<16>,
val<25>, val<36>, val<49>>;

using m = g::sort<>; // lib::type_list<val<1>, val<2>, val<3>, val<4>, val<5>, val<6>,
val<7>>;

bool found = m::binary_search<>::exact(
  5,
  [](auto tag, auto needle) {
    // tag is of type lib::indexed_type_tag<T, Index>
    // ...
  }
);
```

List (some operations)

```
using h = lib::type_list<val<1>, val<2>, val<3>, val<4>, val<5>, val<6>, val<7>>;

bool found = m::binary_search<>::exact(
  5, [](auto tag, auto needle) {
    // tag is of type lib::indexed_type_tag<T, Index>
    // ...
  }
);
```

Generates code similar to:

```
bool exact(TNeedle &&needle, TVisitor &&visitor) {
    if (needle < 4) {
        if (needle < 2) {
            if (needle == 1) { visitor(1...); return true; } else { return false; }
        } else if (needle > 2) {
            if (needle == 3) { visitor(3...); return true; } else { return false; }
        } else { visitor(2...); return true; }
    } else if (needle > 4) {
        if (needle < 6) {
            if (needle == 5) { visitor(5...); return true; } else { return false; }
        } else if (needle > 6) {
            if (needle == 7) { visitor(7...); return true; } else { return false; }
        } else { visitor(6...); return true; }
    } else { visitor(4...); return true; }
};
```

Lists of scalars

Runtime

```
std::vector<int> z{1, 2, 3, 4};
```

TMP

```
using x = lib::constant_sequence<int, 1, 2, 3, 4>;

using y = lib::constant_range<int, 1, 5>; // lib::constant_sequence<int, 1, 2, 3, 4>;

using z = lib::constant_sequence<short, -4, 0, 3, 99, -21>;

using w = lib::constant_sequence<char, 'h', 'e', 'l', 'l', 'o'>;
```

Lists of scalars (some operations)

Besides most list operations

```
using x = lib::constant_sequence<char, 'h', 'e', 'l', 'l', 'o'>;

// gets an automatically allocated std::array
auto y = x::array(); // std::array<char, 5>{{ 'h', 'e', 'l', 'l', 'o' }}

// gets an automatically allocated std::array with a null terminator
auto z = x::z_array(); // std::array<char, 6>{{ 'h', 'e', 'l', 'l', 'o', '\0' }}
```

Strings

Runtime

```
auto s1 = "hello";

std::string s2("world");

auto s3 = L"wide string";
```

TMP

?

Strings

Runtime

```
auto s1 = "hello";

std::string s2("world");

auto s3 = L"wide string";
```

TMP

```
using x = lib:: type_string<char, 'h', 'e', 'l', 'l', 'o'>;

TYPE_STR(y, "hello");

using z = lib::type_string<wchar_t, L'w', L'o', L'r', L'l', L'd'>;

TYPE_STR(w, L"world");

using v = lib:: type_string<int, 4, 9, 1, 5, 6>;
```

type_string inherits from constant_sequence

String (operations)

Besides all sequence operations

```
TYPE_STR(x, "hello");

auto y = x::array(); // just like in constant_sequence

auto z = x::z_array(); // just like in constant_sequence

// gets a standard, dynamically allocated string
auto w = x::string(); // std::string("hello")
```

Maps

Runtime

```
std::unordered_map<std::string, std::string> y{
  { "hello", "world" }, { "foo", "bar" }
};

std::unordered_map<int, int> x{
  { 1, 2 }, { 3, 4 }, { 5, 6 }
};
```

TMP

?

Maps

Runtime

```cpp
std::unordered_map<std::string, std::string> x{
  { "hello", "world" }, { "foo", "bar" }
};
```

TMP

```cpp
using y = lib::type_map<
  lib::type_pair<K1, V1>, // K1 -> V1
  lib::type_pair<K2, V2> // K2 -> V2
>;

using z = lib::build_type_map<
  K1, V1, // K1 -> V1
  K2, V2 // K2 -> V2
>;

TYPE_STR(hello, "hello");
TYPE_STR(world, "world");

using w = lib::build_type_map<
  hello, world // -> "hello" -> "world"
>;
```

Map (operations)

```
using x = lib::build_type_map<K1, V1, K2, V2>;

using y = x::find<K1>; // V1

using z = x::find<K3>; // lib::type_not_found_tag

using w = x::find<K3, Tx>; // Tx

using v = x::keys; // lib::type_list<K1, K2>

using u = x::mapped; // lib::type_list<V1, V2>

bool found = x::visit<K1>([](auto tag) {
  // tag is of type lib::type_pair<TKey, TValue>
  std::cout << "visited " << typeid(decltype(tag)::first).name()
    << " -> " << typeid(decltype(tag)::second).name();
});
```

Map (operations)

```
template <int X> using val = std::integral_constant<int, X>;
using v = lib::build_type_map<val<5>, T5, val<9>, T9, val<1>, T1>;

using h = g::sort<>; // lib::build_type_map<val<1>, T1, val<5>, T5, val<9>, T9>

bool found = x::binary_search<>::exact(
  5,
  [](auto tag) {
    // tag is of type lib::indexed_type_tag<type_pair<TKey, TValue>, Index>
    using pair = decltype(tag)::type;
    std::cout << "visited at “ << decltype(tag)::value << ": "
      << typeid(pair::first).name() << " -> " << typeid(pair::second).name();
  }
);
```

Prefix tree

```
TYPE_STRING(abc, "abc");
TYPE_STRING(abd, "abd");
TYPE_STRING(abcd, "abcd");

// builds a prefix tree for efficient string lookup in runtime
using x = lib::type_prefix_tree_builder<>::build<
  abc, abd, abcd
>;

std::string needle("abcd");

bool found = x::match<>::exact(
  needle.begin(), needle.end(),
  [](auto tag) {
    // tag is of type lib::type_tag<TypeString>
    using str = decltype(tag)::type;
    // ...
  }
);
```

Variants

```
lib::variant<int, double> x; // empty variant that supports int and double

using types = lib::type_list<int, double>;
using var = types::apply<lib::variant>; // lib::variant<int, double>

var v; // empty variant

v = 5; // v now contains an int
auto i = v.get<int>(); // returns 5
auto d = v.get<double>(); // throws, it doesn't contain a double
auto s = v.get<std::string>(); // compilation error: this variant doesn't support std::string

v = 4.2; // v now contains a double
auto k = v.try_get<int>(); // returns nullptr

var.visit([](auto value) { cout << value << endl; }); // prints 4.2

var.is_of<double>(); // returns true
```

Crossing boundaries
TMP <-> Runtime

- It’s easy to use template metadata at runtime:

```
template <typename T> void print() { cout << T::value << endl; }
```

- It’s easy to choose templates based on compile-time parameters

```
using types = lib::type_list<int, double>;

template <std::size_t Index> using type = typename types::template
at<Index>;

template <std::size_t Index> auto read() {  type<Index> out; cin >>
out; return out; }
```

- Is it possible to choose templates based on **runtime** data?

Crossing boundaries

TMP <-> Runtime

- It’s easy to use template metadata at runtime:

```
template <typename T> void print() { cout << T::value << endl; }
```

- It’s easy to choose templates based on compile-time parameters

```
using types = lib::type_list<int, double>;

template <std::size_t Index> using type = typename types::template
at<Index>;

template <std::size_t Index> auto read() {  type<Index> out; cin >>
out; return out; }
```

- Is it possible to choose templates based on **runtime** data?
  - Yes, with visitors

Crossing boundaries

Getting from a runtime integer index to a type

```
using list = lib::type_list<T0, T1, T2, T3, T4>;

int index;
cin >> index;

bool found = list::visit(index, [](auto tag) {
  // now we have the type and index available to the type system
  using T = decltype(tag)::type;
  constexpr std::size_t Index = decltype(tag)::value;
});
```

Crossing boundaries

Getting from a runtime value to a type

```
template <int... V> int_list = lib::type_list<std::integral_constant<int, V>...>;

using list = int_list<0, 1, 2, 3, 4>;

int value;
cin >> value;

bool found = list::binary_search<>::exact(value, [](auto tag, auto needle) {
  // now we have the type and its index available to the type system
  using T = decltype(tag)::type;
  constexpr std::size_t Index = decltype(tag)::value;
});
```

Crossing boundaries

Getting from a string to a type

```
TYPE_STRING(abc, "abc");
TYPE_STRING(abd, "abd");
TYPE_STRING(abcd, "abcd");

using trie = lib::type_prefix_tree_builder<>::build<abc, abd, abcd>;

std::string needle;
cin >> needle;

bool found = x::match<>::exact(
  needle.begin(), needle.end(),
  [](auto tag) {
    // now we have the needle available to the type system
    using TNeedle = decltype(tag)::type;
  }
);
```

Practical example 1

Example 1

Problem statement

- Write an engine capable of running arbitrary stateless operations
  - Read arguments from stdin
- Operations are described with metadata
  - Name, Functor, Argument types, Result type
- Engine API should look like this:
  - `run_operation<operation_metadata>();`

Example 1

Metadata

```
template <typename...> struct op_metadata;

template <typename N, typename M, typename R, typename... Args>
struct op_metadata<N, M, R(Args...)> {
  using name = N;
  using method = M;
  using result = R;
  using args = lib::type_list<Args...>;
};

struct join_functor {
  std::string operator ()(std::string const &lhs, std::string const &rhs) { return lhs + rhs; }
};

namespace str { TYPE_STR("join") join; }
using join_op = op_metadata<str::join, join_functor, std::string(std::string, std::string)>;
```

Example 1

Engine Sketch

```
template <typename Op> void run_operation() {
    // print operation name
    // derive tuple from argument types
    // read arguments into tuple
    // expand tuple
    // call method
}
```

Example 1

Print operation name

```
template <typename Op> void run_operation() {
    cout << "running operation " << Op::name::string();
    // derive tuple from argument types
    // read arguments into tuple
    // expand tuple
    // call method
}
```

Example 1

Derive tuple

```
template <typename Op> void run_operation() {
    cout << "running operation " << Op::name::string();
    typename Op::args::template apply<std::tuple> args; // std::tuple<std::string, std::string>
    // read arguments into tuple
    // expand tuple
    // call method
}
```

Example 1

Read arguments

```
template <typename Op> void run_operation() {
    cout << "running operation " << Op::name::string();
    typename Op::args::template apply<std::tuple> args; // std::tuple<std::string, std::string>
    Op::args::foreach([&](auto arg) { // lib::indexed_type_tag<Arg>
        cin >> std::get<decltype(arg)::value>(args);
    });
    // expand tuple
    // call method
}
```

Example 1

Expand tuple

```
template <typename Op> void run_operation() {
    cout << "running operation " << Op::name::string();
    typename Op::args::template apply<std::tuple> args; // std::tuple<std::string, std::string>
    Op::args::foreach([&](auto arg) { // lib::indexed_type_tag<Arg>
        cin >> std::get<decltype(arg)::value>(args);
    });
    using indexes = lib::constant_range<std::size_t, 0, Op::args::size>;
    call_method<Op::method, Op::result>(indexes(), args);
}

 // call method
```

Example 1

Call method with expanded tuple

```
template <typename Op> void run_operation() {
    cout << "running operation " << Op::name::string();
    typename Op::args::template apply<std::tuple> args; // std::tuple<std::string, std::string>
    Op::args::foreach([&](auto arg) { // lib::indexed_type_tag<Arg>
        cin >> std::get<decltype(arg)::value>(args);
    });
    using indexes = lib::constant_range<std::size_t, 0, Op::args::size>;
    call_method<Op::method, Op::result>(indexes(), args);
}

template <typename M, typename R, typename Args, size_t... Indexes>
R call_method(lib::constant_sequence<std::size_t, Indexes...> indexes, Args &&tuple) {
    M method;
    // expands to, for instance, method(std::get<0>(tuple), std::get<1>(tuple));
    return method(std::get<Indexes>(tuple)...);
}
```

Practical example 2

Example 2

Problem statement

- Extend engine to dynamically choose the operation to run
- Read operation name from stdin
- Operation lookup must be efficient at runtime
- Provide a REPL user interface

Example 2

Metadata manipulation

```
namespace str { TYPE_STR(substr, "substr"); }

struct substr_functor {
  std::string operator ()(std::string const &s, std::size_t i, std::size_t count) { return s.substr(i,
count); }
};

using known_ops = lib::type_list<
  op_metadata<str::join, join_functor, std::string(std::string, std::string)>,
  op_metadata<str::substr, substr_functor, std::string(std::string, std::size_t, std::size_t)>
>;

template <typename T> using get_name = typename T::name;

// a map form name to metadata
using op_map = lib::type_map_from<get_name>::list<known_ops>;

// a prefix tree of operation names
using op_trie = op_map::keys::apply<type_prefix_tree_builder<>::build>;
```

Example 2

REPL

```
int main() {
    for (std::string op; cin >> op; ) {
        bool found = op_trie::match<>::exact(
            op.begin(), op.end(),
            [](auto op_name) {
                using operation = op_map::find<decltype(op_name)::type>;
                run_operation<operation>();
            }
        );

        if (!found) { cout << “operation not found” << endl; }
    }
}
```

Heterogeneous polymorphism

Polymorphism

Virtual inheritance

- Introducing new types is non-intrusive
- Useful when LSP applies
  - Consistent interface and behavior
- Hacky when it doesn't
  - Classes limited to parent's interface
  - Unsupported methods throw? Return special values?

Polymorphism

Virtual inheritance

- Can't call function templates via base class
- Derived types erased
- Performance and footprint
  - V-Table
  - Dynamic allocation
- How to deal with metadata?
  - Runtime maps of factories?

Polymorphism

Variants

- Smart automatic or dynamic storage allocation
- Can call function templates
- Stored types can be unrelated
- TMP friendly
  - But types must be known beforehand
- Visitors dictate interfaces
  - Heterogeneous interfaces friendly

Prototyping our database

A sketch of our database

```
using supported = /* map from data type names to data types */;
using by_name = /* trie of data type names */;

// a variant that can hold a single data type at a time
using var = supported::mapped::transform<get_type_member>::apply<lib::variant>;

class Engine {
  // maps an instance name to an instance
  std::unordered_map<std::string, var> instances_;

  // ...

public:

  // ...

  void create_instance(std::string const &instance_name, std::string const &dt_name) {
    // lookup by_name with dt_name -> TMetadata
    // lookup instances_ with instance_name -> instance
    // derive args tuple from constructor metadata
    // read args into tuple
    instance.emplace<typename TMetadata::type>(/* expand tuple */);
  }
};
```

Call traits

Rough interface description

```
#define CALL_TRAITS(class_name, function_name) // ...

struct class_name {
  template <typename T, typename... Args>
  auto operator ()(T &&subject, Args &&...args) {
    return subject.function_name(std::forward<Args>(args)...);
  }

  template <typename T, typename... Args>
  using supported = ...; // std::true_type or std::false_type
};
```

Call traits

Flexible function binders

- Bind to function name
  - Not to signature
  - Stateless
- Provide reflection information
  - E.g.: is given signature supported?

Call traits

Usage

```
CALL_TRAITS(call_foo, foo);

struct Bar {
    void foo(int x) { cout << x; }
};


cout << call_foo::supported<Bar>::value; // false
cout << call_foo::supported<Bar, int>::value; // true


Bar instance;
call_foo traits;

traits(instance); // compilation error: can’t call x.foo()
traits(instance, 5); // calls x.foo(5)
```

Full circle

Our metadata

```
using supported_data_types = lib::type_list<
  data_type<
    str::list, std::vector<std::string>,
    constructor<>,
    operation<str::at, method::at::member_function, std::string(std::size_t)>,
    operation<str::insert, method::emplace_back::member_function, void(std::string)>,
    operation<str::size, method::size::member_function, std::size_t()>
  >,
  data_type<
    str::string, std::string,
    constructor<std::string>,
    operation<str::get, method::c_str::member_function, std::string()>,
    operation<str::substr, method::substr::member_function, std::string(std::size_t, std::size_t)>,
    operation<str::append, method::append::member_function, void(std::string)>,
    operation<str::size, method::size::member_function, std::size_t()>
  >,
  data_type<
    str::map, std::unordered_map<std::string, std::string>,
    constructor<>,
    operation<str::get, method::operator_square_bracket::member_function, std::string(std::string)>,
    operation<str::insert, method::emplace::member_function, void(std::string, std::string)>,
    operation<str::size, method::size::member_function, std::size_t()>
  >
>;
```

Potential uses

Portable User Interface Design

```
TYPE_STR(username, "username");
TYPE_STR(password, "password");

struct password_form {};

struct password_validator {
  void operator ()(std::string const &username, std::string const &pasword);
};

using forms = lib::build_type_map<
  password_form, form<
    text_field<username>,
    text_field<password>,
    submit_button<password_validator, username, password>
  >
>;

auto result = display_input_form<forms::find<password_form>>(ui_engine);

template <typename TForm> auto display_input_form(gtk_engine &engine) { ... }
template <typename TForm> auto display_input_form(ncurses_engine &engine) { ... }
template <typename TForm> auto display_input_form(html_renderer_engine &engine) { ... }
```

Portable Database Models

```
TYPE_STR(person, "person"); TYPE_STR(name, "name");
TYPE_STR(age, "age");TYPE_STR(description, "description");
TYPE_STR(role, "role"); TYPE_STR(eid, "employee_number");

using entities = lib::type_list<
  entity<person, property<name, std::string>, property<age, unsigned>, pk<property<eid, unsigned>>>,
  entity<role, property<name, str::string>, property<description, std::string>, pk<property<eid, unsigned>>>
>;

struct underage_employees_query {};

using queries = lib::build_type_map<
 underage_employees_query, query<
    join<person, role, eid>,
    where<less_than<member<person, age>, std::integral_constant<unsigned, 21>>,
    output<person, name, age>,
    output<role, alias<name, role>>
  >
>;

auto result_set = execute_query<entities, queries::find<underage_employees_query>>(db_connection);

template <typename TEntity, typename TQuery> mysql_result_set execute_query(mysql_connection &db);
template <typename TEntity, typename TQuery> sqlite_result_set execute_query(sqlite_connection &db);
```

What else?

- Auto-generate stubs for language interop
- Auto-generate IDLs from code
- Derive serialization code from metadata
- Derive concurrency model from metadata

Questions ?

Performance benchmark

type_prefix_tree benchmark

Relative performance (10 characters strings)

algorithm \ n	1	5	10	20	30
type_prefix_tree	100.00%	100.00%	100.00%	100.00%	100.00%
Sorted array	65.74%	42.91%	34.94%	29.98%	30.07%
Sorted vector	61.96%	51.33%	33.82%	32.18%	33.42%
Unordered set	34.89%	36.76%	34.78%	34.64%	36.15%
Set	61.36%	42.21%	34.92%	31.45%	29.46%
Sequential ifs	115.55%	60.94%	32.25%	16.58%	11.57%

type_prefix_tree benchmark

Time per iteration (10 characters strings)

algorithm \ n	1	5	10	20	30
type_prefix_tree	36.15ns	97.75ns	168.55ns	323.97ns	517.88ns
Sorted array	54.98ns	227.77ns	482.33ns	1.08us	1.72us
Sorted vector	58.34ns	190.43ns	498.37ns	1.01us	1.55us
Unordered set	103.62ns	265.91ns	484.59ns	935.26ns	1.43us
Set	58.91ns	231.55ns	482.73ns	1.03us	1.76us
Sequential ifs	31.28ns	160.40ns	522.57ns	1.95us	4.48us

Where to get it

When can I use it?

Does this library exist?

When can I use it?

NOW

- This library is being released to the public as we speak

Facebook Template Library

github.com/facebook/fatal

By the way...

If you're excited about

- Doing templates like this for a living

  or

- Just working with templates for a living

  or

- Using the latest C++ standard for a living

By the way...

We're hiring

facebook.com/careers

facebook