REFLECTION TECHNIQUES IN C++

PAUL FULTZ II

WHAT IS REFLECTION?

- Introspection on types
  - Enumerate over class members.
  - Not predicate-based introspection(such as checking if a class has a member function).
- Express our data as C++ classes
- In C++, we can store reflection data at compile-time, instead of at run-time.
- No native language feature for reflection in C++
  - There is study group working on a proposal for adding reflection to C++

USES

- Message serialization to JSON and XML
  - Treat field names as object keys or tag names

- Object relationship mapping to a database
  - Requires annotating fields with attributes

- General data-driven development

TECHNIQUES

- Boost.Fusion
- The Visitor Pattern
- Do-It-Yourself

BOOST.FUSION

- Provides generic algorithms for tuples

SIMPLE EXAMPLE

```
auto stuff = std::make_tuple(1, 'x', "howdy");
int i = at_c<0>(stuff);
char ch = at_c<1>(stuff);
std::string s = at_c<2>(stuff);
```

PRINT XML

```
template<class T>
void print_xml(const T& t)
{
    for_each(t, [](const auto& x)
    {
        std::cout
            << '<' << typeid(x).name() << '>'
            << x
            << "</" << typeid(x).name() << '>';
    });
}
```

```
print_xml(stuff);
```

ADAPT A STRUCT

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

BOOST_FUSION_ADAPT_STRUCT(person,
    (std::string, name)
    (int, age)
);
```

DEFINE A STRUCT

```
BOOST_FUSION_DEFINE_STRUCT((), person,
    (std::string, name)
    (int, age)
);
```

```
person p = { "Tom", 52 };
std::string name = at_c<0>(p);
int age = at_c<1>(p);

print_xml(p);
```

MEMBER NAMES

```
std::string name = struct_member_name<person, 0>::call();
std::string age = struct_member_name<person, 1>::call();
```

WITH MEMBER NAMES

- Zip member names with their member values.
- The `person` struct with names would look something like this:

```
std::make_tuple
(
  std::make_tuple(name, "name"),
  std::make_tuple(age, "age")
)
```

```
template<class Struct>
auto with_names(const Struct& s)
{
    using range = range_c<int, 0, (result_of::size<Struct>())>;
    static auto names = transform(range(), [](auto i) -> std::string
    {
        return struct_member_name<Struct, i>::call();
    });
    return zip(s, names);
}
```

```
template<class T>
auto get_value(const T& x)
{
    return at_c<0>(x);
}

template<class T>
std::string get_name(const T& x)
{
    return at_c<1>(x);
}
```

PRINT XML

```
template<class T>
void print_xml(const T& x)
{
    for_each(with_names(x), [](const auto& x)
    {
        std::cout
            << '<' << get_name(x) << '>'
            << get_value(x)
            << "</" << get_name(x) << '>';
    });
}
```

Outputs:

```
<name>Tom</name><age>52</age>
```

ASSOCIATIVE TUPLES

- Maps keys to a value
- Keys are a type

```
namespace fields
{
    struct name
    {};
    struct age
    {};
}

typedef map<
    boost::fusion::pair<fields::name, std::string>
  , boost::fusion::pair<fields::age, int> >
person;
```

```
person a_person = make_map<fields::name, fields::age>("Tom", 52);
std::string person_name = at_key<fields::name>(a_person);
int person_age = at_key<fields::age>(a_person);
```

ADAPTED ASSOCIATIVE SEQUENCE

```
BOOST_FUSION_DEFINE_ASSOC_STRUCT((), person,
    (std::string, name, fields::name)
    (int, age, fields::age)
);
```

WITH MEMBER NAMES AND KEYS

```
template<class Struct>
auto with_names_and_keys(const Struct& s)
{
    typedef range_c<int, 0, result_of::size<Struct>::type::value> range;
    static auto names = transform(range(), [](auto i) -> std::string
    {
        return struct_member_name<Struct, i>::call();
    });
    static auto keys = transform(range(), [](auto i)
    {
        return typename struct_assoc_key<Struct, i>::type();
    });
    return zip(s, names, keys);
}
```

```
template<class T>
auto get_value(const T& x)
{
    return at_c<0>(x);
}

template<class T>
std::string get_name(const T& x)
{
    return at_c<1>(x);
}

template<class T>
auto get_key(const T& x)
{
    return at_c<2>(x);
}
```

ATTRIBUTES

```
// Primary key attribute
struct primary_key {};

// Attribute to specify max length
struct max_length_base {};
template<int N>
struct max_length : max_length_base
{
    static const int max_length_value = N;
};
```

```
namespace fields
{
    struct name : primary_key, max_length<250>
    {};
    struct age
    {};
}

BOOST_FUSION_DEFINE_ASSOC_STRUCT((), person,
    (std::string, name, fields::name)
    (int, age, fields::age)
);
```

CHECK FOR ATTRIBUTE

```
template<class Attribute, class Key>
constexpr auto has_attribute(const Key&)
{
    return std::is_base_of<Attribute, Key>();
}
```

```
namespace fields
{
    struct name : primary_key, max_length<250>
    {};
    struct age
    {};
}

bool with_primary_key = has_attribute<primary_key>(fields::name());
bool without_primary_key = has_attribute<primary_key>(fields::age());
```

```
#define REQUIRES(...) typename std::enable_if<(__VA_ARGS__), int>::type=0

template<class Key, REQUIRES(has_attribute<max_length_base>(Key()))>
int get_max_length(const Key&)
{
    return Key::max_length_value;
}

template<class Key, REQUIRES(!has_attribute<max_length_base>(Key()))>
int get_max_length(const Key&)
{
    return -1;
}
```

CREATE DATABASE TABLE

```
template<class T>
std::string create_table(const T& x, const std::string& table_name)
{
    std::stringstream ss;
    ss << "create table " << table_name;
    char delim = '(';
    for_each(with_names_and_keys(x), [&](const auto& field)
    {
        ss << delim << create_column(field);
        delim = ',';
    });
    ss << ')';
    return ss.str();
}
```

```
template<class T>
std::string create_column(const T& x)
{
    std::stringstream ss;
    ss << std::endl << "    " << get_name(x);
    if (std::is_convertible<decltype(get_value(x)), std::string>())
    {
        int max = get_max_length(get_key(x));
        if (max < 0) ss << " text ";
        else ss << " char(" << max << ")";
    }
    else ss << " " << typeid(get_value(x)).name();
    if (has_attribute<primary_key>(get_key(x))) ss << " primary key";
    return ss.str();
}
```

EXAMPLE

```
person p = { "Tom", 52 };
std::cout << create_table(p, "person");
```

Outputs:

```
create table person(
name char(250) primary key,
age int)
```

THIRD-PARTY LIBRARY

- CPPAN
  - https://github.com/tarasko/cppan
- Boost.Hana
  - https://github.com/ldionne/hana
- Hero
  - https://github.com/pfultz2/Hero

VISITOR PATTERN

- The visitor pattern is a pattern where every class provides a `visit` function which takes a visitor and applies that visitor to all its members

EXAMPLE

```
struct person
{
    std::string name;
    int age;

    struct visit
    {
        template<class Self, class F>
        void operator()(Self&& self, F f) const
        {
            f("name", self.name);
            f("age", self.age);
        }
    };
};
```

- Not a member function so we can use the same `visit` for both `const` and non-`const` visits.

```
template<class T, class F>
void visit_each(T&& x, F f)
{
    using visit = typename std::remove_cv_t<std::remove_reference_t<T>>::visit;
    visit()(std::forward<T>(x), f);
}
```

PRINT XML

```
template<class T>
void print_xml(const T& x)
{
    visit_each(x, [](const std::string& name, const auto& var)
    {
        std::cout
            << '<' << name << '>'
            << var
            << "</" << name << '>';
    });
}
```

```
person p = { "Tom", 52 };
print_xml(p);
```

Outputs:

```
<name>Tom</name><age>52</age>
```

ATTRIBUTES

```
struct max_length
{
    int len;
    max_length(int x) : len(x)
    {}
};

struct primary_key
{};
```

```
struct person
{
    std::string name;
    int age;

    struct visit
    {
        template<class Self, class F>
        void operator()(Self&& self, F f) const
        {
            f("name", self.name, max_length(250), primary_key());
            f("age", self.age);
        }
    };
};
```

```
template<class T>
void print_xml(const T& x)
{
    visit_each(x, [](const std::string& name, const auto& var, const auto&... attributes)
    {
        std::cout
            << '<' << name << '>'
            << var
            << "</" << name << '>';
    });
}
```

```
template<class... Ts>
bool is_primary_key(const primary_key&, const Ts&...)
{
    return true;
}

bool is_primary_key()
{
    return false;
}

template<class T, class... Ts>
bool is_primary_key(const T&, const Ts&...xs)
{
    return is_primary_key(xs...);
}
```

```
template<class... Ts>
int get_max_length(const max_length& x, const Ts&...)
{
    return x.len;
}

int get_max_length()
{
    return -1;
}

template<class T, class... Ts>
int get_max_length(const T&, const Ts&...xs)
{
    return get_max_length(xs...);
}
```

CREATE TABLE

```
template<class T>
std::string create_table(const T& x, const std::string& table_name)
{
    std::stringstream ss;
    ss << "create table " << table_name << "(";
    char delim = '(';
    visit_each(x, [&](const std::string& name, const auto& var, const auto&... attributes)
    {
        ss << delim << create_column(name, var, attributes...);
        delim = ',';
    });
    ss << ')';
    return ss.str();
}
```

```
template<class T, class... As>
std::string create_column(const std::string& name, const T& x, const As&... attributes)
{
    std::stringstream ss;
    ss << std::endl << "    " << name;
    if (std::is_convertible<decltype(x), std::string>())
    {
        int max = get_max_length(attributes...);
        if (max < 0) ss << " text ";
        else ss << " char(" << max << ")";
    }
    else ss << " " << typeid(x).name();
    if (is_primary_key(attributes...)) ss << " primary key";
    return ss.str();
}
```

SELECT VISITS

```
person p = { "Tom", 52 };
visit_select<primary_key>(p, [](const std::string& name,
                                 const auto& x, primary_key)
{
    std::cout << name << ": " << x;
});
```

Outputs:

```
name: Tom
```

SEARCH ATTRIBUTE

```
struct not_found {};

template<class Attribute>
struct attribute_finder
{
    template<class... Ts>
    static const Attribute& call(const Attribute& x, const Ts&...)
    {
        return x;
    }

    static not_found call()
    {
        return {};
    }

    template<class T, class... Ts>
    static auto call(const T&, const Ts&...xs)
    {
        return call(xs...);
    }
};
```

```
template<class Attribute, class F>
auto if_found(const Attribute& attribute, F f)
{
    return f;
}

template<class F>
auto if_found(not_found, F)
{
    // Do nothing
    return [](auto&&...) {};
}
```

```
template<class Attribute, class T, class F>
void visit_select(T&& x, F f)
{
    visit_each(x, [f](const std::string& name, auto&& var, const auto&... attributes)
    {
        const auto& attribute = attribute_finder<Attribute>::call(attributes...);
        if_found(attribute, f)(name, var, attribute);
    });
}
```

DO-IT-YOURSELF

```
struct person
{
    REFLECTABLE
    (
        (std::string) name,
        (int) age
    )
};
```

TYPED EXPRESSIONS

```
#define REM(...) __VA_ARGS__
#define EAT(...)

// Strip off the type
#define STRIP(x) EAT x
// Show the type without parenthesis
#define PAIR(x) REM x
```

```
STRIP((int) x) // Expands to x
PAIR((int) x) // Expands to int x
```

ITERATE OVER ARGUMENTS

```
/* This counts the number of args */
#define NARGS_SEQ(_1,_2,_3,_4,_5,_6,_7,_8,N,...) N
#define NARGS(...) NARGS_SEQ(__VA_ARGS__, 8, 7, 6, 5, 4, 3, 2, 1)

/* This will let macros expand before concating them */
#define PRIMITIVE_CAT(x, y) x ## y
#define CAT(x, y) PRIMITIVE_CAT(x, y)

/* This will call a macro on each argument passed in */
#define EACH(macro, ...) CAT(EACH_, NARGS(__VA_ARGS__))(macro, __VA_ARGS__)
#define EACH_1(m, x0) m(x0)
#define EACH_2(m, x0, x1) m(x0) m(x1)
#define EACH_3(m, x0, x1, x2) m(x0) m(x1) m(x2)
#define EACH_4(m, x0, x1, x2, x3) m(x0) m(x1) m(x2) m(x3)
#define EACH_5(m, x0, x1, x2, x3, x4) m(x0) m(x1) m(x2) m(x3) m(x4)
#define EACH_6(m, x0, x1, x2, x3, x4, x5) m(x0) m(x1) m(x2) m(x3) m(x4) m(x5)
#define EACH_7(m, x0, x1, x2, x3, x4, x5, x6) m(x0) m(x1) m(x2) m(x3) m(x4) m(x5) m(x6)
#define EACH_8(m, x0, x1, x2, x3, x4, x5, x6, x7) m(x0) m(x1) m(x2) m(x3) m(x4) m(x5) m(x
```

```
/* Prepend a comma */
#define PRIMITIVE_COMMA(m) , m
#define COMMA(m) PRIMITIVE_COMMA EAT() (m)

/* Remove the first argument and return its tail */
#define PRIMITIVE_TAIL(x, ...) __VA_ARGS__
#define TAIL(...) PRIMITIVE_TAIL(__VA_ARGS__)

/* Enumerate the macro on each argument */
#define ENUM(macro, ...) TAIL(EACH(COMMA(macro), __VA_ARGS__))

/* This will let macros expand before stringizing them */
#define STRINGIZE(x) PRIMITIVE_STRINGIZE(x)
#define PRIMITIVE_STRINGIZE(x) #x
```

```
#define REFLECTABLE(...) \
EACH(MEMBER_EACH, __VA_ARGS__) \
struct unpack \
{ \
    template<class T, class F> \
    static void apply(T&& self, F&& f) \
    { \
        f(ENUM(UNPACK_EACH, __VA_ARGS__)); \
    } \
};
```

```
#define MEMBER_EACH(x) PAIR(x);

#define UNPACK_EACH(x) \
std::pair<const char*, decltype((self.STRIP(x)))>(STRINGIZE(STRIP(x)), self.STRIP(x))
```

```
template<class C, class F>
void for_each(C&& c, F f)
{
    using unpack = typename std::remove_cv_t<std::remove_reference_t<C>>::unpack;
    unpack::apply(c, [&](auto&&... xs)
    {
        (void)std::initializer_list<int>{
            (f(std::forward<decltype(xs)>(xs)), 0)...
        };
    });
}
```

PRINT XML

```
template<class T>
void print_xml(const T& x)
{
    for_each(x, [](auto data)
    {
        std::cout
            << '<' << data.first << '>'
            << data.second
            << "</" << data.first << '>';
    });
}
```

QUESTIONS?