GOODBYE METAPROGRAMMING AND HELLO FUNCTIONAL: LIVING IN A POST-METAPROGRAMMING ERA IN C++

C++Now 2016

Paul Fultz II

WHY METAPROGRAMMING?

- Value computations
- Type computations
- Introspection

LANGUAGE FEATURES

- decltype
- constexpr
- template aliases
- generic lambdas

OVERVIEW

- Dependent types
- Introspection
- Point free style
- Heterogeneous sequences

DEPENDENT TYPES

- From wikipedia:
  - a dependent type is a type whose definition depends on a value.
- Dependent functions
- Dependent pair

DEPENDENT FUNCTION

```
template<int N>
constexpr std::integral_constant<int, N> make()
{
    return {};
}
```

DEPENDENT FUNCTIONS

- Constructor to `std::integral_constant`
- UDL `_c`

EXAMPLE

```
template<class T, class U>
constexpr auto pick(bool b, T x, U y)
{
    if (b) return x;
    else return y;
}
```

```cpp
template<class T, class U>
constexpr auto pick(std::true_type, T x, U y)
{
    return x;
}

template<class T, class U>
constexpr auto pick(std::false_type, T x, U y)
{
    return y;
}
```

```
template<class IntegralConstant, class T, class U>
constexpr auto pick(IntegralConstant b, T x, U y)
{
    if constexpr(b) return x;
    else return y;
}
```

ADVANTAGES

- More natural syntax
- Avoid `.template` disambguation
- With `decltype` can work with non-constexpr expressions
- Better composability

INDEXING FOR A TUPLE

- Access an element using `std::get<N>(t)`
- Access as a member would be `t.template get<N>()`
- Better to access using index operator `[N]`

```
template<class... Ts>
struct modern_tuple
: std::tuple<Ts...>
{
    using base = std::tuple<Ts...>;
    template<class... Xs>
    modern_tuple(Xs&&... xs) : base(std::forward<Xs>(xs)...)
    {}

    auto operator[](int N)
    {
        return std::get<N>(static_cast<base&>(*this));
    }
};
```

```
template<class... Ts>
struct modern_tuple
: std::tuple<Ts...>
{
    using base = std::tuple<Ts...>;
    template<class... Xs>
    modern_tuple(Xs&&... xs) : base(std::forward<Xs>(xs)...)
    {}

    template<class IntegralConstant>
    auto operator[](IntegralConstant)
    {
        return std::get<IntegralConstant::value>(static_cast<base&>(*this));
    }
};
```

```
auto t = make_modern_tuple(1, 2, 3);
auto x = t[std::integral_constant<int, 2>{}];
```

- Could also write `t[2_c]`

```
auto x = t[std::integral_constant<int, 2>{}];
```

ARRAY SIZE FOR STRUCT MEMBERS

- From Jonathan Adamczewski:

```
template<typename T, std::size_t N>
constexpr std::size_t countof(T const (&)[N]) noexcept
{
  return N;
}

struct S
{
  int a[4];
};

void f(S* s)
{
  constexpr size_t s_a_count = countof(s->a);
  int b[s_a_count];
  // ...
}
```

```
template<typename T, std::size_t N>
constexpr std::integral_constant<std::size_t, N> countof(const T (&)[N]) noexcept
{
  return {};
}

struct S
{
  int a[4];
};

void f(S* s)
{
  constexpr auto s_a_count = decltype(countof(s->a)){};
  int b[s_a_count];
  // ...
}
```

GUIDELINE

- At API endpoints return integral constants
- Avoid `_v` classes in C++
  - Should be integral constant not `bool`
  - Doesn't save typing
  - Extra template instantiations

INTROSPECTION

- Query the structural properties of a type

VOID_T

```
template<class T, class=void>
struct has_foo
: std::false_type
{};

template<class T>
struct has_foo<T, void_t<decltype(std::declval<T>().foo())>
: std::true_type
{};
```

CONDITIONAL OVERLOADING

```
FIT_STATIC_FUNCTION(foo) = conditional(f, g);
```

```
if (is_callable<F, Ts...>()) f(xs...);
else if (is_callable<G, Ts...>()) g(xs...);
```

```
FIT_STATIC_LAMBDA_FUNCTION(has_foo) = conditional(
    [](auto x) -> decltype(x.foo(), std::true_type{})
    {
        return {};
    },
    [](auto x) -> std::false_type { return {}; }
);
```

IMPLEMENTATION OF STRINGIFY

```
FIT_STATIC_LAMBDA_FUNCTION(stringify) = conditional(
    [](const auto& x) FIT_RETURNS(std::to_string(x)),
    [](const auto& x) FIT_RETURNS((std::ostringstream() << x).str())
);
```

COMPOSABILITY

```
FIT_STATIC_LAMBDA_FUNCTION(serialize) = conditional(
    [](auto x) FIT_RETURNS(stringify(x)),
    [](auto x) FIT_RETURNS(x.serialize())
);
```

EXAMPLE OF GENERIC FIND

```
FIT_STATIC_LAMBDA_FUNCTION(find_iterator) = conditional(
    [](const std::string& s, const auto& x)
    {
        auto index = s.find(x);
        if (index == std::string::npos) return s.end();
        else return s.begin() + index;
    },
    [](const auto& r, const auto& x) FIT_RETURNS(r.find(x)),
    [](const auto& r, const auto& x)
    {
        using std::begin;
        using std::end;
        return std::find(begin(r), end(r), x);
    }
);
```

```
// ADL Lookup for ranges
namespace adl {

using std::begin;
using std::end;

template<class R>
auto adl_begin(R&& r) FIT_RETURNS(begin(r));
template<class R>
auto adl_end(R&& r) FIT_RETURNS(end(r));
}
```

```
FIT_STATIC_LAMBDA_FUNCTION(find_iterator) = conditional(
    [](const std::string& s, const auto& x)
    {
        auto index = s.find(x);
        if (index == std::string::npos) return s.end();
        else return s.begin() + index;
    },
    [](const auto& r, const auto& x) FIT_RETURNS(r.find(x)),
    [](const auto& r, const auto& x) -> decltype(adl::adl_begin(r), adl::adl_end(r))
    {
        using std::begin;
        using std::end;
        return std::find(begin(r), end(r), x);
    }
);
```

CUSTOMIZATION POINTS

STRONG AND WEAK CUSTOMIZATION POINTS

```
FIT_STATIC_LAMBDA_FUNCTION(find_iterator) = conditional(
    [](const auto& r, const auto& x) FIT_RETURNS(strong_find(r, x)),
    [](const std::string& s, const auto& x)
    {
        auto index = s.find(x);
        if (index == std::string::npos) return s.end();
        else return s.begin() + index;
    },
    [](const auto& r, const auto& x) FIT_RETURNS(r.find(x)),
    [](const auto& r, const auto& x) -> decltype(adl::adl_begin(r), adl::adl_end(r))
    {
        using std::begin;
        using std::end;
        return std::find(begin(r), end(r), x);
    },
    [](const auto& r, const auto& x) FIT_RETURNS(weak_find(r, x))
);
```

DESIGN BY INTROSPECTION

- From Andrei Alexandrescu's allocators in D:
  - Make all allocation primitives optional, except `allocate` and `alignment`
  - All others optional, probed introspectively:
    - `deallocate`
    - `own`

```
template<class P, class F>
struct fallback_allocator
```

```
FIT_STATIC_LAMBDA_FUNCTION(try_deallocate) = conditional(
    [](auto&& a, blk b) FIT_RETURNS(a.deallocate(b)),
    always()
);

FIT_STATIC_LAMBDA_FUNCTION(owns) = [](auto&& a, blk b) FIT_RETURNS(a.owns(b));
```

```
blk allocate(size_t n)
{
    auto r = primary.allocate(n);
    if (!r.ptr) r = fallback.allocate(n);
    return r;
}
```

```
template<class Block>
auto deallocate(Block b) FIT_RETURNS
(
    owns(primary, b) ?
        try_deallocate(primary, b) :
        try_deallocate(fallback, b)
);
```

```
template<class Block>
auto owns(Block b) FIT_RETURNS(owns(primary, b) || owns(fallback, b));
```

POINT FREE STYLE

- A style where the arguments(or points) to the function are not explicity defined.

VARIDIAC PRINT

```
print("Hello", "World\n");
```

```
// Base case
void print()
{}

template<class T, class... Ts>
void print(const T& x, const Ts&... xs)
{
    std::cout << x;
    print(xs...);
}
```

BY ADAPTOR

```
assert(by(p)(xs...) == p(xs)...);

assert(by(p, f)(xs...) == f(p(xs)...));
```

```
FIT_STATIC_FUNCTION(simple_print) = FIT_LIFT(std::ref(std::cout) << _);

FIT_STATIC_FUNCTION(print) = by(simple_print);
```

```
using newline_t = std::integral_constant<char, '\n'>;
FIT_STATIC_FUNCTION(print_lines) = by(capture(newline_t{})(flip(print)));
```

```
print_lines("Hello", "World");
```

```
Hello
World
```

VARIDIAC MAX

```
auto n = max(1, 2, 4, 3); // Returns 4
auto m = max(0.1, 0.2, 0.5, 0.4); // Returns 0.5
```

```
// Base case
template<class T>
T max(const T& x)
{
    return x;
}

template<class T, class... Ts>
T max(const T& x, const T& y, const Ts&... xs)
{
    return std::max(x, max(y, xs...));
}
```

COMPRESS ADAPTOR

```
assert(compress(f)(x) == x);
assert(compress(f)(x, y, xs...) == compress(f)(f(x, y), xs...));
```

```
FIT_STATIC_FUNCTION(max) = compress(FIT_LIFT(std::max));
```

HETEROGENEOUS SEQUENCES

UNPACK SEQUENCE

```
auto check_equal = [](auto x, auto y) { return x == y; };
assert(fit::unpack(check_equal)(std::make_tuple(1, 1)));
assert(fit::unpack(check_equal)(fit::pack(1, 1)));
```

UNPACK STRUCT

```
struct point
{
    int x;
    int y;

    point(int x, int y) : x(x), y(y)
    {}
};

namespace fit {
    template<>
    struct unpack_sequence<point>
    {
        template<class F, class P>
        static auto apply(F&& f, P&& p) FIT_RETURNS
        (
            f(p.x, p.y)
        );
    };
}
```

```
assert(fit::unpack(check_equal)(point(1, 1)));
```

TRANSFORM

```
FIT_STATIC_LAMBDA_FUNCTION(tuple_transform) = [](auto&& sequence, auto f)
{
    return unpack(by(f, construct<std::tuple>()))
        (std::forward<decltype(sequence)>(sequence));
};
```

```
auto t = std::make_tuple(1, 2);
auto r = tuple_transform(t, [](int i) { return i*i; });
assert(r == std::make_tuple(1, 4));
```

POINT FREE

```
FIT_STATIC_FUNCTION(by_tuple) = capture(construct<std::tuple>())(flip(by));

FIT_STATIC_FUNCTION(tuple_transform) =
    flip(compress(apply, compose(unpack, by_tuple)));
```

FOREACH

```
FIT_STATIC_LAMBDA_FUNCTION(tuple_for_each) = [](auto&& sequence, auto f)
{
    return unpack(by(f))(std::forward<decltype(sequence)>(sequence));
};
```

FOLD

```
FIT_STATIC_LAMBDA_FUNCTION(tuple_fold) = [](auto&& sequence, auto f)
{
    return unpack(compress(f))(std::forward<decltype(sequence)>(sequence));
};
```

MONAD

- `tuple_yield`: `T -> tuple<T>`
- `tuple_for`: `tuple<Ts...>, (T -> tuple<Us...>) -> tuple<Us......>`
  - `compose(join, transform)`
    - `tuple_transfom`: `tuple<Ts...>, (T -> tuple<Us...>) -> tuple<tuple<Us...>...>`
    - `tuple_join`: `tuple<tuple<Us...>...> -> tuple<Us......>`

```
assert(fit::unpack(check_equal)(std::make_tuple(1), std::make_tuple(1)));
```

```
FIT_STATIC_FUNCTION(tuple_cat) = unpack(construct<std::tuple>());
```

```
FIT_STATIC_FUNCTION(tuple_join) = unpack(tuple_cat);
```

```
FIT_STATIC_LAMBDA_FUNCTION(tuple_for) = compose(tuple_join, tuple_transform);

FIT_STATIC_FUNCTION(tuple_yield) = construct<std::tuple>();
```

```
FIT_STATIC_LAMBDA_FUNCTION(tuple_yield_if) = [](auto c) FIT_RETURNS
(
    conditional(
        if_(c)(construct<std::tuple>()),
        always(std::tuple<>())
    )
);
```

```
FIT_STATIC_LAMBDA_FUNCTION(tuple_filter) = [](auto&& sequence, auto predicate)
{
    return tuple_for(std::forward<decltype(sequence)>(sequence), [&](auto&& x)
    {
        return tuple_yield_if(predicate(std::forward<decltype(x)>(x)))
            (std::forward<decltype(x)>(x));
    });
};
```

```
auto t = std::make_tuple(1, 2, 'x', 3);
auto r = tuple_filter(t, [](auto x) { return std::is_same<int, decltype(x)>(); });
assert(r == std::make_tuple(1, 2, 3));
```

ZIP

- Use `combine` adaptor:

```
assert(combine(f, gs...)(xs...) == f(gs(xs)...));
```

```
FIT_STATIC_LAMBDA_FUNCTION(tuple_zip_with) = [](auto&& sequence1, auto&& sequence2,
{
    auto&& functions = tuple_transform(
        std::forward<decltype(sequence1)>(sequence1),
        [&](auto&& x)
        {
            return [&](auto&& y)
            {
                return f(std::forward<decltype(x)>(x), std::forward<decltype(y)>(y))
            };
        }
    );
    auto combined = unpack(capture(construct<std::tuple>())(combine))(functions);
    return unpack(combined)(std::forward<decltype(sequence2)>(sequence2));
};
```

```
auto t1 = std::make_tuple(1, 2);
auto t2 = std::make_tuple(3, 4);
auto p = tuple_zip_with(t1, t2, [](auto x, auto y) { return x*y; });
int r = tuple_fold(p, [](auto x, auto y) { return x+y; });
assert(r == (1*3 + 4*2));
```

CONCLUSION

QUESTIONS

https://gitlab.com/pfultz2/cppnow2016