AUTOINTERN

Immer eine zündende Idee.

Most important programming guideline:

Make interfaces easy to use correctly and hard to use incorrectly,

but achieving it can be challenging.

\- Scott Meyers -

Professional Explainer

AUTOINTERN
Immer eine zündende Idee.

Who Am I and What Am I Doing Here?

- C++ Evangalist
- Template Meta Programming Nerd
- Embedded Developer
- Kvasir.io / brigand / SG14 contributor
- Part of an awesome development team!
- Contract developer / Consultant

Library paradigm shifts due to modern metaprogramming

Harnassing the Dragons of C++‘s built in code generator

AUTOINTERN
Immer eine zündende Idee.

What problems does metaprogramming solve?

Different tools solve different things:

- anything that results in a value should be done with constexpr
- anything that results in a type (including different code gen.) should be done with templates
- anything that is meant to sabotage a codebase should be done with macros

AUTOINTERN
Immer eine zündende Idee.

Encapsulation of expertise

- How does std::tie work?
- What optimization is used in std::find() with random access char iterators?
- Is this valid:

```
quantity<length> L = 2.0*meters;// quantity of length
quantity<energy> E = kilograms*pow<2>(L / seconds);
```

?

I don’t know! I don’t have to know, its encapsulated.

What problems does metaprogramming cause?

- Ugly, scary compiler errors
- Code that is impossible for novice users to understand

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

-Brian W. Kernighan and P. J. Plauger in The Elements of Programming Style.

Template Errors are Scary

AUTOINTERN
Immer eine zündende Idee.

Templates

```
template<typename T, int I>
struct Array {
    T data_[I];
    // implementation here
};

template<typename T>
T square(T in) {
    return in*in;
}
```

Using

```
typedef std::vector<int> intVec;

using intVec2 = std::vector<int>;

template<unsigned I>
using intArray = std::array<int, I>;

intArray<4> ia;
```

Variadic Templates

```
template<typename... Ts>
void myPrintf(std::string s, Ts...args) {
   printf(s.c_str(), args...);
}

template<typename... Ts>
struct S : Ts... {};
```

Data Storage

```
struct S {
   friend int getI(const S& s);
   friend bool getB(const S& s);
   S(int i, bool b):i_{i},b_{b} {}
private:
   int i_;
   bool b_;
};
int getI(const S s){
    return s.i_; }
bool getB(const S s){
    return s.b_; }

S myS(4, false);
auto i = getI(myS);
```

```
template<int I, bool B>
struct S {};

template<typename T>
struct GetI;
template<int I, bool B>
struct GetI<S<I, B>> {
   static constexpr int value = I;
}

using MyS = S<4, false>;
auto i = GetI<MyS>::value;
```

Composition

```
template<typename T, bool B>
struct S2 {};

using myS2 = S2<MyS, false>;

template<int I, bool B, bool B2>
struct GetI<S2<S<I, B>, B2>> {
   static constexpr int value = I;
}
```

If / Switch

```
int f(int i, bool b) {
  if (b) {
    switch (i) {
    default:
      return 99;
    case 42:
      return 1
    };
  }
  return 22;
}

int i = f(42);
```

```
template<int I, bool B>
struct F {
  static constexpr int value = 22;
};

template<int I>
struct F<I, true> {
  static constexpr int value = 99;
};

template<>
struct F<42, true> {
  static constexpr int value = 1;
};

int i = F<42>::value;
```

Containers / Loops

AUTOINTERN
Immer eine zündende Idee.

```
template<typename...Ts>
struct list {};

template<int I, typename T>
struct at_impl;
template<int I, typename T, typename...Ts>
struct at_impl<I, list<T, Ts...>> : at_impl<I - 1, list<Ts...>>{};
template<typename T, typename... Ts>
struct at_impl<0, list<T, Ts...>> { using type = T; };

template< typename T, int I >
using at_c = typename at_impl<I, T>::type;

using L = list<int, bool, float, bool>;

at_c< L , 2 > f = 1.4;
```

AUTO INTERN
Immer eine zündende Idee.

Beware of slide 13, the horror begins

Fast Tracked

AUTOINTERN
Immer eine zündende Idee.

```
template<bool Big, int I, typename T>
struct at_impl;
template<int I, typename T0, typename T1, typename T2, typename T3,
    typename T4, typename T5, typename T6, typename T7, typename T8,
    typename T9, typename T10, typename T11, typename T12, typename T13,
    typename T14, typename T15, typename...Ts>
struct at_impl<true, I, list<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10,
    T11, T12, T13, T14, T15, Ts...>> : at_impl<(I>32), I - 16, list<Ts...>>
    {};
template<int I, typename T, typename...Ts>
struct at_impl<false, I, list<T, Ts...>> :
    at_impl<false, I - 1, list<Ts...>> {};
template<typename T, typename... Ts>
struct at_impl<false, 0, list<T, Ts...>> { using type = T; };

template<typename L, int I>
using at_c = typename at_impl<(I>16), I, L>::type;
```

Thermometer encoding

AUTOINTERN
Immer eine zündende Idee.

```
template<class T> struct element_at;
template<class... Ts>
struct element_at<list<Ts...>>{
    template<class T>
    type_<T> static at(Ts..., type_<T>*, ...);
};

template<std::size_t N, typename Seq> struct at_impl;
template<std::size_t N, template<typename...> class L, class... Ts>
struct at_impl<N, L<Ts...>> :
    decltype(element_at<brigand::filled_list<void const *,N>>::
        at(static_cast<type_<Ts>*>(nullptr)...)){};

template <class L, std::size_t Index>
using at_c = typename detail::at_impl<Index, L>::type;
```

Inheritance

AUTOINTERN
Immer eine zündende Idee.

```
template<size_t N, size_t I, typename T>
struct type_on_match {};

template<size_t N, typename T>
struct type_on_match<N, N, T> { using type = T; };

template<size_t N, typename Indexes, typename Seq>
struct at_impl;
template<size_t N, size_t...Is, class<class...>
    class L,class... Ts>
struct at_impl<N, integer_sequence<size_t,Is...>, L<Ts...>> :
    type_on_match<N, Is, Ts>... {};

template <class L, size_t Index>
using at_c = typename at_impl<Index,
    make_integer_sequence<size_t, wrap<L,count>::value>, L>::type;
```

Function Specialized Inheritance

AUTO INTERN
Immer eine zündende Idee.

```
template<size_t N, typename T>
struct indexed_type {};

template<typename Indexes, typename Seq> struct at_impl;
template<size_t...Is, template<typename...> class L, class... Ts>
struct at_impl< integer_sequence<size_t, Is...>, L<Ts...>> :
    indexed_type<Is, Ts>...{};

template<size_t N> struct fetcher {
    template<typename T> static T fetch(indexed_type<N, T>*) {}
};
template<typename L>
using seq_of_length = make_index_sequence<wrap<L, count>::value>;

template <class L, size_t Index>
using at_c = decltype(fetcher<Index>::fetch(
    static_cast<at_impl<seq_of_length<L>, L> *>(nullptr)));
```

Timing

Nieve:	1810 ms
Fast tracked:	270 ms
Thermometer encoding:	130 ms
Inheritance:	250 ms
Function specialized inheritance:	30 ms
Mpl.vector (on 10x shorter lists)	960 ms

Thermometer encoding

AUTOINTERN
Immer eine zündende Idee.

```
template<class T> struct element_at;
template<class... Ts>
struct element_at<list<Ts...>>{
    template<class T>
    type_<T> static at(Ts..., type_<T>*, ...);
};

template<std::size_t N, typename Seq> struct at_impl;
template<std::size_t N, template<typename...> class L, class... Ts>
struct at_impl<N, L<Ts...>> :
    decltype(element_at<brigand::filled_list<void const *,N>>::
        at(static_cast<type_<Ts>*>(nullptr)...)){};

template <class L, std::size_t Index>
using at_c = typename detail::at_impl<Index, L>::type;
```

Timing

Nieve:	1810 ms
Fast tracked:	270 ms (250 ms)
Thermometer encoding:	130 ms
Inheritance:	250 ms
Function specialized inheritance:	30 ms
Mpl.vector (on 10x shorter lists)	960 ms

What about memorization?

This benchmark searched for 25 random but unique indexes in the same list of 500 elements

Fast Tracked

AUTOINTERN
Immer eine zündende Idee.

```
template<bool Big, int I, typename T>
struct at_impl;
template<int I, typename T0, typename T1, typename T2, typename T3,
    typename T4, typename T5, typename T6, typename T7, typename T8,
    typename T9, typename T10, typename T11, typename T12, typename T13,
    typename T14, typename T15, typename...Ts>
struct at_impl<true, I, list<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10,
    T11, T12, T13, T14, T15, Ts...>> : at_impl<(I>32), I - 16, list<Ts...>>
    {};
template<int I, typename T, typename...Ts>
struct at_impl<false, I, list<T, Ts...>> :
    at_impl<false, I - 1, list<Ts...>> {};
template<typename T, typename... Ts>
struct at_impl<false, 0, list<T, Ts...>> { using type = T; };

template<typename L, int I>
using at_c = typename at_impl<(I>16), I, L>::type;
```

AUTOINTERN

Immer eine zündende Idee.

Timing Unique

Nieve:	1810 ms
Fast tracked:	270 ms
Thermometer encoding:	130 ms
Inheritance:	290 ms
Function specialized inheritance:	220 ms
Mpl.vector (on 10x shorter lists)	not tested

Searching for 25 random indexes in 25 unique lists

Function Specialized Inheritance

AUTO INTERN
Immer eine zündende Idee.

```
template<size_t N, typename T>
struct indexed_type {};

template<typename Indexes, typename Seq> struct at_impl;
template<size_t...Is, template<typename...> class L, class... Ts>
struct at_impl< integer_sequence<size_t, Is...>, L<Ts...>> :
    indexed_type<Is, Ts>...{};

template<size_t N> struct fetcher {
    template<typename T> static T fetch(indexed_type<N, T>*) {}
};
template<typename L>
using seq_of_length = make_index_sequence<wrap<L, count>::value>;

template <class L, size_t Index>
using at_c = decltype(fetcher<Index>::fetch(
    static_cast<at_impl<seq_of_length<L>, L> *>(nullptr)));
```

Sort Timing

Brigand 500:	2.51s
Brigand 250:	0.96s
Quicksort 500:	crashes
Quicksort 250:	12.86s
Hana 500:	crashes
Hana 250:	9.31s
Mpl.sort 50:	longer than getting a mate tea

MPL Lambdas

- Awesome!
- Allow composition of metafunctions elevating the level of abstraction
- Slow
- Sloooooooooooooooow
- “Eager” functions don’t work
- Nesting algorithms does not work
- Sloooooooooooooooooooooooooooooow

Leagacy boost.MPL

```
// find the position of a type x in some_sequence such that:
//         x is convertible to 'int'
//      && x is not 'char'
//      && x is not a floating type
typedef mpl::find_if<
    some_sequence,
    mpl::and_<
        boost::is_convertible<_1, int>,
        mpl::not_<boost::is_same<_1, char>>,
        mpl::not_<boost::is_float<_1> >
    >
>::type iter;
```

Leagacy boost.MPL

```
template <int N>
struct arg; // forward declaration

template <> struct arg<1>{
    template <class A1, class... Am>
    struct apply{
        typedef A1 type; // return the first argument
};};
typedef arg<1> _1;

template <> struct arg<2>{
    template <class A1, class A2, class... Am>
    struct apply{
        typedef A2 type; // return the second argument
};};
typedef arg<2> _2;
```

Error:

“” does not satisfy the concept “Food”

Leagacy boost.MPL

```
template <int N>
struct arg; // forward declaration

template <> struct arg<1>{
    template <class A1, class... Am>
    struct apply{
        typedef A1 type; // return the first argument
};};
typedef arg<1> _1;

template <> struct arg<2>{
    template <class A1, class A2, class... Am>
    struct apply{
        typedef A2 type; // return the second argument
};};
typedef arg<2> _2;
```

Bind for eager lambdas

```
template <template<typename...> class F, typename...Ts>
struct bind<F,Ts...>{};
```

```
bind< algorithm, parameter1, parameter2, parameter…>

//equivalent to

algorithm<parameter1, parameter2 parameter…>
```

Named Parameters

```
//we can capture default value as a constexpr
constexpr auto length = p2::make_tag<1>(4);

//we can also use "wrapped" default values as long as they are convertable
constexpr p2::tag<2, p2::convertible<int>,
    std::integral_constant<int,9>> height{};
//if all else fails we can specify a functor which makes our default value
constexpr p2::tag<3, p2::convertible<int>, DepthMaker> depth{};

//we can also use reference wrappers for in out parameters
constexpr p2::tag<4,
    p2::convertible<std::reference_wrapper<int>>> outInt{};
```

Using Named Parameters

```
template<typename...Ts>
void draw(Ts&&...args) {
//make tuple uses D style syntax, compile time values as first arg list and runtime
//in the second arg list. First arg list must contain all name tags in the desired
//positional order. Second list must be the args passed by the user
    auto ta = p2::make_tuple(length, height, depth)(std::forward<Ts>(args)...);
    auto h = ta[height];  //input parameter indexing is trivial
    auto& l = ta[length];
    l = 99;
    auto l2 = ta[length];
    auto d = ta[depth];
}

template<typename...Ts>
void f(Ts&&...args) {
    auto ta = p2::make_tuple(outInt)(std::forward<Ts>(args)...);
    ta[outInt] = 4;
}
```

AUTOINTERN
Immer eine zündende Idee.

Complex Real World Lambda Use

```
using no_tagged_provided = remove_if<
    has_no_default,      //list of arg types that have no default
    bind<
        contains,      //algorithm
        pin<arg_tag_indicies>,   //list of user provided tag indexes
        get_index<_1>>>;      //lambda
Constexpr int unmatched_defaults = size<
    remove_if<
        no_tagged_provided,
        bind<
            any,
            pin<non_positional_args>,
            defer<
                value_fulfills_tag<
                    parent<_1>,
                    _1>>>>>::value;
```

What is Fast and What is Slow?

Using memorized type (essentially free)

Using an alias rather than aliased type directly

Wrapping a type

Adding an extra specialization

Calling a metafunction by inheritance

Calling a metafunction

Calling a nested alias

Calling a nested metafunction

Using SFINAE

New Lambda Backend

```
template <typename T, typename... Ls>
struct apply {
    using type = T;  //default is interpreted as if it were a pin<T>
};
//eager call case
template <template<typename...> class F, typename...Ts, typename... Args>
struct apply<bind<F,Ts...>, Args...>{
    using type = F<typename apply<Ts, Args...>::type...>;
};
//lazy call cases
template <template <class...> class F, class... Ts, class L, class... Ls>
struct apply<F<Ts...>, L, Ls...> :
    F<typename apply<Ts, L, Ls...>::type...>{};
//pin case
template <typename T, typename... Args, typename...Ls>
struct apply<pin<T>, list<Args...>, Ls...>{
    using type = T;
};
```

…continued

```
//arg case
template <std::size_t N, typename L, typename...Ls>
struct apply<args<N>, L, Ls...> {
    using type = at_c<L, N>;
};
//arg fast track
template <typename T, typename...Ts, typename...Ls>
struct apply<_1, list<T, Ts...>, Ls...> {
    using type = T;
};
//arg fast track
template <typename T, typename U, typename...Ts,
typename...Ls>
struct apply<_2, list<T, U, Ts...>, Ls...> {
    using type = U;
};
```

…continued

```
//defer case
template <typename Lambda, typename L, typename...Ls>
struct apply<defer<Lambda>, L, Ls...>{
    using type = packaged_lcall<Lambda, L, Ls...>;
};

//packaged_lcall case
template <template <typename...> class Lambda, typename... Ts,
    typename... PLs, typename L, typename...Ls>
struct apply<packaged_lcall<Lambda<Ts...>, PLs...>, L, Ls...> :
    Lambda<typename apply<Ts, L, Ls..., PLs...>::type...>{};

//parent case
template <typename T, typename L, typename...Ls>
struct apply<parent<T>, L, Ls...> :
    apply<T,Ls...>{};
```

Peter Dimov and mp11

```
template<class L, template<class...> class P, class W>
struct mp_replace_if_impl{
    template<class T> using _f = mp_if<P<T>, W, T>;

    using type = mp_transform<_f, L>;
};
```

AUTO INTERN
Immer eine zündende Idee.

LSM

```
auto sm = LSM::make(data = myData(db, queue),
    transition(“Begin"_s >> “End"_s>,
        guard = [](auto& c, StartQuiery& q) {
            c.super.setQuieryString(q.string);
        },
        rollback = [](auto& c) { c.super.postEv(DBFailed{}); },
        findDatabase,
        guard = databaseFound,
        openDatabase,
        guard = success,
        scopeGuard = [](auto& c) { c.super.db.close(); }
        openTable,
        guard = success,
        rollback = closeTable,
        performQuery,
        guard = [](QType& ev) { return ev.event.value() == "something"; },
        performOtherQuirey,
        guard = success,
        [](auto& c) { c.super.postEv(DBSuccess{}); }
));
```

LSM is Just a Front End

- Will use boost.MSM-lite as backend
- Merges all contiguous lists of actions
- Generates anonymous states
- Generates a “catch all guard” as the negation of each guard set
- Generates “rollback” transitions for the “catch all guard” case
- Translates everything to MSM-lite style transition table

D-Style Template Functions/Classes

```
template<typename T>
struct identity{};  //found in any good MPL

template<typename T, typename U>
struct concrete_thing { concrete_thing(int i){} };

template<template<typename...> class C, typename...T>
struct thing_constructor {
    template<typename...U>
    C<T...> operator()(U&&...args) { return C<T...>{forward<U>(args)...}; }
};

template<typename...T>
auto thing(const identity<T>...)->thing_constructor<concrete_thing, T...> {return{};}

constexpr identity<int> tagA{};
constexpr identity<bool> tagB{};

auto t = thing(tagA, tagB)(1);  //D-Style compile time args first, runtime args second
```

Parsing User Defined Literals

```
namespace literals {
    template <char ...c>
    constexpr auto operator"" _c() {
        return llong_c<ic_detail::parse<sizeof...(c)>({ c... })>;
    }
}
```

Parsing User Defined Literals

```
template<std::size_t N>
constexpr long long parse(const char(&arr)[N]) {
    long long base = 10;
    std::size_t offset = 0;
    if (N > 2) {
        bool starts_with_zero = arr[0] == '0';
        bool is_hex = starts_with_zero && arr[1] == 'x';
        bool is_binary = starts_with_zero && arr[1] == 'b';

        if (is_hex) { base = 16; offset = 2; }
        else if (is_binary) { base = 2; offset = 2; }
        else if (starts_with_zero) { base = 8; offset = 1; }
    }
```

AUTOINTERN
Immer eine zündende Idee.

Parsing User Defined Literals

```
    long long number = 0; long long multiplier = 1;
    for (std::size_t i = 0; i < N - offset; ++i) {
        char c = arr[N - 1 - i];
        number += to_int(c) * multiplier;
        multiplier *= base;
    }
    return number;
}
```

AUTOINTERN
Immer eine zündende Idee.

Policy Based Class Design

- Awesome!
- Ultimate flexibility
- Decomposition can be hard
- Cyclic dependencies are hard
- Ultimately requires TMP ability

Policy Based Class Design

```
template <typename TDeviceSettings, typename... TDeviceClass>
   struct Device
   {
       using PacketType = typename TDeviceSettings::MemoryPolicy::PacketType;
       using AllocatorType = typename TDeviceSettings::MemoryPolicy::AllocType;
       using OutputQueueType = typename TDeviceSettings::MemoryPolicy::QueueType;
       using TransferType = typename TDeviceSettings::MemoryPolicy::TransferType;

       using DeviceClasses = brigand::list<
           brigand::apply<typename TDeviceClass::ClassType, TDeviceClass, Device>...>;
       using EndpointRequirements = list<typename TDeviceClass::EPRequirements...>;
       using FlatEPRequirements = brigand::flatten<EndpointRequirements>;
       using EndpointNumbers = MapCapabilitiesToEndpointNumbers<
           typename TDeviceSettings::PeripheralNumber, FlatEPRequirements>;
```

Policy Based Factories

```
namespace clk = System::clock;
using namespace clk::tags;
auto clockMaster = clk::make(source = clk::ext,//external
    exFreq = 12'000'000_c,              //12 mhz
    inFreq = 48'000'000_c);             //48 mhz
auto myTimer = timeServer::make(clockMaster,   //compile time clock settings
    hw = timer1::peripheral,            //which timer do we use
    memory::policy::singleton);             //how do we get our RAM
auto myUart = uart::make(bustiming, baud = 9600_c, autobaudCapability);
auto myPool = memory::pool::make(packetSize = 100_c, poolSize = 100_c);
auto myIoDevice = IODevice::make(hw = myUart,
    timer = myTimer,  //data must be sent in chunks if larger than the
                      //output buffer so we need a callback timer
    queue = memory::queue::intrusive::fifo::make(myPool, 1_c));

copy("hello world", myIODevice::tx);
```

Questions

AUTO INTERN

Immer eine zündende Idee.

Odin Holmes

holmes@auto-intern.de