

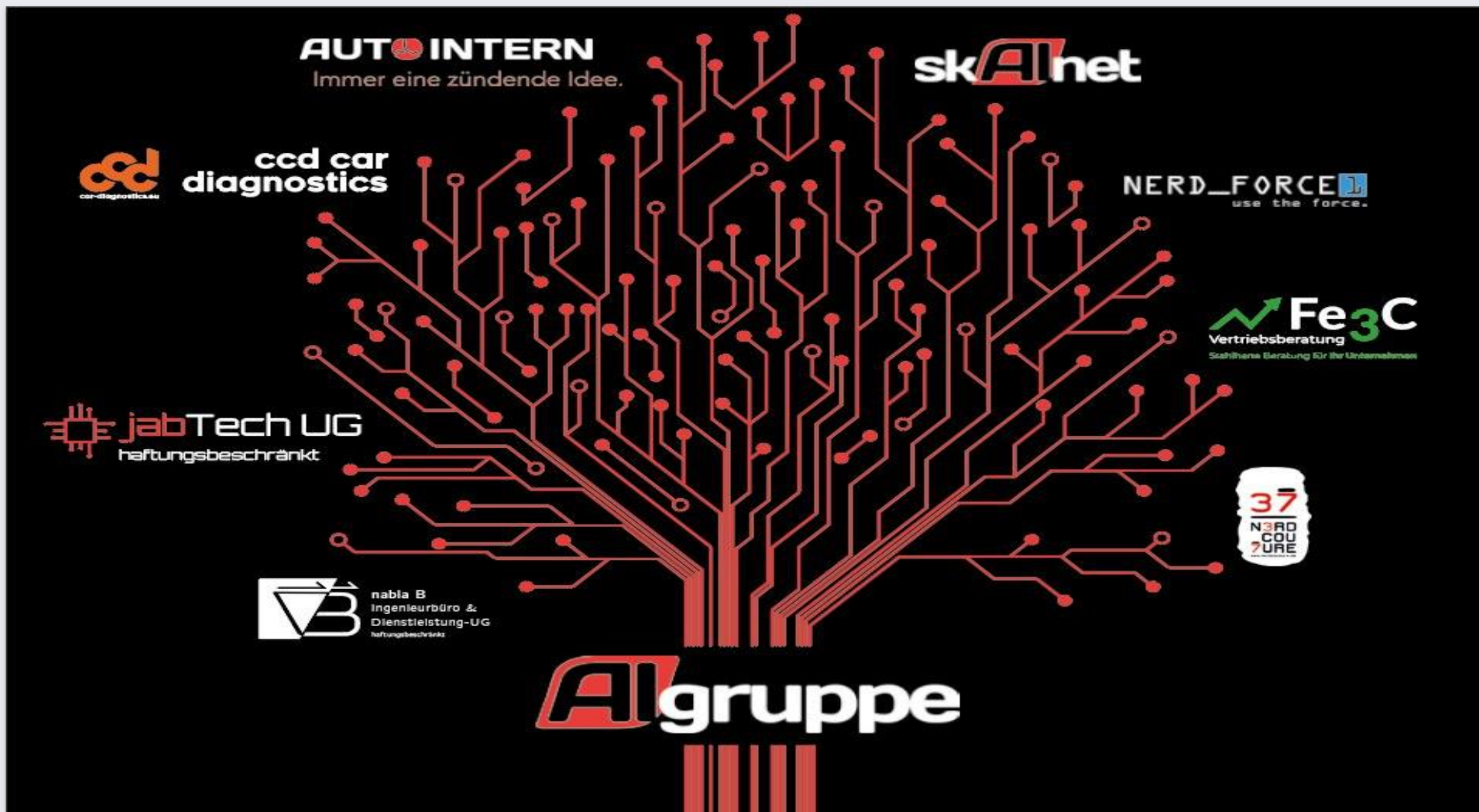


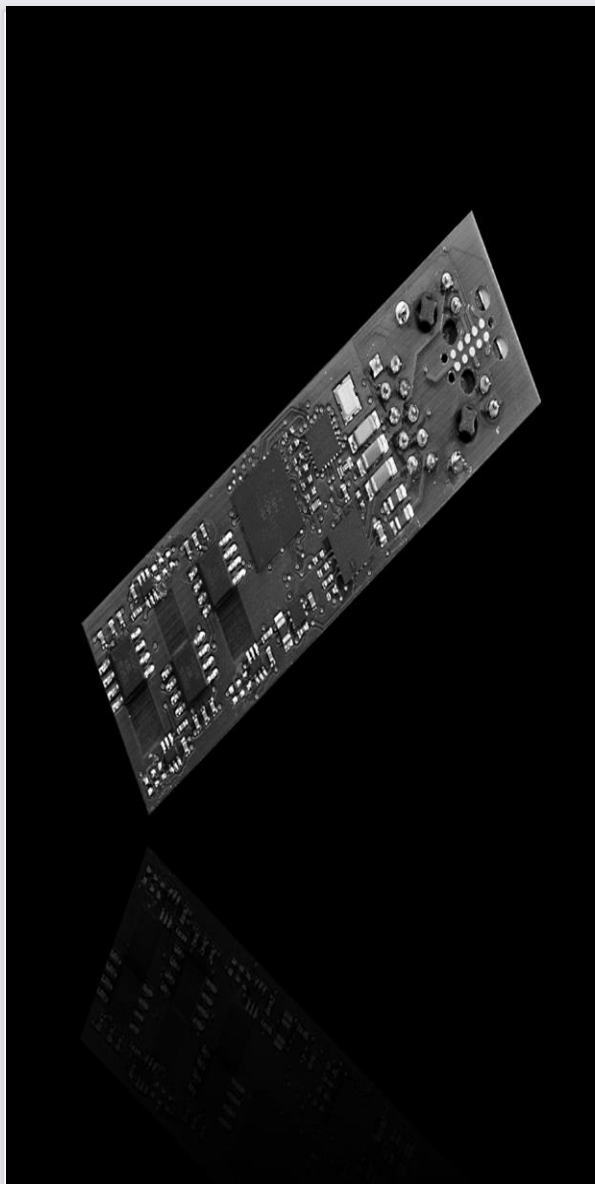
# @odin**th**e**ne**rd

– not the god

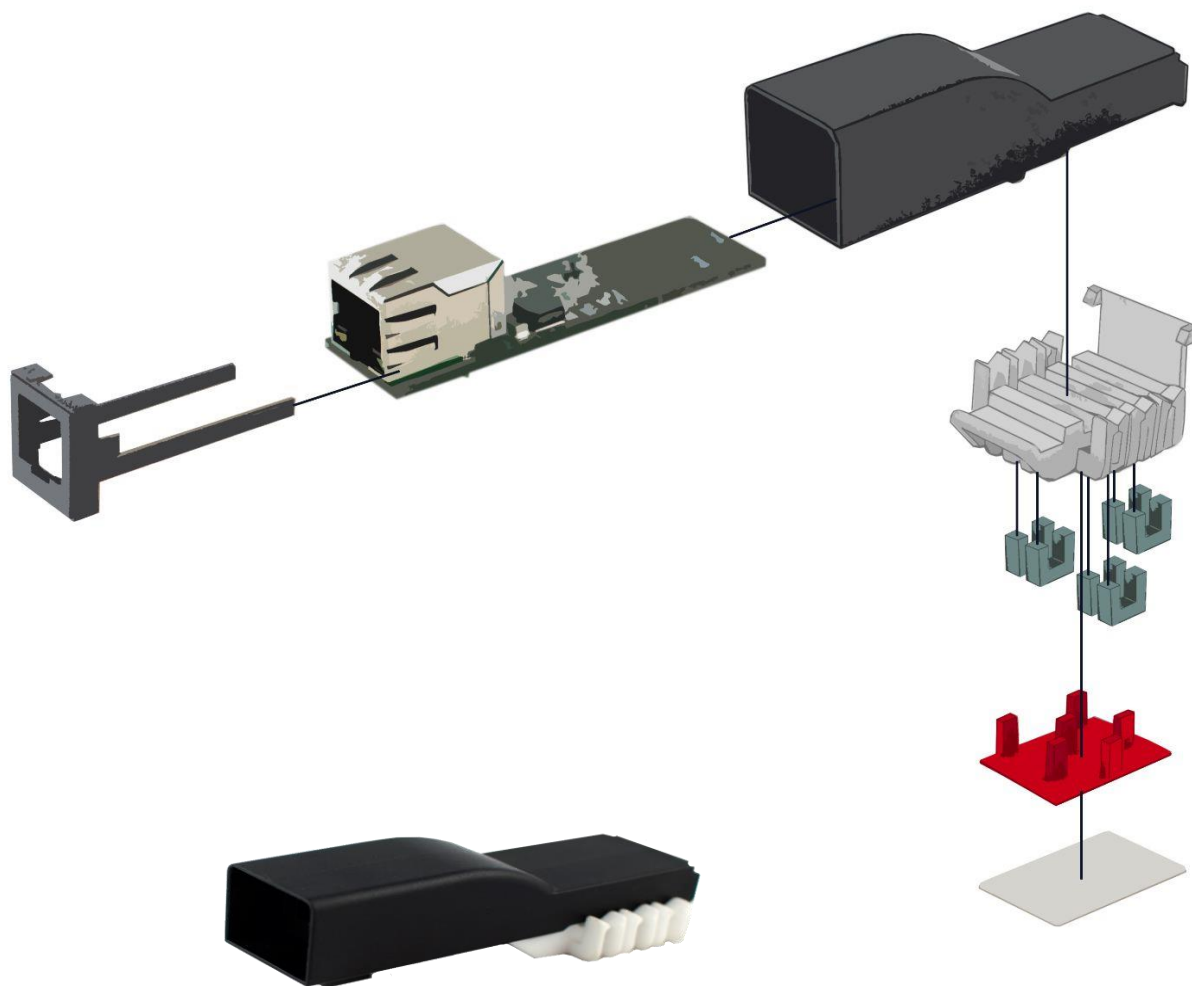


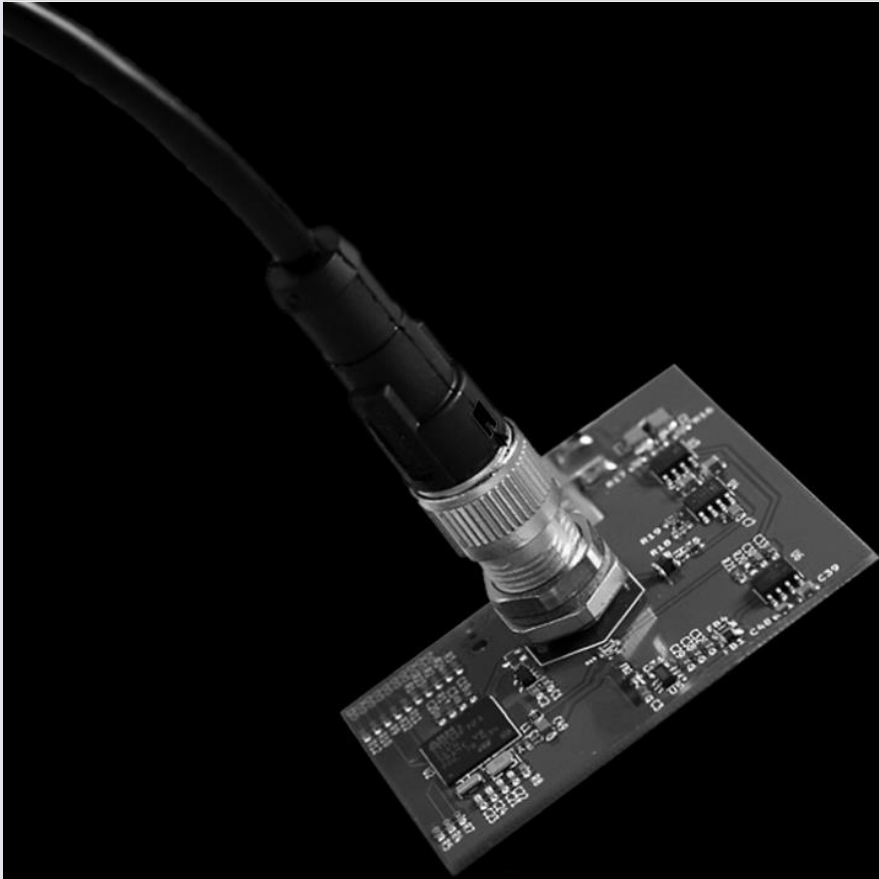
@odinthenerd





04.11.2016





**Application:** Predictive maintenance of plastic injection molding machines

**Delivery:** High-performance ultrasonic amplification circuit, digitization, Ethernet/USB driver and individual software





# std::variant

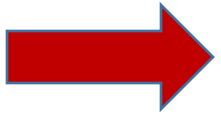


std::variant<  
MonitoringSystems,  
PhysicalSensors,  
PowerOverEthernet,  
TimeSeriesDataProcessing,  
ComeToEmBO>




`std::variant`






```
union U {  
    uint8_t i;  
    bool b;  
};  
  
int main( ) {  
    U u;  
    u.i = 10;  
    std::println("{} ", u.i);  
    return 0;  
}
```





```
union U {
    uint8_t i;
    bool b;
};

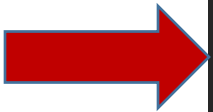
int main( ) {
    U u;
    u.i = 10;
    std::println("{} ", u.i);
    return 0;
}
```



```
union U {  
    uint8_t i;  
    bool b;  
};  
  
int main( ) {  
    U u;  
    u.b = false;  
    std::println("{} ", u.b);  
    return 0;  
}
```



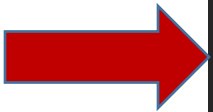
```
union U {  
    uint8_t i;  
    bool b;  
};  
  
int main( ) {  
    U u;  
    u.i = false;  
    std::println("{} ", u.b);  
    return 0;  
}
```







```
union U {  
    uint8_t i;  
    bool b;  
};  
  
int main( ) {  
    U u;  
    u.i = false;  
    std::println("{} ", u.b);  
    return 0;  
}
```



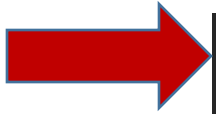


@odinthenerd






# Discriminated union



```
std::variant<int,std::string> v = "one million dollars";  
v = 1000000;  
if(std::holds_alternative<int>(v)){  
    int i = std::get<int>(v);  
}
```



# Discriminated union



```
std::variant<int,std::string> v = "one million dollars";  
v = 1000000;  
if(std::holds_alternative<int>(v)){  
    int i = std::get<int>(v);  
}
```



# That is $O(n)$ => spaghetti

```
std::variant<int,std::string> v = "one million dollars";  
v = 1000000;  
if(std::holds_alternative<int>(v)){  
    int i = std::get<int>(v);  
}
```



# That is $O(n) \Rightarrow$ spaghetti

```
std::variant<int,std::string /*...*/> v = "one million dollars";  
v = 1000000;  
if(std::holds_alternative<int>(v)){  
    int i = std::get<int>(v);  
}  
//...
```



# That is $O(n)$ => spaghetti

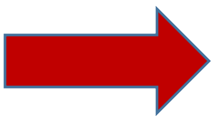
```
std::variant<int,std::string /*...*/> v = "one million dollars";  
v = 1000000;  
if(std::holds_alternative<int>(v)){  
    int i = std::get<int>(v);  
}  
//...
```





## How about in $O(1)$

```
std::variant<int,std::string> v = "one million dollars";  
v = 1000000;
```



```
std::visit(visitor{}, v);
```



# What is std::variant?

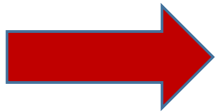
```
std::variant<int,std::string> v = "one million dollars";  
v = 1000000;
```



```
struct visitor{  
    void operator()(int i){  
        //int stuff  
    }  
    void operator()(std::string s){  
        //string stuff  
    }  
};  
std::visit(visitor{}, v);
```



I think about visit as a  
multiplexed/overloaded function call

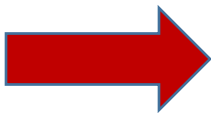


```
T t;  
f(t);  
  
//vs  
  
std::variant<Ts...> v;  
std::visit(fs,v);
```



I think about visit as a  
multiplexed/overloaded function call

```
T t;  
f(t);  
  
//vs  
std::variant<Ts...> v;  
std::visit(fs,v);
```





this seem more logical to me  
but we'll get back to that later

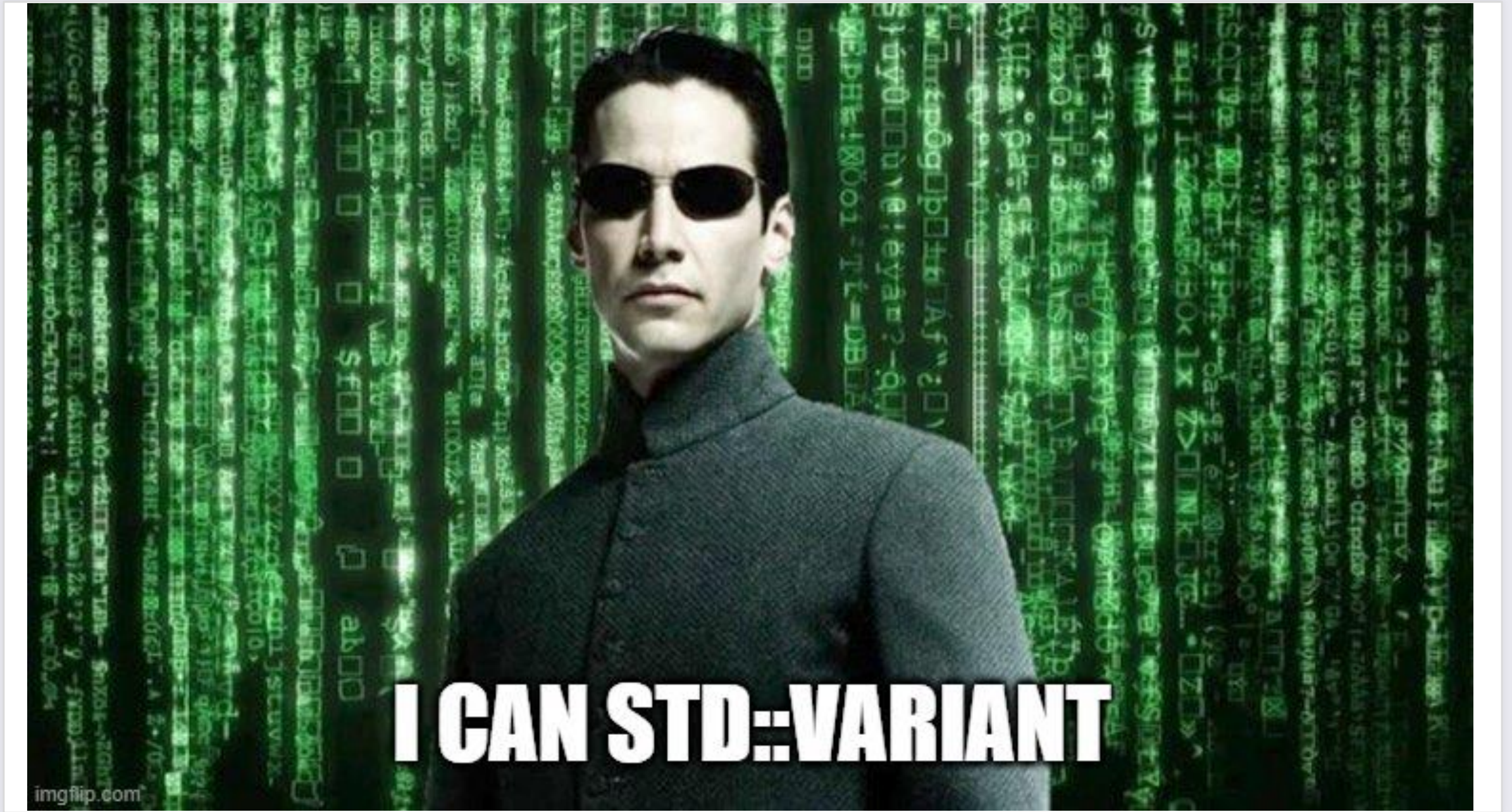
```
T t;  
f(t);  
  
//vs  
  
std::variant<Ts...> v;  
fs(v);
```







@odinthenerd





# What about evil?





# not evil yet

```
struct dr_evil{  
    dr_evil& operator=(const dr_evil& other){  
  
    }  
};
```



## now its evil

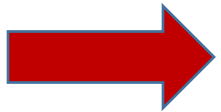


```
struct dr_evil{  
    dr_evil& operator=(const dr_evil& other){  
        throw diabolical_plot{};  
    }  
};
```



## what state is v in?

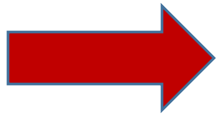
```
struct dr_evil{  
    dr_evil& operator=(const dr_evil& other){  
        throw diabolical_plot{};  
    }  
};  
  
v = dr_evil{};
```





## what should visit do?

```
struct dr_evil{  
    dr_evil& operator=(const dr_evil& other){  
        throw diabolical_plot{};  
    }  
};  
  
v = dr_evil{};  
//...  
std::visit(visitor{}, v);
```





## ~30 years of discussion

- `boost::variant`
- `boost::variant2`
- `QVariant`
- `std::variant`





## Takeaways:



## Takeaways:

- Dimov was right, boost::variant2 is cool



## Takeaways:

- Dimov was right, boost::variant2 is cool
- making things is hard



## Takeaways:

- Dimov was right, boost::variant2 is cool
- making things is hard
- I don't really care



# I've got other problems

```
template<typename T, typename U>
void do_generic_stuff(){
    struct visitor{
        void operator()(const T& t){

        }

        void operator()(const U& u){

        }
    };
    std::visit(visitor{}, f());
}
```



# Mo' legacy mo' problems

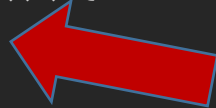
```
std::variant<OBSCURE_OS_ERROR_TYPE,int> f(){  
    if(stuff_works()){  
        return 43;  
    }  
    return OS_CODE_SOMETHING_IS_WRONG_IN_THE_STATE_OF_DENMARK;  
}
```





# Mo' legacy mo' problems

```
std::variant<OBSCURE_OS_ERROR_TYPE,int> f(){  
    if(stuff_works()){  
        return 43;  
    }  
    return OS_CODE_SOMETHING_IS_WRONG_IN_THE_STATE_OF_DENMARK;  
}
```







# Mo' legacy mo' problems

```
std::variant<OBSCURE_OS_ERROR_TYPE,int> f(){  
    if(stuff_works()){  
        return 43;  
    }  
    return OS_CODE_SOMETHING_IS_WRONG_IN_THE_STATE_OF_DENMARK;  
}
```





## What if T and U are the same type?

```
template<typename T, typename U>
void do_generic_stuff(){
    struct visitor{
        void operator()(const T& t){

        }
        void operator()(const U& u){

        }
    };
    std::visit(visitor{}, f());
}
```





# Less memory mo' problems

```
my_buffer<std::variant<bool, char, std::array<long, 10000>>> buff;  
buff.push(true);
```





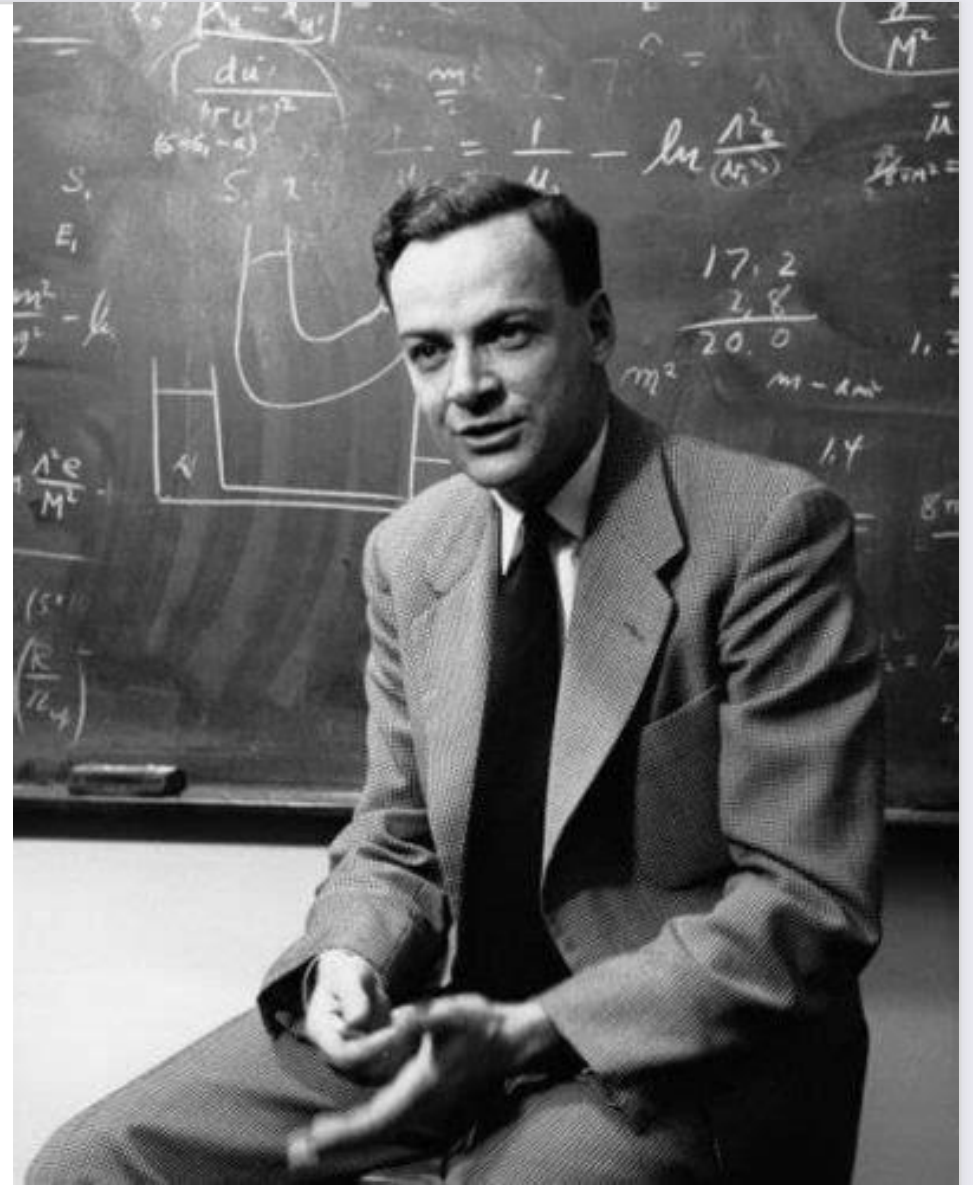
# Less memory mo' problems

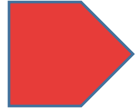
```
my_buffer<bool, char, std::array<long, 10000>> buff;  
buff.push<bool>(true);
```



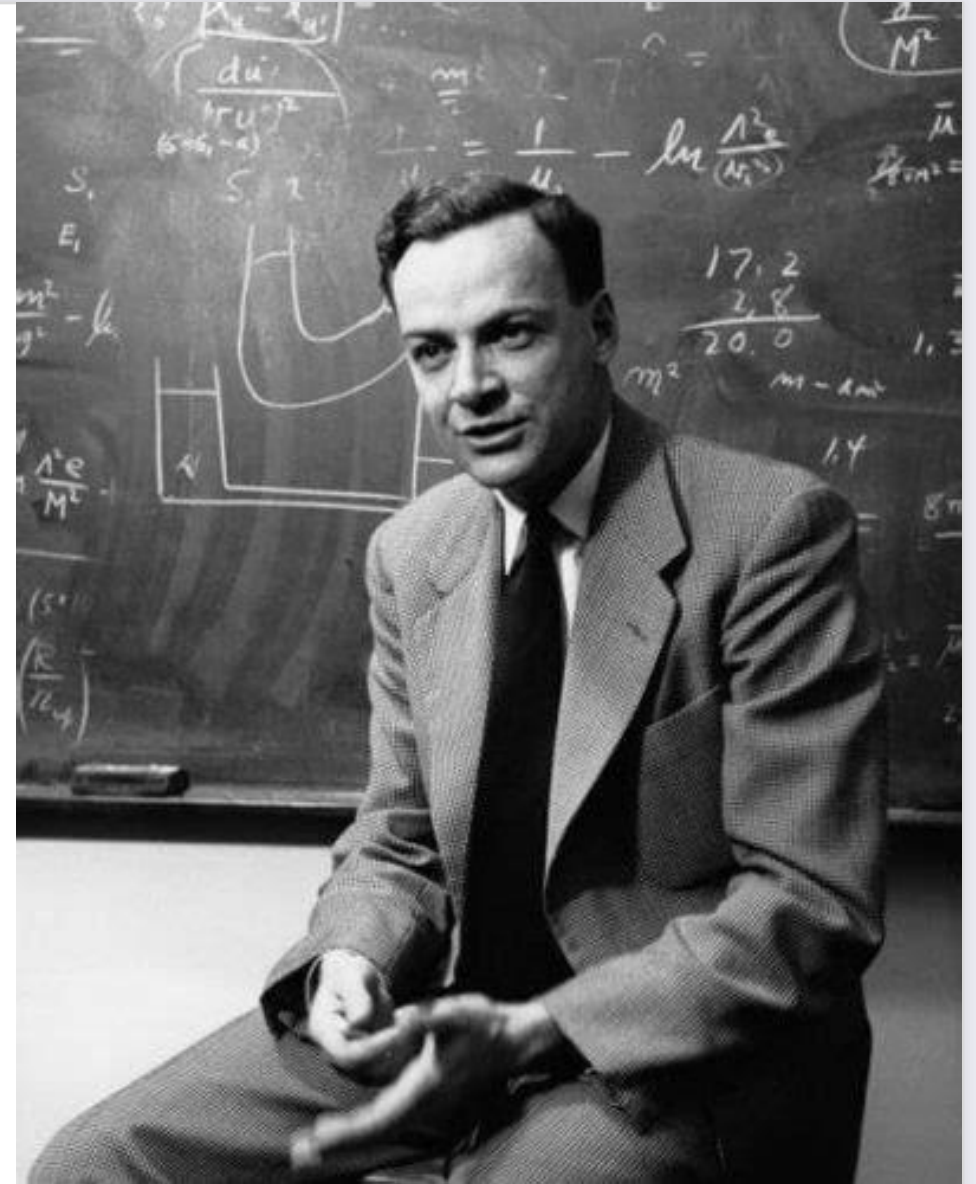
What kind  
of variant  
do I need?



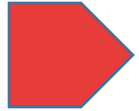




Write down the question.



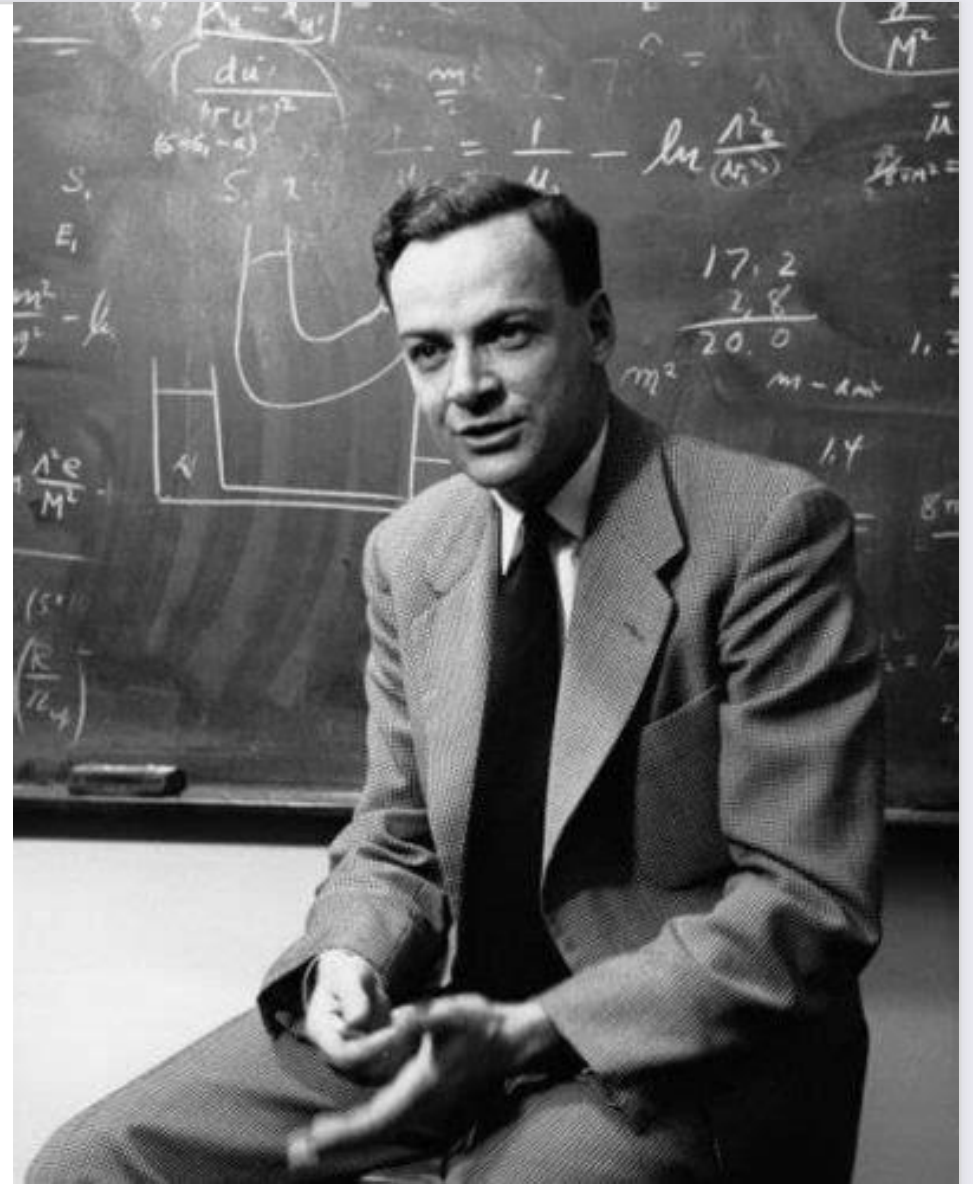


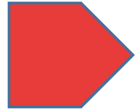


Write down the question.



Think real hard.





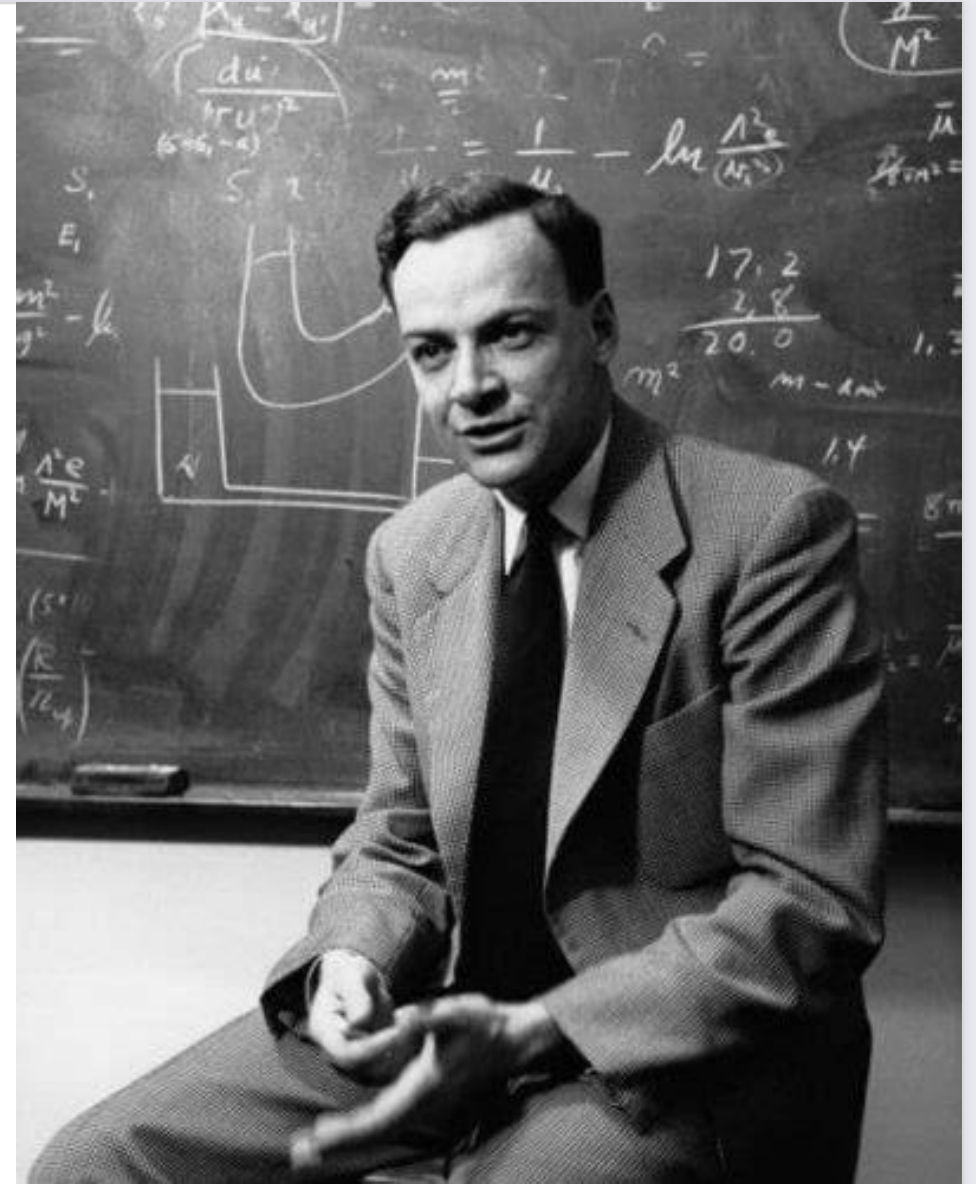
Write down the question.

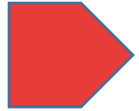


Think real hard.



Write down the answer.





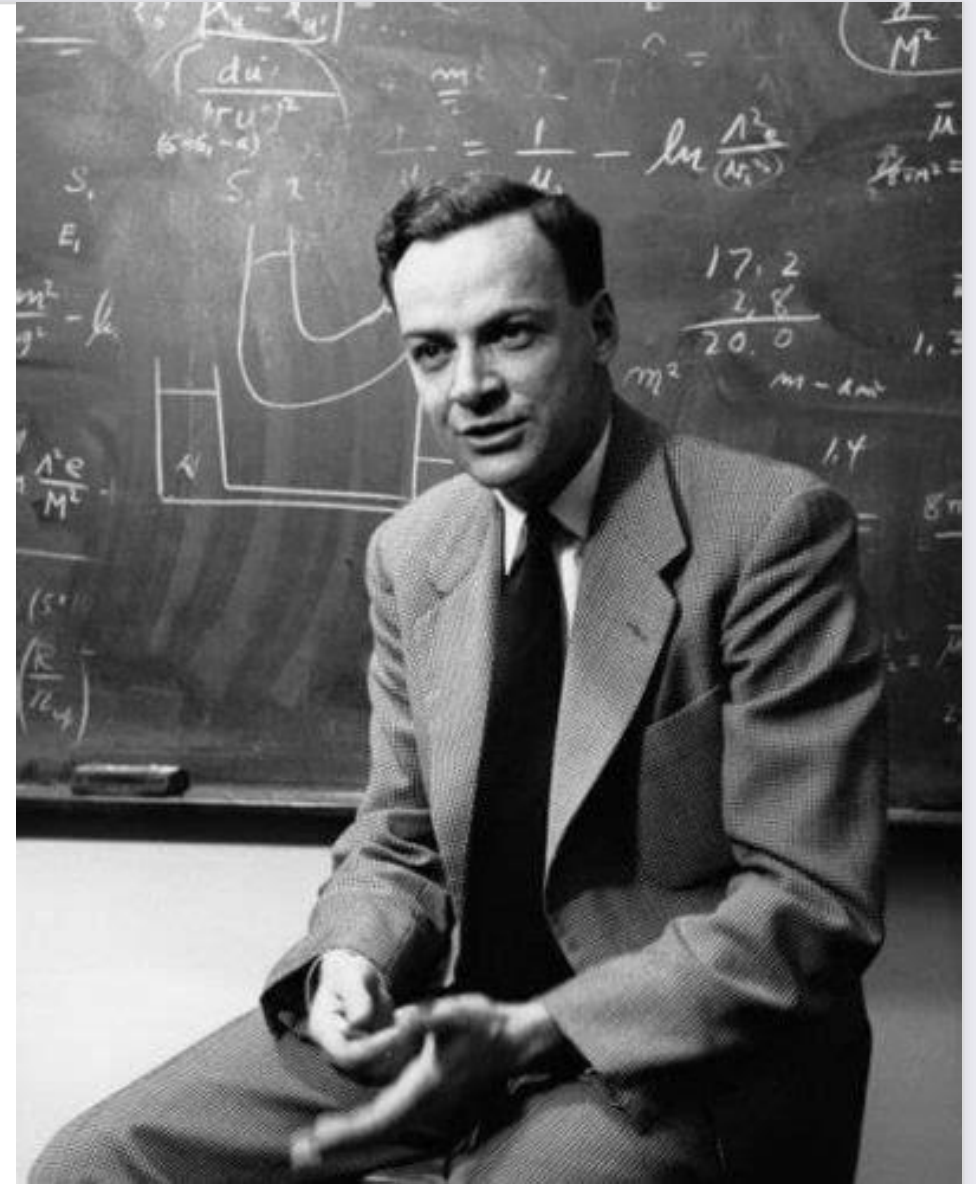
Write down the question.



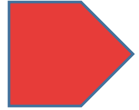
Think real hard.



Write down the answer.







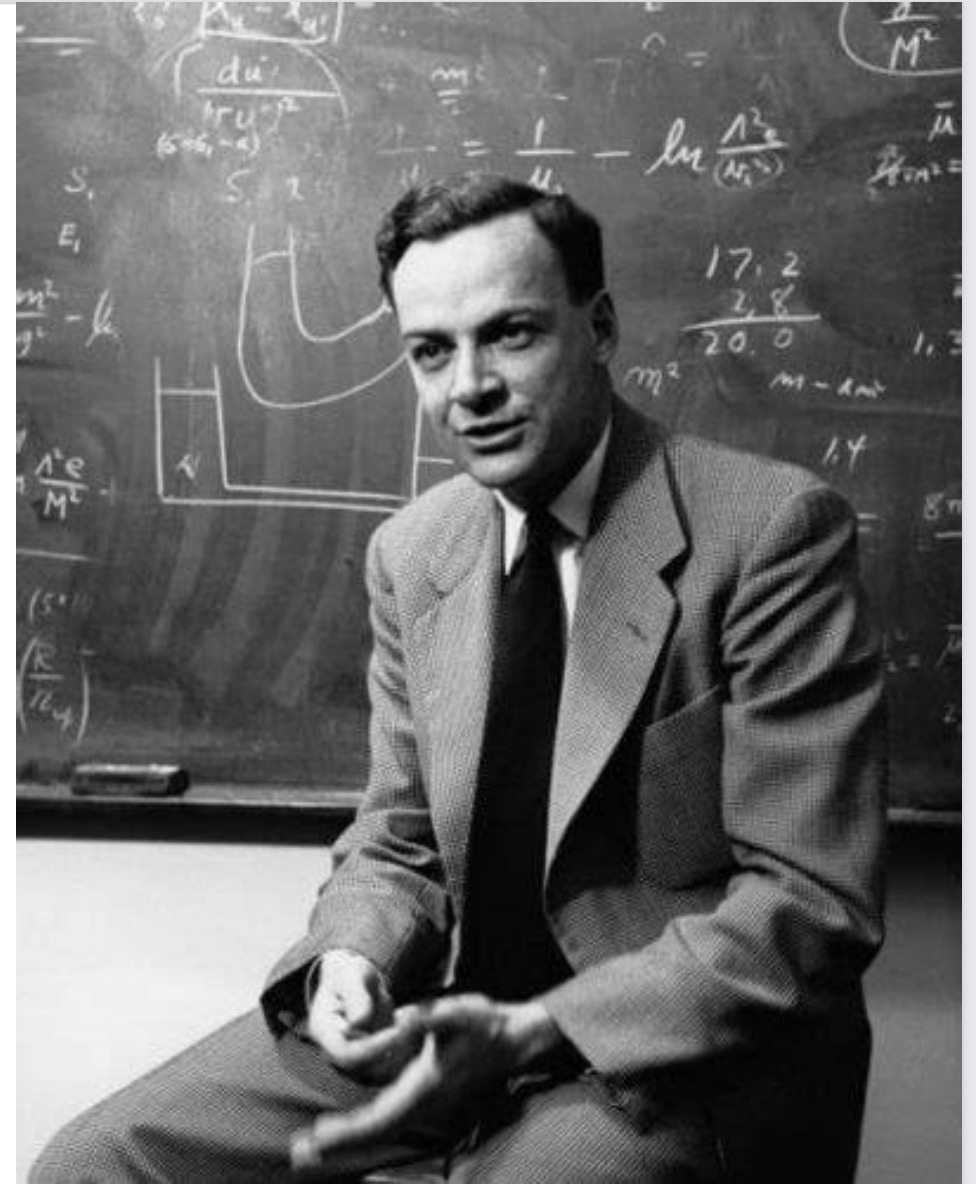
Write down the question.

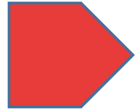


Think real hard.



Write down the answer.





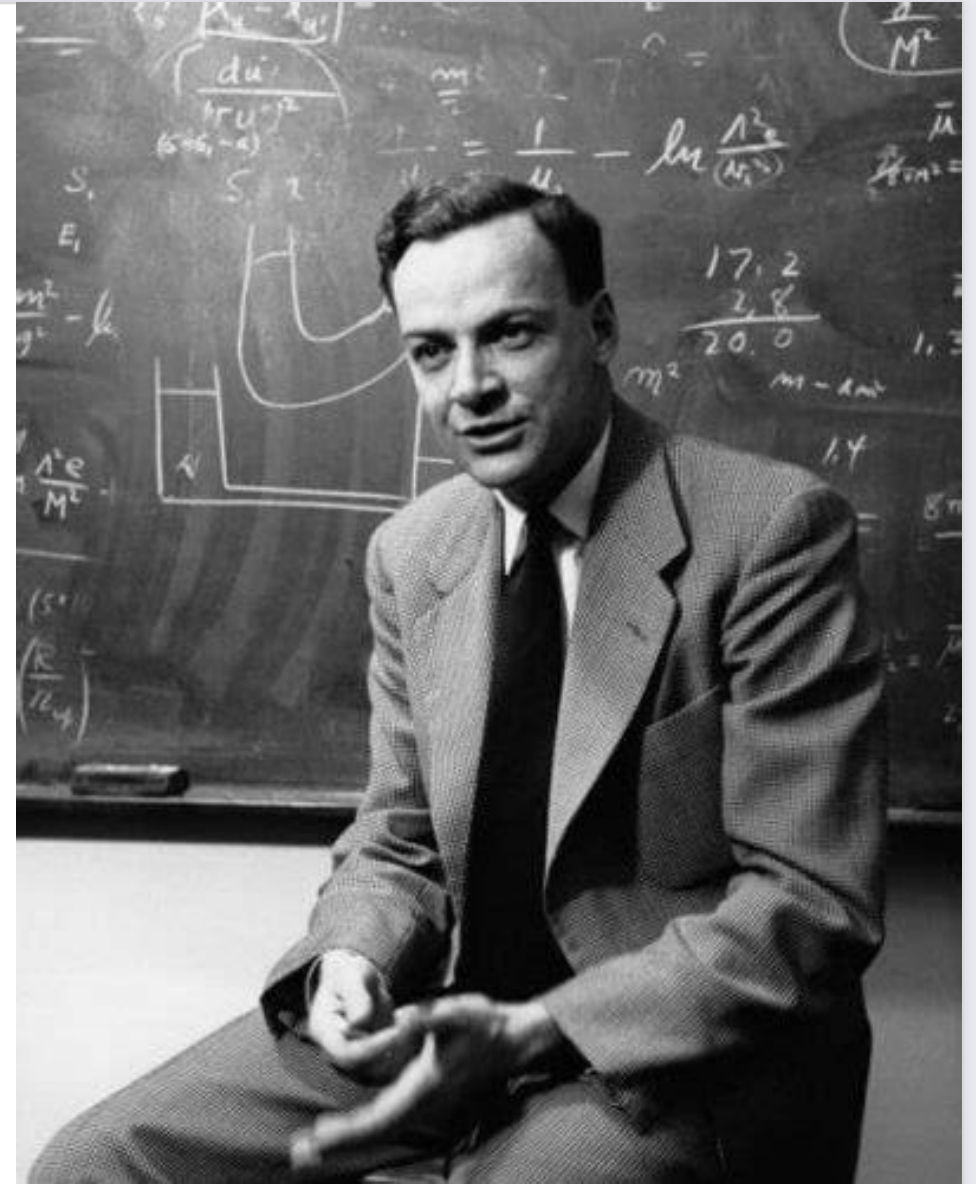
Write down the question.



Think real hard.



Write down the answer.

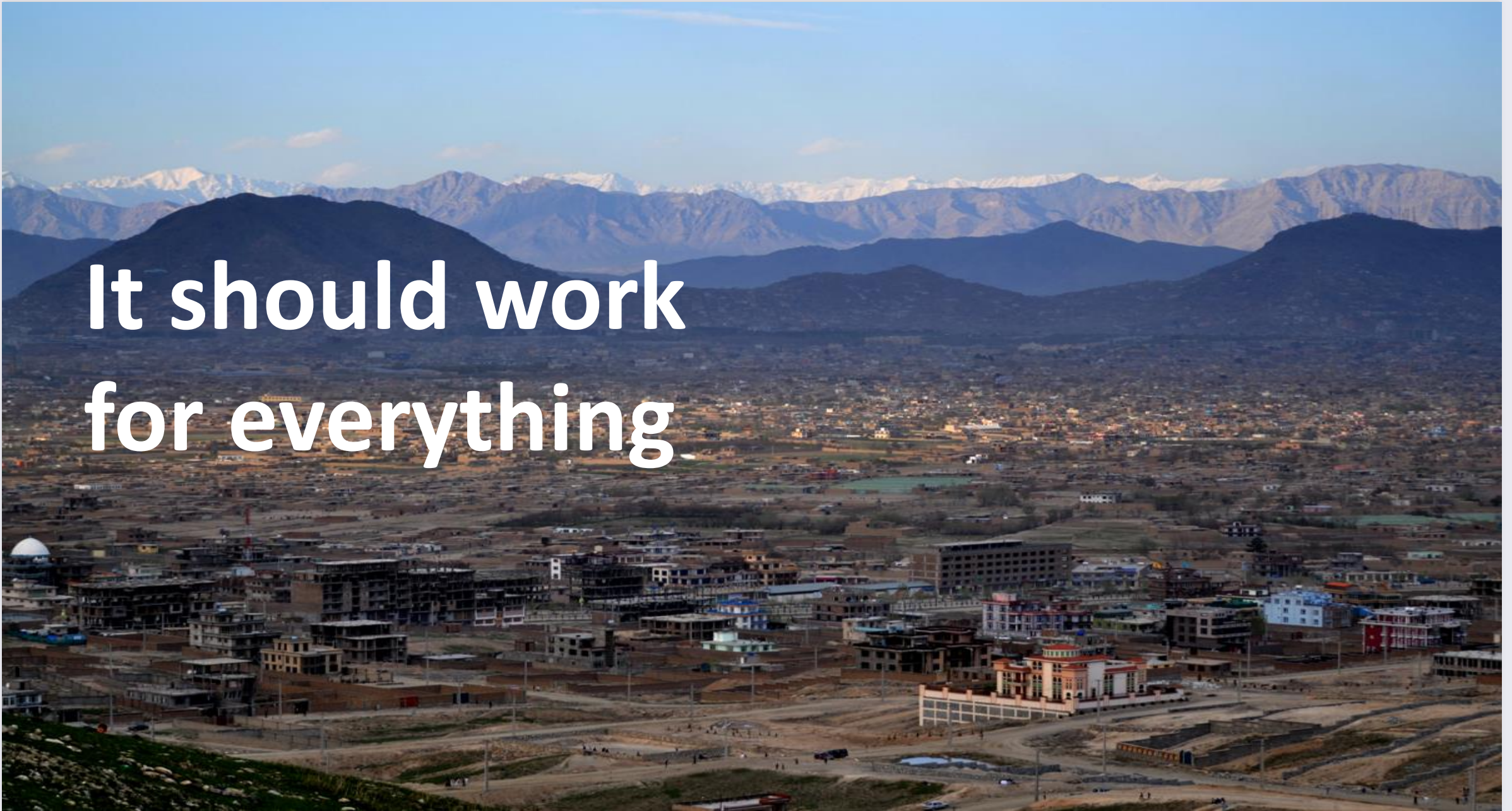






@odinthenerd

It should work  
for everything







**It should work  
for every  
sum type**

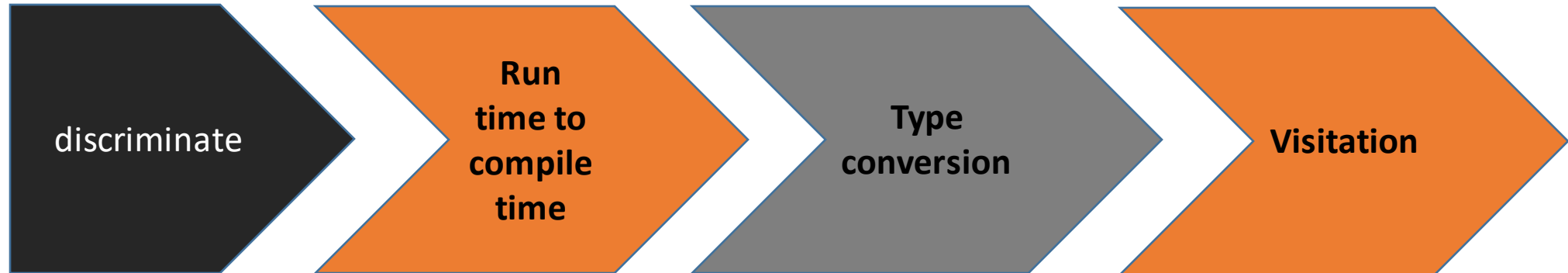


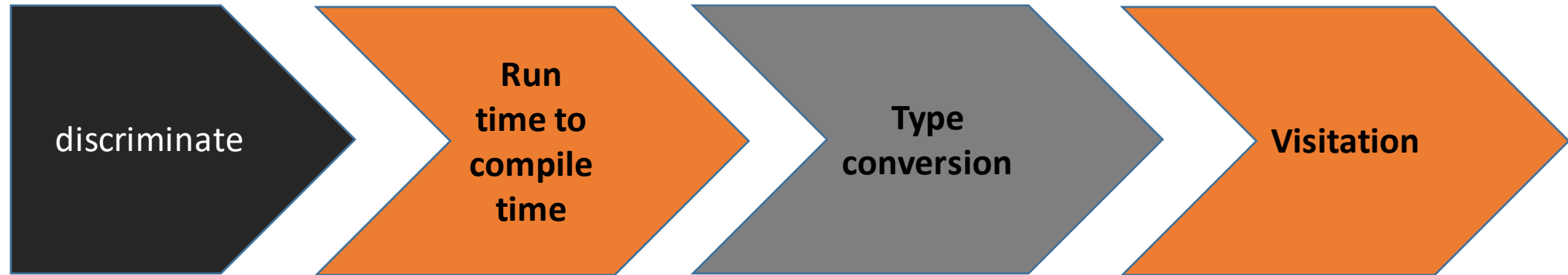


a variant is a type safe visitable closed  
sum type with fixed layout



a variant is a type safe **visitable** **closed**  
**sum type** with **fixed layout**





```
visitor(  
    converter(  
        discriminator(sum),  
        sum  
    )  
);
```

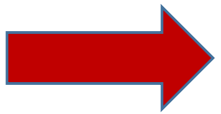




```
struct sum_type{  
    int type;  
    char data[1024]  
};
```



```
struct sum_type{  
    int type;  
    char data[1024]  
};  
  
template<typename T>  
int discriminator(const T& s){  
    return s.type;  
}
```



```
template<typename...Ts>
struct converter{
    template<std::size_t I, typename T>
    auto operator()(std::integral_constant<I>,T& t){
        return *static_cast<call_<at_<int_<I>>,Ts...>*>(
            static_cast<void*>(t.data)
        )
    }
}
```

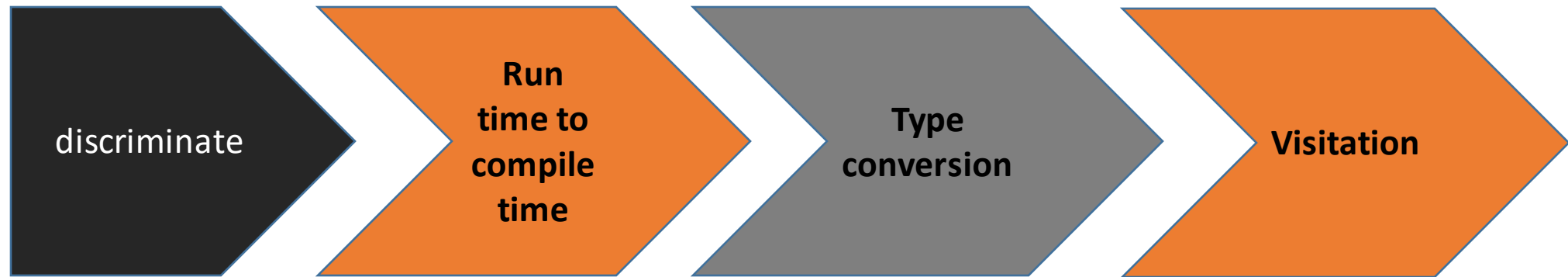


```
template<typename...Ts>
struct converter{
    template<std::size_t I, typename T>
    auto operator()(std::integral_constant<I>,T& t){
        return *static_cast<call_at<int<I>>,Ts...>*>(
            static_cast<void*>(t.data)
        )
    }
}
```

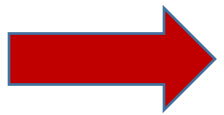
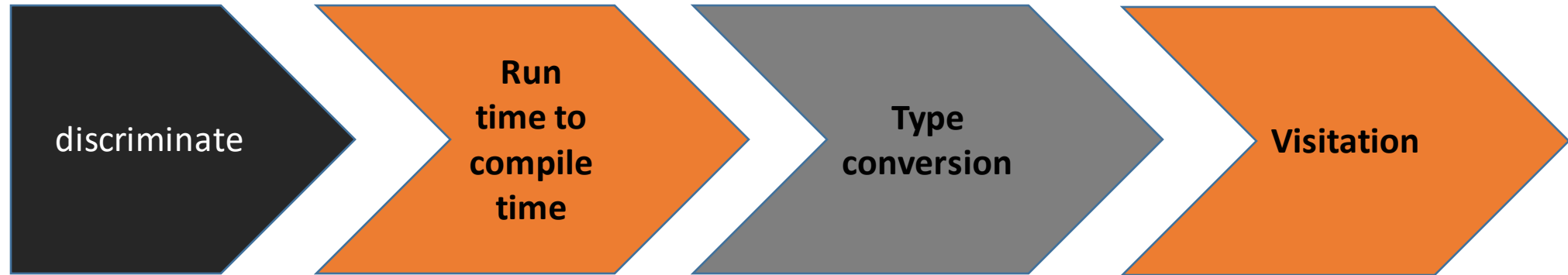


@odinthenerd

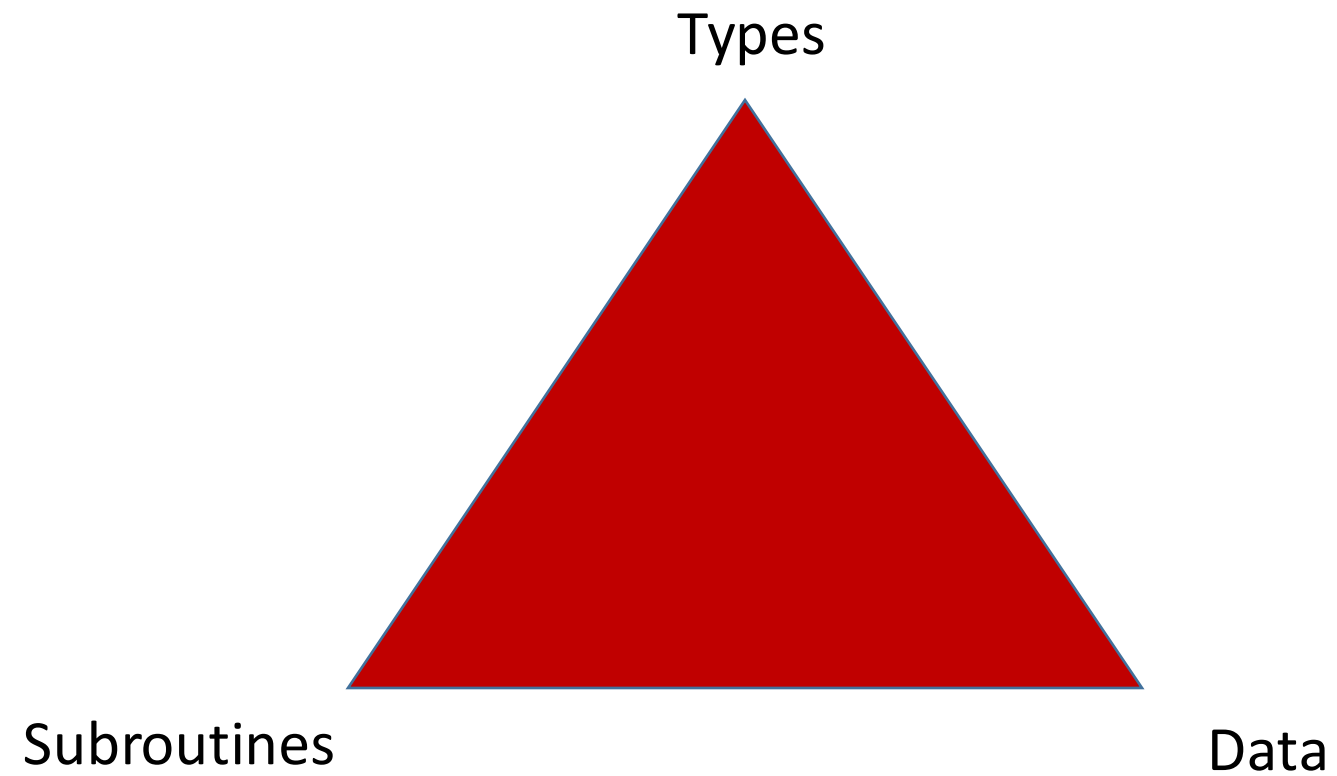




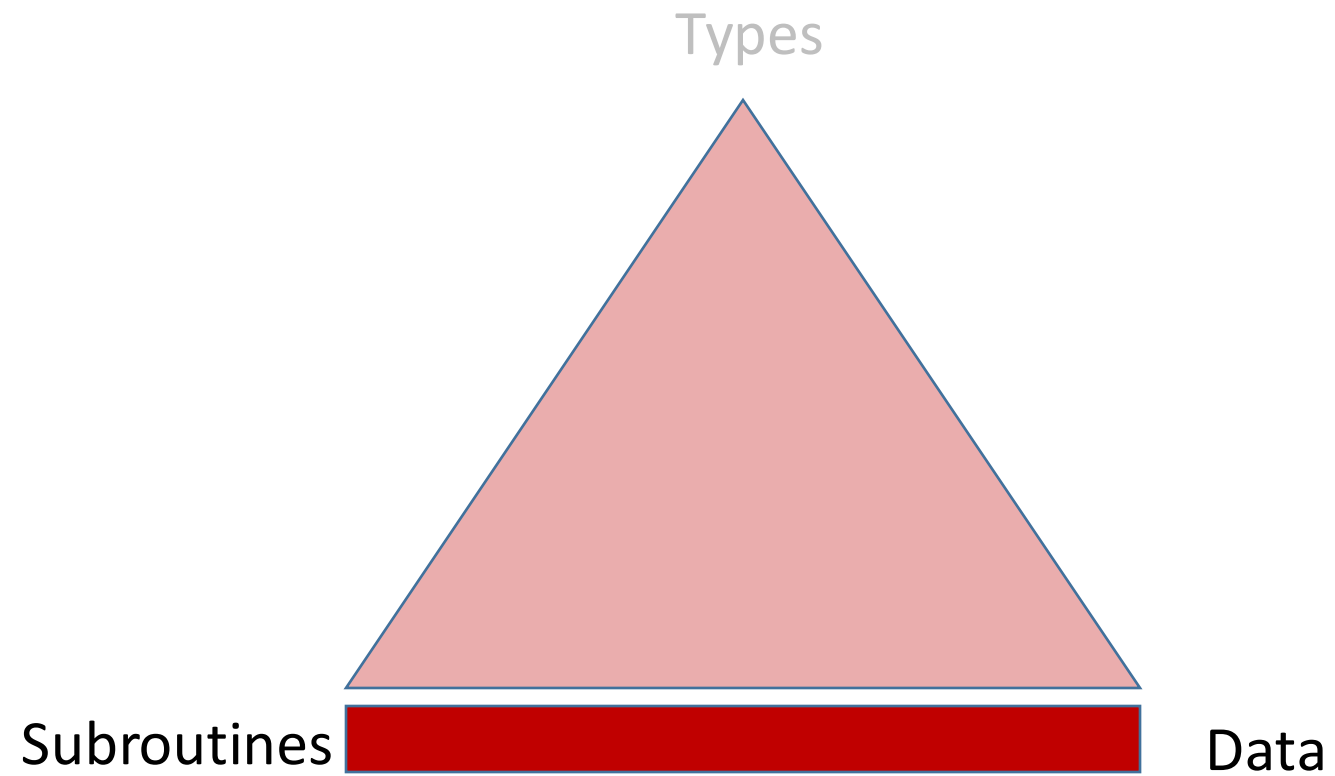
```
visitor(  
  converter(  
    discriminator(sum),  
    sum  
  )  
);
```



```
compose(converter,visitor)(  
    discriminator(sum),  
    sum  
);
```









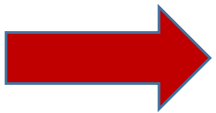
```
fun_ptr_type arr[]{  
    [](auto f, auto s){  
        return f(mp::int_<0>{}, s);  
    },  
    [](auto f, auto s){  
        return f(mp::int_<1>{}, s);  
    },  
    [](auto f, auto s){  
        return f(mp::int_<2>{}, s);  
    }  
};
```



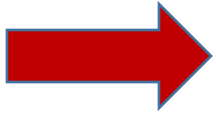
```
fun_ptr_type arr[]{  
    [](auto f, auto s){  
        return f(mp::int_<0>{}, s);  
    },  
    [](auto f, auto s){  
        return f(mp::int_<1>{}, s);  
    },  
    [](auto f, auto s){  
        return f(mp::int_<2>{}, s);  
    }  
};
```



```
fun_ptr_type arr[]{  
    [](auto f, auto s){  
        return f(mp::int_<0>{}, s);  
    },  
    [](auto f, auto s){  
        return f(mp::int_<1>{}, s);  
    },  
    [](auto f, auto s){  
        return f(mp::int_<2>{}, s);  
    }  
};
```



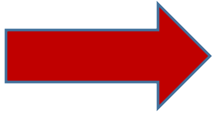
```
arr[idx](func, sum);
```




```
template<typename F, typename V, std::size_t...Is>
struct indexes<F,V,std::index_sequence<Is...>>{
    using ret_type = decltype(std::declval<F>()(mp::int_<0>{}),
                               std::declval<V>())); //<----
    static constexpr fun_ptr<ret_type,F,V> arr[]{
        [](F f, V v){
            return f(mp::int_<Is>{}, v);
        }...
    };
};
```



```
template<typename F, typename V, std::size_t...Is>
struct indexes<F,V,std::index_sequence<Is...>>{
    using ret_type = decltype(std::declval<F>()(mp::int_<0>{}),
                               std::declval<V>())); //<----
    static constexpr fun_ptr<ret_type,F,V> arr[]{
        [](F f, V v){
            return f(mp::int_<Is>{}, v);
        }...
    };
};
```



```
template<typename F, typename V, std::size_t...Is>
struct indexes<F,V,std::index_sequence<Is...>>{
    using ret_type = decltype(std::declval<F>()(mp::int_<0>{}),
                               std::declval<V>()));
    static constexpr fun_ptr<ret_type,F,V> arr[]{}           //<----
    [](F f, V v){
        return f(mp::int_<Is>{}, v);
    }...
};
};
```




```
template<typename F, typename V, std::size_t...Is>
struct indexes<F,V,std::index_sequence<Is...>>{
    using ret_type = decltype(std::declval<F>()(mp::int_<0>{}),
                               std::declval<V>()));
    static constexpr fun_ptr<ret_type,F,V> arr[]{
        [](F f, V v){
            return f(mp::int_<Is>{}, v);           //<----
        }...
    };
};
```






```
template<typename F, typename V, std::size_t...Is>
struct indexes<F,V,std::index_sequence<Is...>>{
    using ret_type = decltype(std::declval<F>()(mp::int_<0>{}),
                               std::declval<V>()));
    static constexpr fun_ptr<ret_type,F,V> arr[]{
        [](F f, V v){
            return f(mp::int_<Is>{}, v);
        }...
    };
};

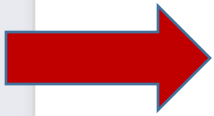
template<typename D, typename F>
constexpr auto make_demux_callable(D d, F f){
    return [=](auto v) constexpr{                                     //<----
        using idxs = std::make_index_sequence<decltype(d(v))::value>;
        auto fn_ptr = indexes<F, decltype(v), idxs>::arr[d(v)];
        return fn_ptr(f,v);
    };
};
```



```
template<typename F, typename V, std::size_t...Is>
struct indexes<F,V,std::index_sequence<Is...>>{
    using ret_type = decltype(std::declval<F>()(mp::int_<0>{}),
                               std::declval<V>()));
    static constexpr fun_ptr<ret_type,F,V> arr[]{
        [](F f, V v){
            return f(mp::int_<Is>{}, v);
        }...
    };
};

template<typename D, typename F>
constexpr auto make_demux_callable(D d, F f){
    return [=](auto v) constexpr{
        using idxs = std::make_index_sequence<decltype(d(v))::value>;
        auto fn_ptr = indexes<F, decltype(v), idxs>::arr[d(v)];
        return fn_ptr(f,v);
    };
};
```

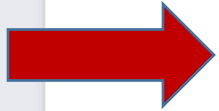




```
constexpr auto fun = make_demux_callable(  
    [](const auto& s)->of_n_alternatives<3>{ return s.type; },  
    [](auto idx, const auto& s){  
        using ty = call_at<decltype(idx)::value>, A, B, C>;  
        auto p = static_cast<ty>(static_cast<void*>(s.data));  
        do_something(*p);  
    }  
);
```



```
constexpr auto fun = make_demux_callable(  
    [](const auto& s)->of_n_alternatives<3>{ return s.type; },  
    [](auto idx,const auto& s){  
        using ty = call_<at_<decltype(idx)::value>,A,B,C>;  
        auto p = static_cast<ty>(static_cast<void*>(s.data));  
        do_something(*p);  
    }  
);  
  
fun(s);
```





```
std::tuple t{
    [](auto in){},
    [](auto in){},
    [](auto in){}
};

constexpr auto fun = make_demux_callable(
    [](const auto& s)->of_n_alternatives<3>{ return s.type; },
    [=](auto idx, const auto& s){
        using ty = call_<at_<decltype(idx)::value>, A, B, C>;
        auto p = static_cast<ty>(static_cast<void*>(s.data));
        std::get<decltype(idx)::value>(t)(*p);
    }
);
```



```
constexpr auto make_indexed_visitor = [] (auto t){
    return [=] (auto idx, const auto& s){
        constexpr int i = decltype(idx)::value;
        return std::get<i>(t)(s);
    };
};

constexpr auto make_converting_visitor = [] (auto typelist, auto f){
    return [=] (auto idx, const auto& s){
        using ty = type_<call_<unwrap_<at_<decltype(idx)::value>>, decltype(typelist)>>;
        return f(ty{}, s);
    };
};
```



```
constexpr auto ep_handlers = std::tuple{
    [](auto s){ return 1; },
    [](auto s){ return 2; },
    [](auto s){ return 2; },
    [](auto s){ return 2; },
    [](auto s){ return 2; },
    [](auto s){ return 2; },
    [](auto s){ return 2; },
    [](auto s){ return 3; }
};

char packet[128];
auto handle_usb_frame = make_demux_callable(
    get_ep,
    make_indexed_visitor(ep_handlers));
handle_usb_frame(packet);
```





# @odinthenerd

- Github.com
- Twitter.com
- Gmail.com
- Blogspot.com
- LinkedIn.com
- Embo.io