### Learning from the best.

Antony Polukhin, the author of Boost.TypeIndex and Boost.DLL libraries; maintainer of some Boos libraries and also the author of the "Boost C++ Application Development Cookbook" gave a very interesting [talk](https://www.youtube.com/watch?v=abdeAew3gmQ) last fall at CppCon2016 called:

"C++14 Reflections Without Macros, Markup nor External Tooling: Metaprogramming tricks for POD Types"

His very exciting, however admittedly brief explanation for the talk goes like this:

"C++ was lacking the reflections feature for a long time. But a new metaprogramming trick was discovered recently: we can get some information about POD structure by probing it's braced initializes. Combining that trick with variadic templates, constexpr functions, implicit conversion operators, SFINAE, decltype and integral constants we can count structure's fields and even deduce type of each field. Now the best part: everything works without any additional markup nor macros typically needed to implement reflections in C++. In this talk I'll explain most of the tricks in detail, starting from a very basic implementation that is only capable of detecting fields count and ending up with a fully functional prototype capable of dealing with nested PODs, const/volatile qualified pointers, pointers-to-pointers and enum members. Highly useful use-cases will be shown at the end of the talk."

He also spoke on the subject in front of Russian-speaking audience (video can be found [here](https://www.google.ru/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwiY1emb2JXXAhVKD5oKHQRxAXAQtwIILDAB&url=https%3A%2F%2Fwww.youtube.com%2Fwatch%3Fv%3DjDI5CHKFKd0&usg=AOvVaw2fonIwI0gSBvDsTl5MAES3)). The source code for the both talks could be found [here](https://github.com/apolukhin/magic_get)

Unfortunately, in those talks he omitted some technical details, and I don't know about you,   
but I personally still had troubles understanding the whole thing (apparently I was not alone in that judging by the comments left by a lot of folks ranging from excitement-frustration to condemnation-rude outrage), so I have to go deeper and dig into the code. Eventually, I think I got most of it, however some parts are still sort of mystery to me. However, I found my own solutions for those mysteries parts (will brag about it later ;) ).

So, basically, the technique is like that:

- using special template class (Anton originally called them “ubiq” after the main theme in Phillip K. Dick’s novel under the same name <https://en.wikipedia.org/wiki/Ubik#Interpretation>) with an implicit template conversion method for aggregate initialization for a given type you get field count for that (POD) type

- using that count you can get type info for that type using another kind of "ubiq"

- type info is presented as a numeric identifier and it can have additional info about being a pointer, volatile, const or all three at the same time (and many levels of them). And here comes the major limitation of the technique - you can convert only known types. But we’ll talk more about it later.

- you construct some sort of tuple (very similar to the one described in "Modern C++ Design" by Andrei Alexandrescu, in Chapter 3, the part about  class  GenScatterHierarchy) converting those numeric type ids into types

- using that resulting you can apparently safely use reinterpret\_cast (?!) to convert original into just constructed one and get data from it since you know the layout by now

You might want to read an alternative explanation provided [here](https://www.reddit.com/r/cpp/comments/6ydxya/the_great_type_loophole/)

If you still struggling with understanding of any part or its implementation, I suggest you go read Step by step section.  
Also, it may come as a surprise to you, but in some development environments they use not fully complaint C++14 compilers or even outdated (with no support for C++14) ones. For most of the projects I'm involved in at the moment this is that unfortunate state of affairs. So the code does not compile there. However, in his Q&A session at CppCon Antony did mentioned it could be done with C++11 by just adding more functional style to constexpr functions/methods, which I did not mind at all. However it turned out to be a lot more involved than that. So let’s go thru that step by step.

### Going step by step

#### Step 1 – finding field count

This step seemed fairly simple and straightforward when you see it on the video first. All you have to do it seems is to give a proper count to a SFINAE expression (see the code below) for *enable\_if and you done! Poof! Pure magic!* Not so fast, pal. You need carefully find starting with some reasonable estimate and try to avoid too many template instantiations (did you know what to many of them can lead to endless compile time or even kill the poor thing?). So here comes first couple good ones from Anton. He suggests that we start with the size of the given structure in bytes. In actual code he goes somewhat even more pessimistic (on behave of possible bit fields lurking somewhere he explains). Also he is using some kind of binary search to do it. While I took a slightly different approach about size I was completely in for the search thing. Experimenting with the size I thought it was too pessimistic. However at first even nothing worked in my development environment even when I tried unaltered code from flat\_magic repository. At some point I nearly gave up. But then I came with different solution which I consider slightly cleaner too. I combined all those constexpr functions in a class so scoping problem took care of itself (yes, it seemed like compilers on my site are not good at solving them when it comes to constexpr functions precedence and such. Take a note, it might save you a lot of time in some different case). Another good (and extremely important, though very subtle) thing which I creatively borrow from flat\_magic (meaning it was not just cut-n-paste, it had to be reworked to fit my needs) was method overloading (see the code below) using difference in type argument. It seems like something minor and you don’t see it at first (I didn’t!), but I found no other way even though I tried many things (I would be happy if somebody could point out another solution). Did you see that little difference in arguments (int vs. long)? This is it, this is how it works. Once we hit the correct count it will activate the correct method in *enable\_if* expression with accept\_any\_type class (an “ubiq”) in decltype expression (see the code below).  
So, finally we got that filed count!

WHAT I ADDED: Sizeof, more optimistic approach  
WHY IT WAS IMPORTANT: you’ll see

#### Step 2 – it was just a begging

Now we can get our hands dirty with some type info. Class called extract\_type\_info (yet another “ubiq”) does that job for each field *recursively.* Here I won’t go around biting the bush and come strait: I’m not sure about the code in flat\_magic: it does something very sophisticated iteratively, somehow getting same flat array of type ids with some of them representing ids for inner boundaries of types inside other types, which might look like that  
My version does that also aggregating some additional info about alignment (the reason for that will be explained later)

#### Encode type as a number (id)

Inside those iiteration/recursion over/in the type we encode types into type ids, because, Anton explained, it cannot be stored as is. His flat\_magic is employing some sort of technique   
type\_cypher (class encode\_cypher)

#### Step 3 – inflate structure from obtained flat array

In his flat\_magic Antony again employing some (painfully) elaborate iterative (???check that) procedures to get some sort of tuple (as already mentioned above very similar to the one described in "Modern C++ Design" by Andrei Alexandrescu) converting those numeric type ids into types  
since I did not really understand exactly the it was done I implemented my own way using structures called build\_type (found in build\_type.h) and functions from mimic\_type.h. Again we employ type\_cypher (class decode\_cypher in this case)