How many options fit into a boolean?

tl;dr: Exactly **254** options fit into a boolean.

```
size_of::<bool>() == 1;
size_of::<Option<bool>>() == 1;
size_of::<nest!(bool, 254)>() == 1;
size_of::<nest!(bool, 255)>() == 2;

// NonZeroU8 cannot be zero, so we use the
// zero value to denote 'None'.
size_of::<Option<u8>>() == 2;
size_of::<Option<NonZeroU8>>() == 1;

// no memory cost of options on references!
size_of::<&T>() == 8;
size_of::<Option<&T>>() == 8;
```

If you touch computers, you will most likely assume that a bool holds exactly two possible values (true and false), and that it takes up one byte of memory (we are ignoring the beautiful gift that is C++'s std::vector<bool> here).

In fact, looking at Rust:

```
==> assert_eq!(size_of::<bool>(), 1);
```

But what about size_of::<Option<bool>>()? For any T, Option<T> represents a value that may or may not exist. The type systems helps keep track of nullability, and you don't have to pass raw pointers everywhere. All of this extends to Rust's sum types in general. (Importantly, they are all **tagged** unions.)

We are using options since those are an easy example, and correspond to exactly one additional state of data. (Nested options happen 'by accident' when APIs interlock, but there is no *practical* reason to construct them. Either there is a value inside of them or not, that's equivalent to a normal option.)

It turns out that Option<bool> takes up exactly one byte of memory, the same as bool! The same is true for Option<Option<bool>>, all the way up to 254 nested options. At 255 nested options the compiler finally relents and requires a wastefully decadent two bytes to satisfy our sick desires.

This is known as the **niche optimization**..

```
// how to nest options hundreds of times?
// just commit recursive crimes with macros!
// you can just do things(tm)
// -> use number of commas to track depth
// __nest!(u8; ,,) == Option<__nest!(u8; ,)>
macro_rules! __nest {
    ($type:ty; , $($count:tt)*) => {
        Option<__nest!($type; $($count)*)>
    };
    ($type:ty;) => {
        $type
    };
}

// nest!(bool, 2) == Option<Option<bool>>
macro_rules! nest {
    ($t:ty, 0) => { __nest!($t; ) };
    ($t:ty, 1) => { __nest!($t; ,) };
    ($t:ty, 2) => { __nest!($t; ,,) };
    ($t:ty, 3) => { __nest!($t; ,,,) };
    // ...another few hundred lines of this
}
```

Taking a look at std::Vec, it turns out that we can nest it in over a thousand options without increasing its memory footprint!

```
size_of::<Vec<T>>() == 24;
size_of::<Option<Vec<T>>>() == 24;
size_of::<nest!(Vec<T>, 1024)>() == 24;
```

If we compare this with C++, we see that std::optional<std::vector<int>> requires a full 8 additional bytes over the base type.

How does this work? Rust's types do not follow the C-ABI (Not unless you add #[repr(C)] annotations.). In fact, the Rust compiler is allowed to reorder the fields of structs, stuff data into unreachable bit patterns, and more. This allows optimizations which C++ is not performing (by default).

A Vec has three fields: A pointer, a length, and a capacity. Length and capacity are assumed to be smaller than the largest pointer-sized signed integer on the platform. As a result, the highest bit of the capacity integer can be reused.

Consequently: (1) If the highest bit is not set, the value exists and the representation of the vector in memory is the 'standard' one. (2) If it is set, the other capacity bits are used to 'count', telling us which exact option is none. Pointer and length are uninitialized memory and not accessible.

Most importantly, **this also applies to structs containing a Vec.** If you have a struct that has a Vec, String, reference, bool, etc. in it, Rust will use the niche optimization to make any option containing your struct cheaper!

I would love to tell you that Rust always uses *all* available optimization space with all of its tagged unions (not just options). This is not the case.

In many cases it is: Result<Vec<i32>, u64> (either a Vec<i32> *or* a u64) has the same size as Vec<i32>.

However, Result<bool, bool> for whatever reason takes up two bytes of memory. This is, of course, deeply upsetting.