# Everything is a []u8

Sep 07, 2025

If you're coming to Zig from a more hand-holding language, one of the things worth exploring is the relationship between the compiler and memory. I think code is the best way to do that, but briefly put into words: the memory that your program uses is all just bytes; it is only the compile-time information (the type system) that gives meaning to and dictates how that memory is used and interpreted. This is meaningful in Zig and other similar languages because developers are allowed to override how the compiler interprets those bytes.

> This is something I've written about before; longtime readers might find this post repetitive.

Consider this code:

```zig
const std = @import("std");
pub fn main() !void {
  std.debug.print("{d}\n", .{@sizeOf(User)});
}

const User = struct {
  id: u32,
  name: []const u8,
};
```

It *should* print 24. The point of this post isn't *why* it prints 24. What's important here is that when we create a `User` - whether it's on the stack or the heap - it is represented by 24 bytes of memory.

If you examine those 24 bytes, there's nothing "User" about them. The memory isn't self-describing - that would be inefficient. Rather, it's the compiler itself that maintains meta data about memory. Very naively, we could imagine that the compiler maintains a lookup where the key is the variable name and the value is the memory address (our 24 bytes) + the type (`User`).

The fun, and sometimes useful thing about this is that we can alter the compiler's meta data. Here's an working but impractical example:

```zig
const std = @import("std");
pub fn main() !void {
  var user = User{.id = 9001, .name = "Goku"};
  const tea: *Tea = @ptrCast(&user);
  std.debug.print("{any}\n", .{tea});
}

const User = struct {
  id: u32,
  name: []const u8,
};

const Tea = struct {
  price: u32,
  type: TeaType,

  const TeaType = enum {
    black,
    white,
    green,
    herbal,
  };
};
```

First we create a `User` - nothing unusual about that. Next we use [@ptrCast]() to tell the compiler to treat the memory referenced by `user` as a `*Tea`. `@ptrCast` works on addresses, which is why we give it address of (`&`) `user` and get back a pointer (`*`) to `Tea`. Here the return type of `@ptrCast` is inferred by the type it's being assigned to.

You might have some questions like what does it print? Or, is it safe? And, is this ever useful?

We'll dig more into the safety of this in a bit. But briefly, the main concern is about the size of our structures. If `@sizeOf(User)` is 24 bytes, we'll be able to re-interpret that memory as anything which is 24 bytes or less. The `@sizeOf(Tea)` is 8 bytes, so this is safe.

I get different results on each run:

```
.{ .price = 39897726, .type = .white }
.{ .price = 75123326, .type = .white }
.{ .price = 6441598, .type = .white }
.{ .price = 77826686, .type = .white }
.{ .price = 4950654, .type = .white }
.{ .price = 69438078, .type = .white }
.{ .price = 78498430, .type = .white }
.{ .price = 79022718, .type = .white }
```

It's possible (but not likely) you get consistent result. I find these results surprising. If had to imagine what the 24 bytes of `user` looks like, I'd come up with:

```
41, 35, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, x, x, x, x, x, x, x, x
```

Why that? Well, I'd expect the first 8 bytes to be the id, 9001, which has a byte representation of `41, 35, 0, 0, 0, 0, 0, 0`. The next 8 bytes I think would be the string length, or `4, 0, 0, 0, 0, 0, 0, 0` The last 8 bytes would be the pointer to actual string value - an address that I have no way of guessing, so I mark it with `x, x, x, x, x, x, x, x`.

> If you think the `id` should only take 4 bytes, given that it's a u32, good! But Zig will usually align struct fields, so it really will take 8 bytes. That isn't something we'll dive into this post though.

Since `Tea` is only 8 bytes and since the first 8 bytes of `user` are always the same (only the pointer to the name value changes from instance to instance and from run to run), shouldn't we always get the same `Tea` value?

Yes, but only if I'm correct about the contents of those 24 bytes for `user`. Unless we tell it otherwise, Zig makes no guarantees about how it lays out the fields of a struct. The fact that our `tea` keeps changing, makes me believe that, for reasons I don't know, Zig decided to put the pointer to our name at the start.

The reason you might get different results is that Zig might have organized the user's memory different based on your platform or version of Zig (or any other factor, but those are the two more realistic reasons).

So while this code might never crash, doesn't the lack of guarantee make it useless? No. At least not in three cases.

### Well-Defined In-Memory Layout

While Zig usually doesn't make guarantees about how data will be organized, C programs **do** . In Zig, a structure declared as `extern` follows that specification. We can similarly declare a structure as `packed` which also has a well-defined memory layout (but just not necessarily the same as C's / `extern`).

`extern` and `packed` structs can only contain `extern` and `packed` fields. In order for a struct to have a well-known memory layout, all of its field must have a well-known memory layout. They can't, for example, have slices - which don't have a guaranteed layout. Still, here's a reliable and realistic example:

```zig
const std = @import("std");
pub fn main() !void {
  var manager = Manager{.id = 4, .name = "Leto", .name_len = 4, .level = 99};
  const user: *User = @ptrCast(&manager);
  std.debug.print("{d}: {s}\n", .{user.id, user.name[0..user.name_len]});
}

const User = extern struct {
  id: u32,
  name: [*c]const u8,
  name_len: usize,
};

const Manager = extern struct {
  id: u32,
  name: [*c]const u8,
  name_len: usize,
  level: u16,
};
```

Part of the guarantee is that the fields are laid out in the order that they're declared. Above, when I guessed at the layout of `user`, I made that assumption - but it's only valid for `extern` structs. We can be sure that the above code will print `4: Leto` because `Manager` has the same fields as `User` and in the same order. We can, and should, make this more explicit:

```zig
const Manager = extern struct {
  user: User,
  level: u16,
};
```

Because the type information is only meta data of the compiler, both declarations of `Manager` are the same - they're the same size and have the same layout. There's no overhead to embedding the `User` into `Manager` this way.

This type of memory-reinterpretation can be found in some C code and thus see in any Zig code that interacts with such a C codebase.

## Leveraging Zig Builtins

While we can't assume anything about the memory layout of non-extern (or packed) struct, we can leverage various built-in functions to programmatically figure things out, such as `@sizeOf`.

Probably the most useful is `@offsetOf` which gives us the offset of a field in bytes.

```zig
const std = @import("std");
pub fn main() !void {
  std.debug.print("name offset: {d}\n", .{@offsetOf(User, "name")});
  std.debug.print("id offset: {d}\n", .{@offsetOf(User, "id")});
}

const User = struct {
  id: u32,
  name: []const u8,
};
```

For me, this prints:

```
name offset: 0
id offset: 16
```

This helps confirm that Zig did, in fact, put the `name` before the `id`. We saw the result of that when we treated the user's memory as an instance of `Tea`. If we wanted to create a `Tea` based on the address of `user.id` rather than `user`, we could do:

```zig
const std = @import("std");
pub fn main() !void {
  var user = User{.id = 9001, .name = "Goku"};

  // changed from &user to &user.id
  const tea: *Tea = @ptrCast(&user.id);
  std.debug.print("{any}\n", .{tea});
}
```

This will now always output the same result. But how would we take `tea` and get a `user` out of it? Generally speaking, this wouldn't be safe since `@sizeOf(Tea) < @sizeOf(User)` - the memory created to hold an instance of `Tea`, 8 bytes, can't represent the 24 bytes need for `User`. But for this instance of `Tea`, we know that there are 24 bytes available "around" `tea`. Where exactly those 24 bytes start depends on the relative position of `user.id` to `user` itself. If we don't adjust for that offset, we risk crashing unless the offset happens to be 0. Since we know the offset is 16, not 0, this should crash:

```zig
const std = @import("std");
```

```
pub fn main() !void {
  var user = User{.id = 9001, .name = "Goku"};
  var tea: *Tea = @ptrCast(&user.id);

  const user2: *User = @ptrCast(&tea);
  std.debug.print("{any}\n", .{user2});
}
```

This is our `user`'s memory (as 24 contiguous bytes of memory, broken up by the 3 8-byte fields):

```
name.ptr => x, x, x, x, x, x, x, x
name.len => 4, 0, 0, 0, 0, 0, 0, 0,
name.id  => 41, 35, 0, 0, 0, 0, 0, 0
```

And when we make `tea` from `&name.id`:

```
       name.ptr => x, x, x, x, x, x, x, x
       name.len => 4, 0, 0, 0, 0, 0, 0, 0,
tea => name.id  => 41, 35, 0, 0, 0, 0, 0, 0
                   more memory, but not ours to play with
```

If we try to cast `tea` back into a `*User`, we'll be 16 bytes off, and end up reading 16 bytes of memory adjacent to `tea` which isn't ours.

To make this work, we need to take the address of `tea` and subtract the `@offset(User, "id")` from it:

```
const std = @import("std");
pub fn main() !void {
  var user = User{.id = 9001, .name = "Goku"};
  const tea: *Tea = @ptrCast(&user.id);
  const user2: *User = @ptrFromInt(@intFromPtr(tea) - @offsetOf(User, "id"));
  std.debug.print("{any}\n", .{user2});
}
```

Because we use `@offsetOf`, it no longer matters how the structure is laid out. We're always able to find the starting address of `user` based on the address of `user.id` (which is where `tea` points to) because we know `@offsetOf(User, "id")`.

## As Raw Memory

The above example is convoluted. There's no relationship between the data of a `User` and of `Tea`. What does it mean to create `Tea` out of a user's `id`? Nothing.

What if we forget about `user`'s data, the `id` and `name`, and treat those 24 bytes as usable space?

```
const std = @import("std");
pub fn main() !void {
  var user = User{.id = 9001, .name = "Goku"};

  const tea: *Tea = @ptrCast(&user);
  tea.* = .{.price = 2492, .type = .black};

  std.debug.print("{any}\n", .{tea});
}
```

`user` and `tea` still share the same memory. We cannot safely use `user` after writing to `tea.*` - that write might have stored data that cannot safely be interpreted as a `User`. Specifically in this case, the write to tea has probably made `name.ptr` point to invalid memory. But if we're done with `user` and know it won't be used again, we just saved a few bytes of memory by re-using its space.

This can go on forever. We can safely re-use the space to create another `User`, as long as we're 100% sure that we're done with `tea:`:

```
pub fn main() !void {
  var user = User{.id = 9001, .name = "Goku"};

  const tea: *Tea = @ptrCast(&user);
  tea.* = .{.price = 2492, .type = .black};
  std.debug.print("{any}\n", .{tea});

  const user2: *User = @ptrCast(@alignCast(tea));
  user2.* = .{.id = 32, .name = "Son Goku"};
  std.debug.print("{any}\n", .{user2});
}
```

We can re-use those 24 bytes to represent anything that takes 24 bytes of memory or less.

The best practical example of this is `std.heap.MemoryPool(T)`. The `MemoryPool` is an allocator that can create a single type, `T`. That might not sound particularly useful, but using what we've learned so far, it can efficiently at re-use memory of discarded values.

We'll build a simplified version to see how it works, starting with a basic API - one without any recycling ability. Further, rather than make it generic, we'll make a `UserPool` specific for `User`:

```
pub const UserPool = struct {
  allocator: Allocator,

  pub fn init(allocator: Allocator) UserPool {
    return .{
      .allocator = allocator,
    };
  }

  pub fn create(self: *UserPool) !*User {
    return self.allocator.create(User);
  }

  pub fn destroy(self; *UserPool, user: *User) void {
    self.allocator.destroy(user);
  }
};
```

As-is, this is just a wrapper that limits what the allocator is able to create. Not particularly useful. But what if instead of destroying a `user` we made it available to subsequent `create`? One way to do that would be to hold an `std.SinglyLinkedList`. But for that to work, we'd need to make additional allocations - the linked list node has to exist somewhere. But why? The `@sizeOf(User)` is large enough to be used as-is, and whenever a `user` is destroyed, we're being told that memory is free to be used. If an application *did* use a `user` after destroying it, it would be undefined behavior, just like it is with any other allocator. Let's add a bit of decoration to our `UserPool`:

```
pub const UserPool = struct {
  allocator: Allocator,
  free_list: ?*FreeEntry = null,

  const FreeEntry = struct {
      next: ?*FreeEntry,
  };

  // rest is unchanged . . . for now.
};
```

We've added a linked list to our `UserPool`. Every `FreeEntry` points to another `*FreeEntry` or `null`, including the initial one referenced by `free_list`. Now we change `destroy`:

```zig
pub const UserPool = struct {
  // ...

  pub fn destroy(self: *UserPool, user: *User) void {
    const entry: *FreeEntry = @ptrCast(user);
    entry.* = .{ .next = self.free_list };
    self.free_list = entry;
  }
};
```

We use the ideas we've explored above to create a simple linked list. All that's left is to change `create` to leverage it:

```zig
pub const UserPool = struct {
  // ...

  pub fn create(self: *UserPool) !*User {
    if (self.free_list) |entry| {
      self.free_list = entry.next;
      return @ptrCast(entry);
    }
    return self.allocator.create(User);
  }
};
```

If we have a `FreeEntry`, then we can turn that into a `*User`. We make sure to advanced our `free_list` to the next entry, which might be `null`. If there isn't an available `FreeEntry`, we allocate a new one.

As a final step, we should add a `deinit` to free the memory held by our `free_list`:

```zig
pub const UserPool = struct {
  // ...

  pub fn deinit(self: *UserPool) void {
    var entry = self.free_list;
    while (entry) |e| {
      entry = e.next;
      const user: *User = @ptrCast(e);
      self.allocator.destroy(user);
    }
    self.free_list = null;
  }
};
```

That final `@ptrCast` from a `*FreeEntry` to a `*User` might seem unnecessary. If we're freeing the memory, why does the type matter? But allocators only know how much memory to free because the compiler tells them - based on the type. Freeing `e`, a `*FreeEntry` would only work if `@sizeOf(FreeEntry) == @sizeOf(User)` (which it isn't).

In addition to being generic, Zig's actual `MemoryPool` is a bit more sophisticated, handling different alignments and even handling the case where `@sizeOf(T) < @sizeOf(FreeEntry)`, but our `UserPool` is pretty close.

## Conclusion

By altering the compiler's view of our program, we can do all types of things and get into all types of trouble. While these manipulations can be done safely, they rely on understanding the lack of guarantees Zig makes. If you're programming in Zig, this is the type of thing you should try to get comfortable with. Most of this is fundamental regardless of the programming language, it's just that some languages, like Zig, give you more control.

I had initially planned on writing a version of `MemoryPool` which expanded on the standard library's. I wanted to create a pool for multiple types. For example, one that can be used for both `User` and `Tea` instances The trick, of course, would be to always allocate memory for the largest supported type (`User` in this case). But this post is already long, so I leave it as an exercise for you.