Friday, June 9, 2023

Problems of C, and how Zig addresses them

Aryan Ebrahimpour
@avestura

C is a low-level systems programming language with almost no abstraction over memory (so the memory management is all yours) and with minimal abstractions over assembly (yet expressive enough to support some general concepts like having a type system). It is also a very portable programming language, so when written correctly, it can run on your kitchen toaster even if it has some obscure architecture.

The characteristics of C make it a highly suitable language for its intended purposes. However, this does not imply that its design decisions are flawless by today's standards. In this blog post, we will discuss certain issues that have led to multiple attempts at creating alternative languages aimed at replacing C.

The Zig programming language has garnered considerable attention as a new systems programming language, positioning itself as *the better C*. But how does Zig achieve this? In this blog post, our aim is to examine some of the issues associated with C and explore how Zig intends to address them.

Table of Differences

- C has a more complex syntax to deal with

`Comptime` over Textual Replacement Preprocessing

Replacing the text in the source code using preprocessors is not exclusive to C. It existed way before the creation of C and can be traced back to earlier examples such as the SAP assembler for IBM 704 computers. Below is an example of an AMD64 assembly snippet that defines a `pushr` macro and substitutes it with either `push` or `pushf` based on its argument:

amd64-macro.asm

```
%macro pushr 1
%ifidn %1, rflags
pushf
%else
push %1
%endif
%endmacro

%define regname rcx

pushr rax
pushr rflags
pushr regname
```

C being a minimal abstraction over assembly adopted the same method to support macros, which can easily turn into a foot gun. As a small example:

footgun-macro.c

```
#define SQUARE(x) x * x

int result = SQUARE(2 + 3)
```

One might expect this code to set the value of the `result` to `square of (2 + 3) = (2 + 3)^2 = 25`. However, due to the textual replacement nature of the `SQUARE` macro function, the expansion results in `2 + 3 * 2 + 3`, which evaluates to 11, not 25.

To make this work correctly, it is crucial to ensure that all our macros are properly parenthesized:

footgun-macro.c

```c
#define SQUARE(x) ((x)*(x))
```

C will not tolerate such errors, nor will be kind enough to notify you about them. An error can show itself much later, on another input, in a completely irrelevant part of the program.

Zig, on the other hand, employs a much more intuitive approach for such tasks by introducing `comptime` parameters and functions. This enables us to execute functions during compile-time rather than runtime. Here's the same C `SQUARE` macro in Zig:

zig

```zig
fn square(x: anytype) @TypeOf(x) {
    return x * x;
}

const result = comptime square(2 + 3); // result = 25, at compile-time
```

Another advantage of the Zig compiler is its ability to perform type checking on inputs, even if it is `anytype`. When calling the `square` function in Zig, if a type is used that does not support the `*` operator, it would result in a compile-time type error:

zig

```zig
const result = comptime square("hello"); // compile time error: type mismatch
```

`Comptime` allows execution of any arbitrary code in compile time:

comptime-example.zig

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

fn fibonacci(index: u32) u32 {
    if (index < 2) return index;
    return fibonacci(index - 1) + fibonacci(index - 2);
}

pub fn main() void {
  const foo = comptime fibonacci(7);
  std.debug.print("{}", .{ foo });
}
```

This Zig program defines a `fibonacci` function and then calls the function at compile-time to set the value of `foo`. No `fibonacci` is called at runtime.

Zig's comptime evaluations can also cover some of the small C quirks: For instance, in a platform where the minimum `signed` value is -2^15=-32768 and the maximum value is (2^15)-1=32767, it is impossible to write the minimum value of the type `signed` as a literal constant in C.

c

```c
signed x = -32768; // not possible in C
```

That is because in C `-32768` is actually `-1 * 32768` and `32768` is not in the boundaries of the `signed` type. In Zig, however, `-1 * 32768` is a compile time evaluation.

zig

```zig
const x: i16 = -1 * 32768; // Valid in Zig
```

Memory Management, and Zig `Allocator`s

As I've previously mentioned, C has almost no abstraction over memory. This has pros and cons:

- **Pro**: One has full control over memory and can do whatever one wants with it
- **Con**: One has full control over memory and can do whatever one wants with it

With great power comes great responsibility. Mismanaging memory in a language like C with manual memory-management can have great security consequences. At best it can result in a denial of service, and at worst it can let an attacker to execute any arbitrary code. Many languages tried to reduce that responsibility by either imposing coding restrictions, or by erasing the entire question using a Garbage Collector. However, Zig adopts a different approach.

Zig offers several advantages simultaneously:

- Manual Memory Management: You do you. Control over memory is in your hands. No coding restrictions as you see in Rust.

- No Hidden Allocation: None of the standard library APIs allocate on the heap, without you knowing it and letting it happen. Zig utilizes the `Allocator` type to achieve this. Any standard library function that allocates on the heap receives an `Allocator` as parameter. Anything that doesn't do so won't allocate on heap, guaranteed.

— What about libraries other than the standard library?

> It is possible for the developers not to follow this convention in their codebase (whether it's a library or an executable). Although one can allocate without accepting an `Allocator`, it would be contrary to the No Hidden Allocation philosophy of Zig. It specially can be problematic when the project is a library that other people depend on.

- Safety tools to avoid memory leaks e.g. `std.heap.GeneralPurposeAllocator`

Zig doesn't limit you on the way you code as Rust does, helps your remain safe and avoid leaks, but still let you go full rogue as you can do in C. I personally think it might be a convenient middle ground.

zig

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

test "detect leak" {
    var list = std.ArrayList(u21).init(std.testing.allocator);
    // defer list.deinit(); <- this line is missing
    try list.append('☔');

    try std.testing.expect(list.items.len == 1);
}
```

The above Zig code utilized the built-in `std.testing.allocator` to initialize an `ArrayList` and lets you `allocate` and `free`, and test if you're leaking memory:

Note: Some paths are shortened with triple dots for more readability

```
$ zig test testing_detect_leak.zig
1/1 test.detect leak... OK
[gpa] (err): memory address 0x7f23a1c3c000 leaked:
.../lib/zig/std/array_list.zig:403:67: 0x21ef54 in ensureTotalCapacityPrecise (test)
                const new_memory = try self.allocator.alignedAlloc(T, alignment, new_capacity);
                                                                  ^
.../lib/zig/std/array_list.zig:379:51: 0x2158de in ensureTotalCapacity (test)
            return self.ensureTotalCapacityPrecise(better_capacity);
```

```
                                                  ^
.../lib/zig/std/array_list.zig:426:41: 0x2130d7 in addOne (test)
            try self.ensureTotalCapacity(self.items.len + 1);
                                        ^
.../lib/zig/std/array_list.zig:207:49: 0x20ef2d in append (test)
            const new_item_ptr = try self.addOne();
                                                ^
.../testing_detect_leak.zig:6:20: 0x20ee52 in test.detect leak (test)
    try list.append('☔');
                   ^
.../lib/zig/test_runner.zig:175:28: 0x21c758 in mainTerminal (test)
        } else test_fn.func();
                           ^
.../lib/zig/test_runner.zig:35:28: 0x213967 in main (test)
        return mainTerminal();
                           ^
.../lib/zig/std/start.zig:598:22: 0x20f4e5 in posixCallMainAndExit (test)
            root.main();
                     ^


All 1 tests passed.
1 errors were logged.
1 tests leaked memory.
error: the following test command failed with exit code 1:
.../test
```

— What are some of the built-in Zig allocators?

> Zig provides several built-in allocators, including but not limited to:
>
> - FixedBufferAllocator
> - GeneralPurposeAllocator
> - TestingAllocator
> - c_allocator
> - StackFallbackAllocator
> - LoggingAllocator
>
> You can always implement your own allocator.

Billion dollar mistake vs Zig Optionals

This C code crashes abruptly, leaving you no clue other than a `SIGSEGV` about what the hell is going on:

c

```c
struct MyStruct {
    int myField;
};

int main() {
    struct MyStruct* myStructPtr = NULL;
    int value;

    value = myStructPtr->myField;  // Accessing field of uninitialized struct

    printf("Value: %d\n", value);

    return 0;
}
```

On the other hand, Zig does not have any `null` references. It has Optional types instead denoted by a question mark at the beginning. You can assign `null` only to optional types, and you can only reference them when you have checked if they are not null using `orelse` keyword or simply via an `if` expression. You'll end up facing a compile error otherwise.

zig

```zig
const Person = struct {
    age: u8
};

const maybe_p: Person = null; // compile error: expected type 'Person', found '@Type(.Null)'

const maybe_p: ?Person = null; // OK

std.debug.print("{}", { maybe_p.age }); // compile error: type '?Person' does not support field access

std.debug.print("{}", { (maybe_p orelse Person{ .age = 25 }).age }); // OK

if (maybe_p) |p| {
    std.debug.print("{}", { p.age }); // OK
}
```

— Technical guarantees of Zig `null`s

> - An optional pointer is guaranteed to be the same size as a pointer.
> - The `null` of the optional is guaranteed to be address `0`.

Pointer arithmetics vs Zig `Slice`s

In C, an address is represented as a numeric value, which allows developers to perform arithmetic operations on pointers. This feature enables C developers to access and modify arbitrary memory locations by manipulating addresses.

Pointer arithmetic is commonly used for tasks such as manipulating or accessing specific parts of an array or navigating through dynamically allocated memory blocks efficiently, without the need for copying. However, due to the unforgiving nature of C, pointer arithmetic can easily lead to issues such as segmentation faults or undefined behavior, making debugging a true pain in the arms.

Most of such issues can be fixed using `Slice`s. Slices provide a safer and more intuitive way to manipulate and access arrays or sections of memory:

```zig
var arr = [_]u32{ 1, 2, 3, 4, 5, 6 }; // 1, 2, 3, 4, 5, 6
const slice1 = arr[1..5];             //    2, 3, 4, 5
const slice2 = slice1[1..3];          //       3, 4
```

Explicit memory alignment

Each type has an alignment number which defines the memory addresses that are considered legal for that type. Alignment is measured in bytes, and it ensures that the starting address of a variable is evenly divisible by the alignment value. For example:

- The `u8` type has the natural alignment of 1, meaning it can reside in any memory address.
- The `u16` type has the natural alignment of 2, meaning it can only reside in locations of memory with addresses that are evenly divisible by 2, such as 0, 2, 4, 6, 8, etc...
- The `u32` type has the natural alignment of 4, meaning it can only reside in locations of memory with addresses that are evenly divisible by 4, such as 0, 4, 8, 12, 16, etc...

The CPU enforces these alignment requirements. If a variable's type is not properly aligned, it can lead to program crashes such as segmentation faults or result in `illegal instruction` errors.

Now we are going to intentionally create a misaligned pointer to an `unsigned int` in the code below. This code will crash on runtime on most CPUs:

```c
int main() {
    unsigned int* ptr;
    char* misaligned_ptr;

    char buffer[10];

    // Intentionally misalign the pointer so it won't be evenly divisible by 4
    misaligned_ptr = buffer + 3;

    ptr = (unsigned int*)misaligned_ptr;
    unsigned int value = *ptr;

    printf("Value: %u\n", value);

    return 0;
}
```

Working with low-level languages can brings its own challanges like managing the alignment of the memory. Making a mistake can cause a crash, and C won't help with that. How about Zig?

Let's write a similar code in Zig:

```zig
pub fn main() void {
    var buffer = [_]u8{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // Intentionally misalign the pointer so it won't be evenly divisible by 4
    var misaligned_ptr = &buffer[3];

    var ptr: *u32 = @ptrCast(*u32, misaligned_ptr);
    const value: u32 = ptr.*;

    std.debug.print("Value: {}\n", .{value});
}
```

If you compile the code above, Zig will complain and prevent compiling as there is an alignment issue:

```
.\main.zig:61:21: error: cast increases pointer alignment
    var ptr: *u32 = @ptrCast(*u32, misaligned_ptr);
                    ^
```

```
.\main.zig:61:36: note: '*u8' has alignment 1
    var ptr: *u32 = @ptrCast(*u32, misaligned_ptr);
                                   ^
.\main.zig:61:30: note: '*u32' has alignment 4
    var ptr: *u32 = @ptrCast(*u32, misaligned_ptr);
                             ^
```

Even if you try to fool zig with an explicit `@alignCast`, Zig will add a pointer alignment safety check to the generated code on safe build modes to make sure the pointer is aligned as promised. So if the alignment is wrong on runtime, it will panic with a message and a trace for you to have a clue where the problem is coming from. Something that C doesn't do for you:

zig

```zig
pub fn main() void {
    var buffer = [_]u8{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // Intentionally misalign the pointer so it won't be evenly divisible by 4
    var misaligned_ptr = &buffer[3];

    var ptr: *u32 = @ptrCast(*u32, @alignCast(4, misaligned_ptr));
    const value: u32 = ptr.*;

    std.debug.print("Value: {}\n", .{value});
}
// Compiles OK
```

And on runtime you'll receive:

```
main.zig:61:50: 0x7ff6f16933bd in ain (main.obj)
    var ptr: *u32 = @ptrCast(*u32, @alignCast(4, misaligned_ptr));
                                                 ^
...\zig\lib\std\start.zig:571:22: 0x7ff6f169248e in td.start.callMain (main.obj)
            root.main();
                     ^
...\zig\lib\std\start.zig:349:65: 0x7ff6f1691d87 in td.start.WinStartup (main.obj)
    std.os.windows.kernel32.ExitProcess(initEventLoopAndCallMain());
                                                                ^
```

Cool stuff!

Arrays as values

The semantics of C defines arrays to be always passed as references:

c

```c
void f(int arr[100]) { ... } // passed by ref
void f(int arr[]) { ... }    // passed by ref
```

The solution in C is to create a *wrapper* struct and pass the struct instead:

c

```c
struct ArrayWrapper
{
    int arr[SIZE];
};

void modify(struct ArrayWrapper temp) { // passed by value using a wrapper struct
    // ...
}
```

In Zig it just works:

zig

```zig
fn foo(arr: [100]i32) void { // pass array by value

}

fn foo(arr: *[100]i32) void { // pass array by reference

}
```

Error handling

Many C APIs have the concept of error codes where the return value of a function represents either the success status or an integer indicating the specific error that occurred.

Zig uses the same method to handle errors, but improves upon this concept by capturing it in a more useful and expressive manner within the type system.

An error set in Zig is like an enum. However, each error name across the entire compilation gets assigned an unsigned integer greater than 0.

An error set type and a normal type can be combined with the ! operator to form an error union type (example: `FileOpenError!u16`). Values of these types may be an error value, or a value of the normal type.

zig

```zig
const FileOpenError = error{
    AccessDenied,
    OutOfMemory,
    FileNotFound,
};

const maybe_error: FileOpenError!u16 = 10;
const no_error = maybe_error catch 0;
```

Zig does have `try` and `catch` keywords but they are unrelated to `try` and `catch` in other languages as Zig does not have exceptions.

`try x` is a shortcut for `x catch |err| return err`, and is commonly used in places where handling an error isn't appropriate.

Overall, Zig's error handling mechanism is similar to C, but has the type system's support.

— How does Zig determine at runtime whether the returned value represents the error code or the actual output?

> `!T` has to be thought of as:
>
> zig
>
> ```zig
> struct {
>   errorCode: GlobalErrorEnum, // u16
>   result: T
> }
> ```

> And the 0 case for errorCode is considered as the "ok" case. When a function returns !T, it's really `u16 enum` + `T`

— Technical guarantees of Zig errors

> Same error name more than onc gets assigned the same integer value.
>
> zig
>
> ```
> const FileOpenError = error {
>     AccessDenied,
>     OutOfMemory,
>     FileNotFound,
> };
>
> const AllocationError = error {
>     OutOfMemory,
> };
>
> // AllocationError.OutOfMemory == FileOpenError.OutOfMemory
> ```

Everything is an expression

Coming from higher level languages to C, you might have missed features like this:

IIFE.js computedBindings.fs

```
const firstName = "Tom";
const lastName = undefined;

const displayName = (() => {
    if(firstName && lastName)
        return `${firstName} ${lastName}`;
    if(firstName)
        return firstName;
    if(lastName)
        return lastName;
    return "(no name)";
})()
```

The beauty of Zig is that you can behave Zig blocks as if they are expressions.

zig

```zig
const result = if (x) a else b;
```

Or for a more complex example:

zig

```zig
const firstName: ?*const [3:0]u8 = "Tom";
const lastName: ?*const [3:0]u8 = null;
var buf: [16]u8 = undefined;
const displayName = blk: {
    if (firstName != null and lastName != null) {
        const string = std.fmt.bufPrint(&buf, "{s} {s}", .{ firstName, lastName }) catch unreachable;
        break :blk string;
    }
    if (firstName != null) break :blk firstName;
    if (lastName != null) break :blk lastName;
    break :blk "(no name)";
};
```

Every block can have a label like `:blk` and `break` from that block with `break blk:` to return a value.

C has a more complex syntax to deal with

Look at this C type:

c

```c
char * const (*(* const bar)[5])(int)
```

This declares `bar` as a constant pointer to the array 5 of the pointer to the function (int) returning a constant pointer to char. Whatever that means.

There are even tools like cdecl.org which helps you read C types and humanizes them for you. I am pretty sure it might not be that challenging for practical C developers out there to deal with such types. Some people are blessed with such abilities to be able to read the language of gods. But for a simple man like me who rather keep things stupid simple, Zig types are easier to read and maintain.

— Show me a fun C code

> This is a valid C code by the way:
>
> c
>
> ```c
> inline int volatile long typedef _Atomic _Complex const long unsigned A;
> ```
>
> — Show me a fun Zig code
>
> > This is a valid Zig code by the way:
> >
> > zig
> >
> > ```zig
> > var x: *allowzero align(8) addrspace(.generic) const volatile u8 align(8) addrspace(.generic) linksection("unused_feature_section") = undefined;
> > ```

Conclusion

In this blog post, we have discussed some of the issues with C that have led people to seek or create alternatives for this relic of the past.

Zig, in summary, tackles these problems by:

- Zig Comptimes
- Zig Allocators
- Zig Optionals
- Zig Slices
- Zig Explicit Alignment
- Zig Arrays
- Zig error types
- Zig expressions

Related blog posts

- An opinion on what's a good general-purpose programming language

Appreciation

Thanks to my friend Thomas for technically reviewing this blog post.

References

These are some of the references used to write this article:

- Gustedt, J. (2019). Modern C. Manning.
- Zhirkov, I. (2017). Low-Level Programming: C, Assembly, and Program Execution on Intel x86-64 Architecture. Apress.
- Zig Language Reference. [Online]. ziglang.org/documentation/master
- Zig Learn. [Online]. ziglearn.org

TAGS
zig c system-programming programming-languages low-level