Methods in Languages for Systems Programming

I'd been working on this article on and off for around a month or so when I realized it'd probably just end up sitting uncommitted on my laptop forever. I'm sure anyone who has a sporadically-updated blog can relate. Something about it *feels* wrong to me, and I can't quite put my finger on why.

Fast-forward and it's now been two months since I last edited this accursed post. I go to write a blog post following a surge of inspiration. I open Vim. *This* greets me.

```
Untracked (7)
? content/methods.md
? ... a couple more unpublished articles
```

"Oh", I thought, "I'd forgotten about you! Surely I can just make a quick editing pass through this, and I'll finally be done with it." And here I'm typing these words, three hours and a lot of CSS fiddling later.

Fuck it, it's `git add` time. No more editing. Please excuse the disjointed prose, the arguments that feel like they're going somewhere but then just … don't, and the overall aura of unease. *Look at those section headings. They're taunting you, aren't they? And the code blocks, oh god the code blocks there's so many* aaaaaaaa

Ahem.

Back to our regularly-scheduled ill-defined opinions

Lately there has been an explosion of new systems programming languages. Some of these have methods, while others don't. In a way, methods delineate the dividing lines within this new generation of systems languages.

A review of methods in C

In C, it’s trivial to recreate the common pattern of a data structure along with a bunch of functions that operate on that data structure. Let’s take the interface of a dynamic array as an example.

```c
typedef struct Array {
    uint32_t length;
    uint32_t capacity;
    uint32_t element_size;
    float    growth_factor;
    void     *ptr;
} Array;

Array init(
    size_t capacity,
    size_t element_size,
    float growth_factor);

void *at(const Array *a, size_t index);
void push(Array *a, const void *element);
void pop(Array *a, void *element);
void insert(Array *a, const void *element, size_t index);
```

C doesn’t have namespacing of any description, so to avoid stepping on the toes of other code we’ll prefix each of our functions.

```c
Array array_init(
    size_t capacity,
    size_t element_size,
    float growth_factor);

void *array_at(const Array *a, size_t index);
void array_push(Array *a, const void *element);
void array_pop(Array *a, void *element);
void array_insert(Array *a, const void *element, size_t index);
```

And there we have a primordial class with some methods: a data structure, along with some functions that operate on it. (Of course, C doesn’t have full-fat inheritance so it’s doubtful whether I can really call this a class, but we’ll ignore that for the purposes of this article.)

Adding a common prefix to all of our functions helps solidify that each of these operate on the same data structure; that these functions are really “methods”, in a way.

There’s examples of this pattern being used all over the place. I took a look at some of the most popular C projects on GitHub and spotted it in redis, XNU, Neovim, Curl and Nuklear – I suspect it’s close to universal.

A quick survey of the landscape

Let’s take a look at some of the extant C replacements I was referring to earlier, starting with Odin. Odin is a new systems programming language that positions itself as a sane, simple, fun alternative to C. It does not have methods. The FAQ page explains the rationale for this decision:

> **Why does Odin not have any methods?**
>
> We believe that data and code should be separate concepts; data should not have “behaviour”.
>
> Use a procedure.

Similarly, Hare, another minimalist systems programming language, doesn’t include methods either, though I couldn’t find the rationale for this decision. As a counterpoint, Zig and C3 both have methods.

For what it’s worth, antirez of Redis fame thinks C should have methods:

> **@antirez** — 4 November 2022
>
> Who is responsible for the fact C has no simple struct methods that are called with an implicit self pointer? Who is responsible for the fact Undefined Behaviours became a nightmare? And, finally, who is responsible for the stagnation of libc? Simple questions, far effects.

A strong opinion. Maybe that’s just Twitter.

Next: a small demonstration

Let’s take the example of a programming language parser. In Zig:

```
const Parser = struct {
    // fields

    fn parseStatement(self: *Parser) Statement;
    fn parseExpression(self: *Parser) Expression;
    fn expect(self: *Parser, message: []const u8);
    fn next(self: *Parser);
    fn current(self: *const Parser) TokenKind;
};
```

In Odin or Hare I believe you’d just end up writing something equivalent to the following C:

```
typedef struct Parser {
    // fields
} Parser;

Statement  parser_parse_statement(Parser *p);
Expression parser_parse_expression(Parser *p);
void       parser_expect(Parser *p, const char *message);
void       parser_next(Parser *p);
TokenKind  parser_current(const Parser *p);
```

In such a case it makes sense to me to group all this code (the type definition and functions related to the type) into a unit separate from the rest of the codebase; say, `parser.h` and `parser.c`. Now we have a module of code independent of the surrounding system which consists of a data type, plus some behavior which operates upon this data type. It sounds like “data should not have ‘behavior’” is being violated here, no? If not, what’s the difference between this module and a class?

Ideology

I don't see how being forced to write `parser_next(p)` instead of `self.next()` or `array_push(clients, client)` instead of `clients.push(client)` enforces a separation of data and code.

Eskil Steenberg's opinion is that classic C-style `object_method(o)` methods are superior since they are honest about what is actually occurring:

> Object-orientation is something that people think you cannot do in C, because it's not an object-oriented language. But I find that object-orientation is very straightforward in C, and in fact I think it's better in C than in other languages because other languages try to fool you into thinking that there is such a thing as object-orientation in computers when there isn't.
>
> They try to tell you that there is something that has both code and data in it, and that's actually not true at all.

I don't agree with such an argument out of principle. For instance, there is no such thing as a struct; after all, it's just a bag of bytes with certain data located at certain offsets inside that bag of bytes. Each abstraction language designers invent technically does not exist, but that doesn't mean that it can't useful.

Are you really sure you want to type

```
uint32_t length = *(uint32_t *)((uint8_t *)(&parser) + 8);
```

instead of

```
uint32_t length = parser->length;
```

because "structs don't exist"?

On the other hand, I entirely sympathize with wanting to remove unnecessary language complexity and abstractions. It is debatable whether the (minimal) savings in typing methods cause outweigh the (minimal) complexity they add.

In my opinion, methods aren't *evil;* sometimes you just have a bunch of functions that operate on the same bundle of data and as such belong in a logical group, and in those cases a data structure plus some methods seems perfectly fine to me.

A side note on C++ replacements

I put languages such as Rust, Nim, Crystal, Carbon and D in a different bucket to Zig, Hare and the rest. In my mind, the former have more features, more elaborate runtimes and less of a focus on minimalism. (*All* of these points don't apply to *all* of these languages, but the general theme stands.) They fit more closely as replacements to C++, rather than C. Case in point: each of those C++ replacements I listed have methods. As such, I haven't included any of these languages in this article.

Luna Razzaghipour
24 April 2023