# A PARSER–GENERATOR IN $100_{16}$ LINES OF C++

*Please excuse the longish intro – I promise this is going somewhere!*

In the past, my day job consisted of creating and maintaining a Material Flow Control System (often called $MFCS_{opt}$) for warehouses and production plants. This necessitated connecting to various PLCs controlling the mechanical parts – from huge cranes, through conveyor belts, and all the way down to LED systems telling humans what to do.

All those systems had one particular thing in common: they all defined similar, but different text-based protocols to be used to communicate with them over TCP. In a simplified example, that's how a message to a crane could be defined:

| Field name | Field type | Comment |
|---|---|---|
| Begin | ALPHA[1] | Character [ |
| Message Type | ALPHA[3] | "MOV" for move |
| X To | NUM[3] | |
| Y To | NUM[3] | |
| End | ALPHA[1] | Character ] |

*move crane*

Have you ever needed to implement a third-party plaintext protocol? It's as simple as it's boring. And Deity forbid if the documentation changes after initial implementation. You'll waste so much time! At least that's what I told my boss when I started creating a templated declarative parser.

To be fair, I was fairly accurate. I inherited code that used `std::map<std::string, std::string>`, and I wager that I wasted multiple days hunting all the typos in those strings.

Since C++ is a fairly strongly-typed language, there is no need for that – we should be able to leverage the type system to ensure that both our keys and values are correct. Let's discuss the API:

- keys (field names) should be verified at compilation time – none of these pesky typos can pass here,
- values need to be of correct type, not the all-catching `std::string`,
- the code should be as close as possible to the documentation. Ideally, it'd be the documentation.

For example, we could want our MOV telegram to be defined as follows:

```
using mov = message<
  element<struct begin, char_constant<'['>>,
  element<struct message_type, text<3>>,
  element<struct x_to, number<3>>,
  element<struct y_to, number<3>>,
  element<struct end, char_constant<']'>>
>;
```

The usage should be also simple. For receiving:

```
auto data = socket.read();
mov m = mov::parse(data);
log << m.value<x_to>() << m.value<y_to>();
```

And for sending:

```
mov m;
m.value<message_type>() = "MOV";
m.value<x_to>() = 13;
m.value<y_to>() = 37;
socket.send(m.to_string());
```

This approach is Good Enough™. We have type safety, and we can even extend it to use custom types. For example, the above will write the following to the socket (note the padding: zeros for numbers, text would use spaces):

```
[MOV013037]
```

Moving on, the internal implementation is surprisingly simple. The main class template accepts a list of key-type pairs as variadic pack. It uses keys only to map them to values. The type has a bit more to do – each type is expected to know its length, and how to serialize and deserialize itself (or signal an error).

```
template<typename... Elements>
struct message
{
  static message parse(string_view buf);
  void write(char* buf, size_t size) const;
  string to_string() const;
  template<typename Key>
  constexpr auto& value();
private:
  tuple<typename d::element_value_type<
    Elements>::type...> data;
};
```

*Class message definition – shortened and modified to fit here*

```
template<size_t Length>
struct number
{
  static constexpr size_t length = Length;
  using value_type =
    d::type_to_hold_number<Length>;
  static void write(value_type const& val,
    char* buf);
  static value_type parse(string_view buf);
};
```

*Class number definition – shortened and modified to fit here*

As of writing this article, the whole proto.hpp has 248 lines, and I haven't performed any line-saving optimizations on the file.

The code may be accessed at the following address: https://github.com/KrzaQ/protocol_parser_generator.