Code Generator

1. Usage workflow
    1. generate_codec.py usage
        1. help command
        2. preview command
        3. create
2. Baseline requirements for code generation
3. What is Fuzzing and why is it part of this interview ?
    1. Does every fuzz test failure need to be addressed ?
4. Do I need to code generate everything ?
    1. Scenario 1
    2. Scenario 2
5. Common recommendations for this project

C1 BOE Code generator

Your task is to write a python program which can be used to generate a code for a BOE Protocol
https://cdn.cboe.com/resources/membership/US_Options_BOE_Specification.pdf
The python program would be called **generate_codec.py**

Usage workflow

generate_codec.py usage

help command

```
./generate_codec.py --protocol BOE --exchange C1 --version 2.11.66 --preview
```

The above command should show case all files and folders that would be generated by this command.

The above command should generate the following output

```
usage: generate_codec.py [-h] -c CONFIG [-n]

Detailed description of what it does

options:
  -h, --help            show this help message and exit
  -c Config, --config CONFIG
                        YAML config of the spec
  -n, --preview

```

preview command

```
python generate_codec.py --config c1_boe_2_11_66.yaml --preview
```

The above command should generate the following output in preview

```
following paths and files will be generated
src/codecs/C1/BOE/2_11_66/spec/c1_boe_2_11_66.yaml
src/codecs/C1/BOE/2_11_66/BOE_Msgs.h
```

```
src/codecs/C1/BOE/2_11_66/BOE_Stream.h
src/codecs/C1/BOE/2_11_66/BOE_Encoder.h
src/codecs/C1/BOE/2_11_66/BOE_Decoder.h
src/codecs/C1/BOE/2_11_66/test/test.cpp
src/codecs/C1/BOE/2_11_66/fuzzer/fuzz.cpp
```

The above is just a broad set of files we would need. The key thing to remember it list all files that running this command will generate. The header files generated by this program if those header files contain some other files of **#include "bla.h"** those files do not need to printed. Only ones generated from the current run should be added here

create

```
./generate_codec.py --config c1_boe_2_11_66.yaml --preview --create
```

The above command will generate the following files

```
following paths and files will be generated
src/codecs/C1/BOE/2_11_66/spec/c1_boe_2_11_66.yaml
src/codecs/C1/BOE/2_11_66/BOE_Msgs.h
src/codecs/C1/BOE/2_11_66/BOE_Stream.h
src/codecs/C1/BOE/2_11_66/BOE_Encoder.h
src/codecs/C1/BOE/2_11_66/BOE_Decoder.h
src/codecs/C1/BOE/2_11_66/test/test.cpp
src/codecs/C1/BOE/2_11_66/fuzzer/fuzz.cpp
```

The files should be according to standardized coding guidelines and should be cleanly spaced and should not have any lint errors.

The files should leave proper function stubs, if certain pieces need custom logic, so users when they used this program know to modify the file after code generation.

The code generation should generate as much code as possible and should be clean to use. All header files are purely header files, they should not need any external dependencies. Usage of STL is ok, but only if needed.

Performance should be high and as much usage of templates is encouraged to reduce duplication of code. The generated code should be crisp, concise and fast using templates or any other C++11 or above constructs.

Baseline requirements for code generation

- Ostream operators for printing messages should be in a separate header file. So developers can decide if they want to use the ostream provided by code generators or they want a custom one
- you can decide the YAML specification design based on the BOE protocol specification. Idea being someone writes a YAML file expressing BOE spec structure, it should be able to generate C++ code
- If new BOE protocol version comes by, I can just copy paste the yaml and make necessary changes to it to generate new variant of BOE
- Header files should not have any external library dependencies but are allowed to use STL and any C++11 and above features
- Any specific features of C++11 or above if used kindly document it so the interviewer knows
- Code generated protocol library should have a fuzzer test case also generated mandatory
- Code generated protocol library should have basic unit test which encodes and decodes the payload.
- Payload examples can be referred from the BOE spec and hand created manually or can be made as part of the code generation process

What is Fuzzing and why is it part of this interview ?

Candidates are encouraged to read and understand fuzzing as this will be part of the process and will help in thinking about parsers thoroughly and ensure good decision

GitHub - google/fuzzing: Tutorials, examples, discussions, research proposals, and other resources related to fuzzing

Remember while fuzzing is very important, sometimes there are limitations to fuzzing. Documenting the fuzz findings in documentation is encouraged so interviewers can review and give feedback.

Does every fuzz test failure need to be addressed ?

Answer is it depends. Ideally documenting the cause for failure in case of lack of fixes would be very useful. In case fuzz tests denotes

Do I need to code generate everything ?

Well it depends. Lets walk through scenarios where certain pieces are not code generated but the generated code compiles and works as expected

Scenario 1

In order to decode and encode a payload you would need some kind of status object that would tell you whether you decoded or encoded the payload successfully. These objects are going to be common to any protocol BOE or otherwise. Such common stuff should be placed under **src/codecs/common**

- Status object definition - **src/codecs/common/status.h**
- Custom type definitions - **src/codecs/common/types.h**
- Conversions for types - **src/codecs/common/conversions.h**

In such cases the file will get included during code generation step and the functions from these files would be used in code generation. The files will include code handwritten by you. The # include would be code generated.

Scenario 2

What if two messages have same header can I do some inheritance and all through code generation. The important thing to understand here is inheritance makes program slow, to avoid this challenge, we go down route of code generation. You could always use templates if you want to reuse things, they might not be as convenient as inheritance, but they might help you achieve many things inheritance would have helped you with. This does not mean templates is substitute for inheritance, it just means you need to figure efficient and performant solution.

Compiler Explorer is a good tool to profile behaviors and codes to understand the effects of your generated code.

Common recommendations for this project

- Use enums if you can instead of magic numbers
- Raw pointers are ok to use, but ideally if you can achieve performance with smart pointers recommended is to use smart pointers
- No need to handle multithreading behaviors inside the header file. The api user should handle this
- Each data field should have 4 functions. 3 getters and one or more setter.
  - Getter - Const reference, reference and by value
  - Setter - only by value if primitive types in case of complex types you can make your own judgements, clarify the assumptions or decisions you made
- Constexpr always preferred over pre-processor macros. This does not mean you should not use macros, it just means use it wisely only where needed and when the disadvantages to using it irrelevant
- Avoid code generation by concatenating strings, there are lots of templating libraries which you could use. Many html templating libraries can also be used to generate C++ code.
- Ensure headers have right set of namespaces and forward declaration as needed.
- **Design of your generate_codec.py should be modularized, meaning if tomorrow I want to implement a new protocol using this, then I just need to implement a template for the code generation of that protocol and design a yaml spec file for that protocol**

  - **Example we could implement template and a yaml spec for generating code for protocol like iLINK, Otto, Ouch etc. to name a few.**
- Avoid repetitive fields in YAML file, schema of the config should be easy, clean to understand and key names should be intuitive with proper comments
- Decoder and Encoder should ideally allow callbacks so the user can implement the call backs when they want to process or do something with decoded and encoded message
- Status object should support atleast following three status at minimum
  - Need more data
  - Error processing the payload
  - parsed successfully
- Additional status as needed should be used, you can use discretion and design principles to help you guide with this. Sane choices always encouraged