

Review of a Doxygen validator

See the analyzed project: <https://github.com/Bender250/pa193/>

Methods

- CPPCheck with `--enable=all` - Nothing interesting found
- CLion checks - Some style issues warning only
- Valgrind - memcheck
- Debugging
- Analyzed some design issues
- Looked for obsolete (error-prone) C functions usage.
- Manual deep review:
 - Fully analyzed the tokenizer (very strongly believing there are no undetected issues).
 - Fully analyzed the main.cpp (very strongly believing there are no issues at all).
 - Analyzed the parser somehow, catching some issues. However, I can't put my hand into fire for not being some uncovered issues here. The code is very stateful and the design makes it tricky to review.

Tokenizer

- Tokens might overlap in any way. It makes more tricky to write the parser properly, so it is a poor design decision in my view.
- Tokenizer removes all backslashes in `removeBackslashes` method.
 - First, it is problematic for correctness reasons, but if we use it only as a validator, this seems to be OK.
 - Second, it changes the input string and consequently changes the string passed to `Tokenizer::tokenize(std::string&)`, which might be weird for the library user and is not documented.
 - Moreover, this feature seems to be required for correct functionality. If we removed the '&', the parser would get wrong positions of the tokens. However, it seems that it could lead "only" to wrong parsing (and consequently to false positive/negative). However, it can never result in `std::out_of_range` thrown by `std::string::substr(std::string::size_type, std::string::size_type)` or in some overrun.
 - When user passes two different instances of the same string to tokenizer and parser, it can also lead in unexpected results.
 - When user wants to read the string after passing it to the tokenizer, he might get some unexpected results.
 - When there is a large `"\n\n\n"` sequence, it takes $O(n^2)$ time, which can be used for DoS in some cases (e.g. online validator).
 - Some measurements for many `\` characters in file: (~0:53 for ~2MiB file, ~4:03 for ~4MiB file, ~20 minutes for ~8MiB file), which is a great opportunity for an attacker that wants to cause (D)DoS.
 - You can compare run times with `dos/tricky.c` and `dos/nontricky.c`.
- Except the `removeBackslashes` function, it is good that the tokenizer always finishes in a linear time with respect to the input length.
- It seems to take about linear amount of memory. That's quite decent, but I hope it could be better. It could accept a stream and use a constant amount of memory. If I wanted to DoS a server running this validator, I'd pass many large `\\\\\\\\\\â€¦!` inputs in parallel. When using many `\`s, it would ensure allocating such memory for a long time with high CPU consumption.

When considering the tokenizer alone, it has a tricky API with some tricky undocumented (but seemingly intended) behaviour. When considering the tokenizer and how it is used in the `main.cpp` file, it works correctly. However, the design (namely overlapping tokens) makes it hard for the parser to process the tokenizer output correctly.

Parser

- As noted in tokenizer, it assumes that it has input with `\` escape sequences filtered out by the tokenizer.
- The parser seems to be very stateful, which I don't like. It makes hard for me to reason about it.
- The parser uses a linked list. Although it erases some tokens, it does not increase the complexity. Unlike the tokenizer, the parser seems to always complete in $O(n)$.
- Some wrongly processed inputs found. (No memory corruption so far, though.) See the `wrongly-processed` directory.
- Potential memory corruption: When `Parser::filterUnreachableNontokens()` finds the "at" token (i.e. @ symbol *not* followed by a known keyword like `author`, `version`, `param` and so on) in a doxygen comment, it calls `Parser::checkForBadKeyword(std::list< Tokenized >::iterator)` without any further check. However, the `Parser::checkForBadKeyword(std::list< Tokenized >::iterator)` reads the current *and the subsequent* token *without any check*. If there is a @ symbol at the end of the file inside an unclosed doxygen comment, the method will try to dereference iterator `nonterminalsList.end()`, which is undefined. In my experience, it returns some rather random results. In some cases (depending on the luck in this Russian roulette), the called `isKeyword` method may return true, which would lead in trying to erase `nonterminalsList.end()`, which is likely to do some memory corruption. See `potential-memory-`

Review of a Doxygen validator

`corruption/isKeyword-read-overflow.c` as an example of such a tricky input. Note that the file intentionally does not end with new line.

- When `Parser::filterUnreachableNontokens()` finds an ordinary block comment, it looks for the end of the comment. It also detects unterminated comments. There are two checks in the loop. One check looks for end comment, the other check looks for end of file. However, these checks are in wrong order, so unterminated non-Doxygen comment causes dereferencing `nonterminalsList.end()` before checking for `it == nonterminalsList.end()`.
- The same applies for string literals in `Parser::filterUnreachableNontokens()`.
- There is a very similar issue with `//` comments. The code slightly differs, but the issue is the same. End of the comment (i.e. end of line) is checked before end of the token list is checked.
- It uses modern safe C++, which is not so much error prone as old C functions.