flecstan:

FleCSI Static Analyzer

Part I: Status, and Basic Use

12 March 2019

Martin Staley

CCS-7: Applied Computer Science

Introduction

The **FleC**SI **St**atic **An**alyzer, now named **`flecstan`**, has evolved and improved considerably since I last spoke about my static analysis work, in October.

- Early work by another employee was integrated into Kitsune, for no particular reason I'm aware of. `flecstan` is now **entirely standalone** (except it uses the clang/llvm libraries).
- `flecstan` can now ingest **JSON "compilation database"** files, which I understand can be generated by cmake (or you can easily write them).
- `flecstan` can also ingest **C++ files** directly, along with **compilation flags** to be passed through to clang++.
- In fact, we have outfitted `flecstan` so that you can provide, on its command line, an essentially ***arbitrary* set of JSON files, C++ files, and clang++ options**.
- `flecstan` has **many of its own command line options**.
- We now **analyze codes** and **generate reports**. (Earlier, we just collected information and placed it into a YAML document, for which later analysis was intended.)

In short, `flecstan` has become a **very flexible tool that can be used right now**.

Example: with clang++

Let's begin with a very simple example, in order to illustrate how we'd replace a `clang++` compilation with a `flecstan` analysis. (These two aren't FleCSI codes; they don't need to be.)

one.cc

```
#include <iostream>

#ifdef FUNA
   void a()
   {
      std::cout << "a()" << std::endl;
   }
#endif

#ifdef FUNB
   void b()
   {
      std::cout << "b()" << std::endl;
   }
#endif
```

two.cc

```
#include <iostream>

extern void a();
extern void b();

#ifdef FUNC
   void c()
   {
      std::cout << "c()" << std::endl;
   }
#endif

int main()
{
   a(); b(); c();
}
```

Here's a basic, one-line `clang++` compilation. (We realize that `one.cc` doesn't need `-DFUNC`, and `two.cc` doesn't need `-DFUNA` or `-DFUNB`. However, for a one-line full compilation...)

```
clang++ -DFUNA -DFUNB -DFUNC one.cc two.cc
```

Question: what's a corresponding `flecstan` command?

Example: with flecstan

Here are the same two files again. We wish to perform a `flecstan` analysis that treats each file in the same way our compilation did, with respect to compiler flags.

one.cc

```
#include <iostream>

#ifdef FUNA
   void a()
   {
      std::cout << "a()" << std::endl;
   }
#endif

#ifdef FUNB
   void b()
   {
      std::cout << "b()" << std::endl;
   }
#endif
```

two.cc

```
#include <iostream>

extern void a();
extern void b();

#ifdef FUNC
   void c()
   {
      std::cout << "c()" << std::endl;
   }
#endif

int main()
{
   a(); b(); c();
}
```

Here's a simple way to do it:

```
flecstan -flags -DFUNA -DFUNB -DFUNC -cc one.cc two.cc
```

The `-flags` option means "pass these flags to clang++," while `-cc` (which, as we'll soon see, is not generally needed) means we'll be providing one or more C++ files.

Example: with flecstan, again

Looking at the same two files yet again, we see that the first only needs `-DFUNA` and `-DFUNB`, while the second only needs `-DFUNC`.

one.cc

```
#include <iostream>

#ifdef FUNA
   void a()
   {
      std::cout << "a()" << std::endl;
   }
#endif

#ifdef FUNB
   void b()
   {
      std::cout << "b()" << std::endl;
   }
#endif
```

two.cc

```
#include <iostream>

extern void a();
extern void b();

#ifdef FUNC
   void c()
   {
      std::cout << "c()" << std::endl;
   }
#endif

int main()
{
   a(); b(); c();
}
```

Consider this variation of the earlier `flecstan` command:

```
flecstan -flags -DFUNA -DFUNB -cc one.cc -flags -DFUNC -cc two.cc
```

A given C++ file is compiled with `clang++` flags as given in the *last preceding* `-flags` specification. We can, in effect, *provide any number of independent compilation commands*.

Example: with flecstan, one more time

Looking at the same example one more time, we remark that `flecstan` automatically considers any `*.cc`, `*.cpp`, `*.cxx`, and `*.C` arguments to be C++ files.

one.cc

```
#include <iostream>

#ifdef FUNA
   void a()
   {
      std::cout << "a()" << std::endl;
   }
#endif

#ifdef FUNB
   void b()
   {
      std::cout << "b()" << std::endl;
   }
#endif
```

two.cc

```
#include <iostream>

extern void a();
extern void b();

#ifdef FUNC
   void c()
   {
      std::cout << "c()" << std::endl;
   }
#endif

int main()
{
   a(); b(); c();
}
```

The file names end in `.cc`, so we can omit the `-cc` in either of the earlier `flecstan` commands:

```
flecstan -flags -DFUNA -DFUNB -DFUNC one.cc two.cc

flecstan -flags -DFUNA -DFUNB one.cc -flags -DFUNC two.cc
```

You can still use `-cc` if you wish, or use it for C++ files that have other (or no) file extensions.

Examples: Summary

Here's our original `clang++` command, along with several `flecstan` variants – some given already, some not – with their command line arguments all lined-up for your viewing pleasure:

`clang++`:

```
clang++          -DFUNA -DFUNB -DFUNC     one.cc                        two.cc
```

`flecstan`:

```
flecstan -flags -DFUNA -DFUNB -DFUNC -cc one.cc                -cc two.cc
flecstan -flags -DFUNA -DFUNB -DFUNC -cc one.cc                    two.cc
flecstan -flags -DFUNA -DFUNB -DFUNC     one.cc                -cc two.cc
flecstan -flags -DFUNA -DFUNB -DFUNC     one.cc                    two.cc

flecstan -flags -DFUNA -DFUNB         -cc one.cc -flags -DFUNC -cc two.cc
flecstan -flags -DFUNA -DFUNB         -cc one.cc -flags -DFUNC     two.cc
flecstan -flags -DFUNA -DFUNB             one.cc -flags -DFUNC -cc two.cc
flecstan -flags -DFUNA -DFUNB             one.cc -flags -DFUNC     two.cc
```

Perhaps that lined-up list looks goofy and OCD, but we hope it helps you see how `flecstan` processes its command line arguments. **Don't worry, there's much more to come....**

JSON “Compilation Databases”

The Clang Tooling API provides direct functionality for reading a “compilation database” in JSON file format. Consider the following file:

one-two.json

```
[
  {
    "directory" : "/home/staley/flecstan/example",
    "command"   : "clang++ -DFUNA -DFUNB one.cc"
    "file"      : "one.cc"
  },
  {
    "directory" : "/home/staley/flecstan/example",
    "command"   : "clang++ -DFUNC two.cc"
    "file"      : "two.cc"
  }
]
```

Given this file, we can initiate a `flecstan` analysis by invoking either of these commands:

```
flecstan -json one-two.json

flecstan one-two.json
```

The second is fine because `flecstan` recognizes the `.json` extension, as it should. (If your JSON file doesn’t end in `.json`, for whatever reason, then just use the `-json`.)

Regarding FleCSI: supposedly, `cmake` can be told to produce these JSONs. Yay!

Multiple JSON Files

A compilation database file might typically represent an entire build. However, there's nothing fundamentally wrong with having more than one such file. For example:

one.json

```
[
  {
    "directory" : "/home/staley/flecstan/example",
    "command"   : "clang++ -DFUNA -DFUNB one.cc"
    "file"     : "one.cc"
  }
]
```

two.json

```
[
  {
    "directory" : "/home/staley/flecstan/example",
    "command"   : "clang++ -DFUNC two.cc"
    "file"     : "two.cc"
  }
]
```

`flecstan` has you covered, in precisely the way you'd expect:

```
flecstan one.json two.json
```

In fact, you can arbitrarily combine direct compilations of C++ files, as shown earlier, and compilations based on JSON files.

Consider that someone builds a FleCSI-based library that has its own build system (think, for example, FleCSALE), and you have your own set of files that will link with the library. Then, you can analyze all elements together, as a whole, by invoking a single `flecstan` command.

Usage Notes

Regarding input files and compiler flags, here are a few miscellaneous remarks.

- `flecstan` also provides a `-dir` (directory) option. C++ and JSON files that are given later in the command line, are searched for in (only) the given directory.
- `flecstan -cc foo` looks for (`foo`, `foo.cc`, `foo.cpp`, `foo.cxx`, `foo.C`), in that order. Comments are welcome regarding other extensions, or the ordering. A Windows user, for example, ~~deserves to have a hard time~~ might want support for ALL CAPS file extensions.
- `flecstan -json bar` looks for (`bar`, `bar.json`), in that order.
- Invoked with *no* C++ or JSON files, `flecstan` looks for the file `compile_commands.json`, first in any given `-dir` directory, then in the current directory. (`compile_commands.json` appears to be a standard thing in the clang/llvm world.) Exception: `flecstan` invoked with just `-version` or `-help` prints its version, or (not-yet-helpful) help, then exits.
- The ordering of the various command options we've talked about is important! They're read left-to-right. Where `-dir` and `-flags` are used, their most recent values are used.

All of this, and much more, will eventually be documented in a more precise way. For now, in this talk, we won't bore you with too many details.

Output and Analysis

So, `flecstan` is flexible in terms of its input. Does it do something useful, too...? Fortunately, yes. `flecstan` prints up to four “sections” of information, all to C++ standard output:

- **Arguments**
  This is an “FYI” section, with `flecstan` essentially telling you how it’s interpreting the command-line arguments. It helps you to see if you’re doing what you wanted to do.
- **Compilation**
  Here, `flecstan` prints some basic information about what it finds in (1) the preprocessing stage, and (2) the abstract syntax tree (AST) visitation stage, of each C++ file.
- **Analysis**
  Real analysis happens here, with files having been compiled and their information aggregated. We report, for example, if a task executed anywhere was registered nowhere.
- **Summary**
  In this section, `flecstan` summarizes any issues it found in the analysis stage, and makes some closing remarks.

Intentionally, `flecstan` is, by default, somewhat ~~obnoxious~~ verbose. However, we can switch off any of the sections, or switch off particular types of output (e.g. warnings) across the board.

COLOR!!!

For your entertainment and amusement (and, we hope, because it's ***helpful***), `flecstan`'s output is colorized. Our choice of colors was very deliberate, and involved several considerations.

- Neutral colors for "neutral" and "structural" printing. **Section headers are gray**, for example, and **debug information is white**. (White wouldn't work well on a white screen, of course, but ~~no sane person has a glaring white screen~~ debug mode is intended just for developers, and is unlikely to be used very much.)
- Increasingly "hot" colors for diagnostics: **Notes**, **Warnings**, and **Errors**. You could also think of these as "traffic light" colors, with the obvious interpretation.
- Cooler colors, courtesy Roy G. Biv, for printing that's more informational than diagnostic in nature. This includes **file names**, **report labels**, and **report text**.
- Basic ANSI color-code escape sequences for terminal output, likely to work on most terminals. See `https://en.wikipedia.org/wiki/ANSI_escape_code` for more information.
- To avoid visual overload, we don't have too many colors! The above, in fact, is it.

You can switch coloring off if you don't want it. Perhaps, for example, you're redirecting output to a plain old text file, and don't want color-code escape sequences trashing up the file.

Output: Example 1

Command: `flecstan`

```
--------------------------------------------------------------------------------
Arguments
--------------------------------------------------------------------------------

Note:
   No input file(s) specified; looking for compile_commands.json.

Warning:
   Could not find JSON file compile_commands.json.

Error:
   No input file(s) specified, and default (compile_commands.json) not found.
   You can provide JSON and/or C++ input files.

--------------------------------------------------------------------------------
Compilation
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
Analysis
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------

Warning:
   FleCSI static analysis may be incomplete; errors occurred.
```

Remarks:

While more verbose, this mirrors the behavior I see with g++ and `clang++`. The command, run by itself, with no options, produces an error – as opposed to defaulting to a "usage" message or other behavior.

This example illustrates `flecstan`'s output style...

Section headings are, well, obvious.

Notes, warnings, and errors are colorized.

Output elements are spaced apart by blank lines. This is ~~because the author is just spaced out~~ to enhance readability. (But feel free to try `-short` form output.)

The content of each diagnostic is indented.

Our intention here is *clarity*. And, in particular, to give us a better visual experience than we may have when viewing diagnostics from typical C++ compilers. (A low bar, admittedly, to achieve.)

Output: Example 2

Command: `flecstan one.cc two.cc`

```
--------------------------------------------------------------------------------
Arguments
--------------------------------------------------------------------------------

Note:
   Queueing C++ file one.cc

Note:
   Queueing C++ file two.cc

--------------------------------------------------------------------------------
Compilation
--------------------------------------------------------------------------------

File:
   one.cc

Scanning for FleCSI macros...

Visiting the C++ abstract syntax tree...

File:
   two.cc

Scanning for FleCSI macros...

Error:
   Clang reports an error:
      use of undeclared identifier 'c'
   File: two.cc
   Line: 15

Visiting the C++ abstract syntax tree...

--------------------------------------------------------------------------------
Analysis
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------

Warning:
   FleCSI static analysis may be incomplete; errors occurred.
```

Remarks:

Oh no! There's an error! (Such a rare thing in the computer science world.)

Recall our earlier `one.cc` and `two.cc` files. It turns out that the error occurred because we forgot the `-DFUNC` compiler flag.

In fact, we also forgot the `-DFUNA` and `-DFUNB` compiler flags. Review the codes, though, and you'll see why no error showed up due to this omission. (Hint: we're just compiling, not linking, with `flecstan`.)

A takeaway message: `flecstan` intercepts `clang++`'s error messages, and emits them in our own format. We anticipate that some users will welcome this.

Output: Example 3

Command: `flecstan -flags -DFUNA -DFUNB one.cc -flags -DFUNC two.cc`

```
--------------------------------------------------------------------------------
Arguments
--------------------------------------------------------------------------------

Note:
   Found flag -DFUNA.

Note:
   Found flag -DFUNB.

Note:
   Setting C++ flags to -DFUNA -DFUNB.

Note:
   Queueing C++ file one.cc

Note:
   Found flag -DFUNC.

Note:
   Changing C++ flags to -DFUNC.

Note:
   Queueing C++ file two.cc

--------------------------------------------------------------------------------
Compilation
--------------------------------------------------------------------------------

File:
   one.cc

Scanning for FleCSI macros...

Visiting the C++ abstract syntax tree...

File:
   two.cc

Scanning for FleCSI macros...

Visiting the C++ abstract syntax tree...

--------------------------------------------------------------------------------
Analysis
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------

Note:
   FleCSI static analysis completed.
```

Remarks:

That’s better. No errors or warnings. Just “cool” colors. But wait, where’s the analysis? Well, these two example codes had no actual FleCSI content, so there’s no FleCSI-related information to report!

Although you can see now that `flecstan` is an amazing, incredible, and earth shattering tool, we’re deferring real FleCSI-equipped examples to another talk! There’s just too much to show in this respect.

Command Options – 1

We've already spoken about some of `flecstan`'s command line options. Here's a complete list, roughly organized in terms of the nature of what each option does.

Simple Options

`-version`
Print version number.

`-help`
Print help. (Not helpful yet. `flecstan` is a work in progress.)

Compilation

`-clang++`
Stipulate a particular `clang++`, e.g. `/usr/bin/clang++`, to the clang/llvm API. (Note: the API accepts this information, but it isn't entirely clear if, when, and how the API uses it!)

`-flags`
Flags for `clang++`.

`-cc-line`
Print some information about the compile commands that `flecstan` creates, behind-the-scenes, for each input C++ file. (I should give this option a better name.)

Files

`-dir`
Directory for *input* files (JSON and C++). Does *not* apply to an output YAML file as described below.

`-json`
Input `JSON` "compilation database" file(s).

`-cc`
Input `C++` file(s).

`-yaml`
Output YAML file, if you want `flecstan` to create a record of the FleCSI constructs it finds. We originally planned for `flecstan` to do its job by aggregating data (across all input files) into a YAML output file, then processing the file. This is no longer how `flecstan` works, but we still make good use of the YAML-enabled data structures, internally. And, the ability to write such a file may eventually prove to be useful.

Command Options – 2

Output: Sections

`-[no-]arguments`
Yes/no: print the Arguments section? The section is on, by default, because it's very informative about how the command line arguments are being interpreted. Knowledgeable users may want to switch it off, though, because they may regard what it tells them to be obvious and not needed.

`-[no-]compilation`
Yes/no: print the Compilation section? This is a useful section for watching how the compilation proceeds across files and stages (preprocessing and AST visitation), but may not be important for users who are focused on the bottom line: code analysis. ⟵ the bottom line of this paragraph, at least

`-[no-]analysis`
Yes/no: print the Analysis section? Most users will want to see this section.

`-[no-]summary`
Yes/no: print the Summary section? This section repackages some of the analysis into a shorter form, and makes a few closing remarks.

Output: Elements

`-[no-]files`
Yes/no: print file names as we visit the files?

`-[no-]reports`
Yes/no: print `flecstan` reports? "Reports" consist of certain informational messages; we'll clarify this elsewhere.

`-[no-]notes`
Yes/no: print `flecstan` notes? Includes `clang++` diagnostics that map into our notes.

`-[no-]warnings`
Yes/no: print `flecstan` warnings? Includes `clang++` diagnostics that map into our warnings.

`-[no-]errors`
Yes/no: print `flecstan` errors? Includes `clang++` diagnostics that map into our errors.

`-[no-]debug`
Yes/no: debug mode? *Off* by default. Users won't want debug mode, which is intended for `flecstan` developers.

Command Options – 3

Output: Formatting

-long
Long-form output; the default.

-short
Short-form output; less bulky.

-[no-]color
Yes/no: Colorize the output.

-print
Print mode. May be useful if you redirect output to a file, for the purpose of printing on an actual printer. Switches coloring off, and emits formfeeds between sections.

Output: General Content

-quiet
Quiet mode. Switches off debug mode (which in fact would have already been off by default), as well as notes and warnings (but not errors, reports, or file names), section headings, and all sections except Analysis.

-verbose
Verbose mode. Switches on section headings, all sections, file names and reports, and notes, warnings, and errors.

-[no-]columns
Yes/no: print column numbers, too, wherever and whenever line numbers are printed. Off, by default, because many people find columns numbers to be distracting, and to be too easy to confuse with line numbers.

-[no-]headings
Yes/no: print section headings for our four sections (Arguments, Compilation, Analysis, Summary). You can slightly reduce output bulk by switching off the headings.

Command Options: Remarks – 1

- **Sections**. All sections are *on* by default. Most printing, in fact, is on by default, except for debug output (which few people will want), and a couple of obscure things.
- **Error Persistence**. Even if a section is switched off, *errors* that occur in that section are still printed. (But you can explicitly switch off all printing of non-fatal errors by using `-no-errors`.)
- **Independence**. Printing-related flags are generally independent, in terms of their effects. For example, `-short` and `-long` apply to what otherwise gets printed, that is, to anything that wasn't switched off. `--no-headings` scraps the bulky section headings, but leaves printing within the sections intact unless you switched off such printing by other means.
- **Ordering**. Often, but not always, the ordering of command line options matters. We saw this already when we gave different `-flags` for different `-cc` files. Ordering of other options can have minor effects. Say the Arguments section is on, as it is by default. Give `-no-color` first, and you'll see no color anywhere. Give it later, and the Arguments section will colorize its chatter about arguments...until it gets to the `-no-color` argument! This behavior is intentional.
- **Sections vs. Content**. Section printing (Arguments, Compilation, Analysis, Summary) is sort of "orthogonal" to content printing (notes, warnings, errors, file names, reports). Omit a section, and you'll see nothing (except errors) that would have printed in that section. Switch off warnings, and they're switched off everywhere – in all sections.

Command Options: Remarks – 2

- **Multiple Forms**. All options have multiple forms. All support both "`--`" and "`-`" as a prefix: you can say `-version` or `--version`, for instance. Many options can be expressed in either singular or plural form, the latter at least if it "sounds right": `-warning` and `-warnings`, for example, but not `-quiets`. A few have other variations: `-cxx` and some others in place of `-cc`, and either `-headers` or `-headings` for section-header printing (I thought that the two words were similar enough that they'd be easily confused.)
- **Multiple Forms??** Allowing multiple forms, as just described, may be excessive. We wish to keep users from having to remember tiny details ("was such-and-such singular or plural?"), but would also prefer to avoid the possible side effect: one user says `-warning` while another says `-warnings`, then they see each other's commands and wonder "what's the difference?"
- **Not Yet Perfect**. Some of the options need a bit of work. Section toggles, for example, should perhaps begin with "`-section`." I also need to check that my options don't conflict with those of `clang++`. If they do, then certain parsing indeterminacies might be possible, in view of `flecstan`'s ability to pass `-flags` through to the compiler.
- **Rules**. Our rules all make sense, I believe, but they should all be documented precisely. **Our goal of having so many options is to make `flecstan`'s behavior highly customizable, so that it's as useful as possible to a wide variety of users.**

Getting `flecstan`

`flecstan` is available now, on Github, in FleCSI's **`feature/staley/flecstan`** branch. All associated code is in **`flecsi/tools/flecstan/`**, the contents of which are as follows:

- **`readme`**
  The one and only.
- **`CMakeLists.txt`**
  For FleCSI's sake, and if you like `cmake`.
- **`compile.sh`**
  Simple `bash` compilation script, if you don't like cmake.
- **`src/*.h`**
  `src/*.cc`
  Header and source files. (Duh.)

At the moment, we don't have *formal* tests, examples, or documentation. Efforts have focused on **learning clang/llvm**, providing lots of **command line options**, allowing for a **wide variety of input** (multiple JSONs, and/or C++ files with arbitrary compiler flags), writing **infrastructure** (e.g. for the formatting and coloring of output), and focusing on **obvious candidates for analysis**, such as the mistake of trying to execute an unregistered task.

Goals, and Future Work

- **Robustness**. Careful work (not that we've thus far been careless) might make this very FleCSI-detail-dependent code a bit less brittle with respect to future changes in FleCSI.
- **More Analysis**. We've worked on infrastructure (e.g. for `flecstan`'s reports), on examining codes and collecting information, and on learning clang/llvm. More analysis of the collected information is still possible.
- **User Input**. Ask ~~victims~~ users to try `flecstan` and provide feedback.
- **FleCSI Integration**. Truly integrate with the rest of FleCSI. `flecstan` is usable right now, but let's make things easy for users. For example, building a FleCSI-based code should, at least optionally, create a JSON file.
- **Documentation, Test Suite, & Examples**. I have many of my own tests and examples, but should put them in a clearer form and place them into the repo. Documentation is in my brain now, and needs external backup.
- **Output Control**. Allow for finer-scale control of the what/when/where/how of what `flecstan` reports. Users will want neither too little nor too much information, and what qualifies as such is a matter of opinion.
- **Refactoring**. `flecstan` is in good shape, but could use a basic cleanup/refactoring pass.
- **More Talks**. I'd like to give at least one more talk, soon, to show several examples of specific FleCSI codes and their analysis with this tool. Users should probably be invited.