Syntax rules

Mehrdad Hasanpour
July 23, 2016

1 RULES

- Identation rule

```
switch(a) {
  case "a":
    break;
  case "b":
    break;
}
```

  - **Use two-spaces identation**

- Whitespace rule
  - **Use one space before opening braces `{` and parenthesis `(` (unless they are not crowded) (recommended).**
  - **Use one space after comma `,` and not before it.**

- Naming

```
bool Input::atom_style () {
  auto atom_style_identifier = parser->get_identifier();
  if (! parser->end_of_line ()) error->all (FILE_LINE_FUNC, "Invalid
      syntax");
  if (atom_style_identifier != "simple") {
    delete atom_data;
```

  - **Use _ instead of space.**
  - **variables are defined with small characters.**
  - **Typedefs and Classnames are started with capital letters.**
  - **MACROS are all in capitals like `FILE_LINE_FUNC` but try not to use them.**

- Other
  - **Do not open braces in a new line. (recommended)**
    Like this:

```
switch(a) {
  case "a":
    break;
  case "b":
    break;
}
```

    And not like this:

```
switch(a)
{
  case "a":
    break;
  case "b":
    break;
}
```

2 Recommended Reading

Some hints to go on: **For more information see here**

- Write programs for people, not computers.
  - A program should not require its readers to hold more than a handful of facts in memory at once.
  - Make names consistent, distinctive, and meaningful.
  - Make code style and formatting consistent.
- Let the computer do the work.
  - Make the computer repeat tasks.
  - Save recent commands in a file for re-use.
  - Use a build tool to automate workflows.

- Make incremental changes.
  - Work in small steps with frequent feedback and course correction.
  - Use a version control system.
  - Put everything that has been created manually in version control.
- Don't repeat yourself(or others).
  - Every piece of data must have a single authoritative representation in the system.
  - Modularize code rather than copying and pasting.
  - Re-use code instead of rewriting it.
- Plan for mistakes.
  - Add assertions to programs to check their operation.
  - Use an off-the-shell unit testing library.
  - Turn bugs into test cases.
  - Use symbolic debugger.
- Optimize software after it works correctly
  - Use a profiler to identify bottlenecks.
- Collaborate
  - Use pre-merge code reviews.
  - Use pair programming if necessary.
  - Use an issue tracking tool.