# C Pretty Evaluator

This is a C application that will be used by COMP 206 and COMP 310 students to evaluate the professional quality of their source code. This program will evaluate the student's code in these areas:

- Modular programming
- Code indentation
- Commenting
- Documentation
- Poor variable names
- Built-in test cases

## Sample Test Code

We will use real student programs from COMP 206 and COMP 310 to test this application. In the beginning the developer can test the application with their own assignments.

## Command-line Syntax

This program is designed to run only on mimi, but the source code should be recompilable on other computer systems (like Mac and Windows, and other Linux distros).

The code evaluator is a command-line C program having the following syntax:

$ ceval filename1.c filename2.c … filename.c
$ ceval *.c

> By default, the C evaluator will test the filenames provided for all the evaluation areas above. The filenames can have any file extension. They do not need to have the .c file extension. The output of the evaluation is displayed to standout, which can be redirected by the user to a file. The output of this command is a summary table followed by a detailed analysis. The output will look like this:

```
ceval version 1.0
---= Code Evaluator Summary =---
Modular programming rating : [n]/[m] hits = [100-%]
Code indentation rating    : [n]/[m] hits = [100-%]
Commenting rating          : [n]/[m] hits = [100-%]
Documentation rating       : [n]/[m] hits = [100-%]
Poor variable names rating : [n]/[m] hits = [100-%]
Built-in test cases rating : [n]/[m] hits = [100-%]

---= Flagged Code =---
[filename][line number][Error message]
[copy of the line of code flagged]
[filename][line number][Error message]
[copy of the line of code flagged]
--= ceval END =--
```

$ ceval -help

The -help switch takes precedence. If there are other switches or filenames present at the same time as the -help switch, then these other switches/filenames are ignored. Only the -help switch is executed. The -help switch displays information about the syntax of the command-line and a description of what each switch does. It will basically display the same information you see here in this document but formatted and presented in a more appropriate way. You can look at the Linux man output for an example of what the format of -help should look like.

$ ceval -ssf filename1.c filename2.c … filename.c
$ ceval -ssf *.c

The -ssf switch modifies the standard execution of the ceval program by removing the modular programming test. The -ssf switch means Single Source File. It is assumed that the user is writing a single source file application. The summary output will display DEACTIVATED in the place where it would display the rating for modular programming.

$ ceval -nodoc filename1.c filename2.c … filenameN.c
$ ceval -nodoc *.c

The -nodoc switch modifies the standard execution of the ceval program by removing the source file documentation test. The -nodoc switch means No Documentation in the file. The summary output will display DEACTIVATED in the place where it would display the rating for source file documentation.

$ ceval -notest filename1.c filename2.c … filenameN.c
$ ceval -notest *.c

The -notest switch modifies the standard execution of the ceval program by removing the built-in test cases architecture. The -notest switch means No Test cases. The summary output will display DEACTIVATED in the place where it would display the rating for test cases.

$ceval -novars filename1.c filename2.c … filenameN.c
$ ceval -novars *.c

The -novars switch modifies the standard execution of the ceval program by removing the check for variable names. The -novars switch means No Variables testing. The summary output will display DEACTIVATED in the place where it would display the rating for test cases.

$ ceval -ssf -nodoc -notest -novars filename1.c filename2.c … filenameN.c
$ ceval -ssf -nodoc -notest -novars *.c

Any of the modification switches can be applied at the same time. They can appear in any order. One, two or all three switches can appear. When a switch appears then it will affect the evaluation of the source code as already described. The switches must appear before the file names.

## How to Rate the Code

Presented here are the rules for evaluating the quality of the source code.

- Modular programming
  - Rule 1: Every .c file must have a matching .h file.
  - Rule 2: Every matching .h file must be protected from multiple includes.
  - Rule 3: A .c file cannot have function prototypes from another .c file.
  - Rule 4: A .c file cannot have an extern statement.
  - Rule 5: There must be at least one private function in a .c file.
  - Rule 6: There must be at least one private global variable in a .c file.
  - Rule 7: All #include directives must be at the beginning of the file, before the comments that **may** exist at the top of the file.
- Code indentation
  - Rule 1: Code must be indented with **at least** 4 spaces or **exactly** 1 tab **per nest**.
  - Rule 2: The open and close curly brackets must be aligned with the beginning of the statement. Two forms are permitted:

    ```
    statement {
    }
    ```

    ```
    statement
    {
    }
    ```

  - Rule 3: All nested statements are indented to the exact same column.
  - Rule 4: If the developer used spaces to indent, then all indentation must use spaces. If the developer used tabs to indent, then all indentation must use tabs.
- Commenting
  - Rule 1: The total number of lines, in a .c file, with comments must be greater than the number of functions + global variables in that same .c file.
  - Rule 2: Each function should have at least one comment.
  - Rule 3: Each .c file, at the first line of the file, must have a comment of this exact form:

    ```
    // Programmer: word1 word2
    ```

    ```
    /* Programmer: word1 word2 */
    ```

    ```
    /*
    Programmer: word1 word2
    */  ← this last end of comment bracket can be further down
    ```

    The tokens word1 and word2 is assumed to be the name of the programmer. It does not matter what they provide for word1 and word2. It does not matter is there are more than two tokens. The program simply counts the number of tokens and flags an error when there are less than 2 tokens.
- Poor variable names
  - Rule 1: Variables must be more than 2 letters in length.
  - Rule 2: User defined type names must be all CAPS.
  - Rule 3: Constants must be all CAPS.

- Rule 4: All #define identifiers must be all CAPS.

- Documentation
  - Rule 1: Every .c file must begin with the following comment (shown with // but it can be created with /* */ as well):

```
// Programmer    : word1 word2
// Created       : word1 word2
//
// Purpose:
// word1 word2 word3 word4
//
// Modifications:
// Initial Date       Short Description
// <none>
```

Note the "purpose" is meant to permit multiple lines of comments, where there is at least one line of comment below the title "Purpose:" with a minimum of four tokens.

Note the "modifications" area is a way to track the modification history by other developers. This is an easier way to track other developer modifications that is lacking in code repositories. The "other" developer writes their initial, the date they made the change, and a short description as to what they changed and where. But this is assumed and not tested. What is tested is the word <none> if there have been no modifications then: exactly one token for "Initial"; exactly two or three tokens for "Date"; more than two tokens for "Short Description".

  - Rule 2: Every function must begin with the following comment (shown with // but it can be created with /* */ as well):

```
// ------------------------------------------------
// Programmer    : word1 word2    ← optional, for "other" developer
// Created       : word1 word2
//
// Purpose:
// word1 word2 word3 word4
//
// Parameters    : word1 word2
// Returns       : word1 word2
// Side-effects  : word1 word2
```

The horizontal line must be greater than 20 characters.

The "programmer" is not normally present. When it is present the tokens cannot be the same tokens as in the header comment programmer area.

The number of "words" shown are the minimum that must appear.

In the case of parameters, returns and side-effects then can be multi-lined comments.

- Rule 3: Every variable must have the following comment (shown with // but it can be created with /* */ as well):

```
type identifier-list;    // word1 word2
```

- Rule 4: Every struct/union must begin with at least one line of comment.

- Built-in test cases
  - Rule 1: Every .c file must have a testFILENAME() function.
  - Rule 2: The .c file with the main() function must have a test() function. In other words, this is an exception to rule 1. The function name is only the word test().
  - Rule 3: The test() function must call every testFILENAME() function at least once.
  - Rule 4: The main() function must have a call to the test() function within an if-statement. It is assumed (not tested) that there is a way to activate this test (maybe through a switch or a menu selection).

### Project Validation

To prove that the evaluator works, it will be tested with student assignment from COMP 206 and COMP 310 running on mimi. Once this has passed, the program will be recompiled on Windows and Mac and the same test will be repeated.

### Special Reporting

After the project is finished, it is important for the student to include in their final report ideas and suggestions on how their program might be improved by future students. This should include bug fixes, but also suggestions that will improve the usefulness and usability of the program.

### The Repository

I will create a McGill gitlab repository for this project. Please use this repository.

### About Making the Repo Public

If this project works out, we will create a public repo. Users will be able to download the code, while developers will be able to contribute to the project. It is very important that you write your code with this in mind:

- Code formatted professionally.
- Proper modular design
- Good comments
- Good documentation