C Code Style Rules

Context

The rules listed here are not absolute. They emerged from the typical practical context in which we write code. Good code should be:

- Easy to reuse. This saves a lot of time for debugging programs, you do not need to test new functions and there is less chance of making a mistake by adding new logic.

  Examples of violations:

  - global variables;
  - functions that do too much different work;
  - functions that simultaneously encode logic and use I / O.
- Easy to modify. Errors are frequent guests in programs, so difficult to modify means difficult to fix errors.

  Examples of violations:

  - global variables;
  - duplication of code (all duplicates need to be modified, how can I find them?)
- Easy to read. You and your coworkers are constantly reading the code.

  Examples of violations:

  - you mix `camelCase` and `snake_case`.
  - big functions
  - magic constants. You read the code and see the number 428, what does it mean in context? Better to use a named constant.
- Easy to test.

  Examples of violations:

  - big functions
  - global variables
  - functions that return their result through a pointer they receive in parameters.

```
void f(int* return_value ) { *return_value = 42; }
```

Sometimes these rules *can't be applied*. For example, code for microcontrollers is often compiled by low quality proprietary compilers that cannot optimize the code properly. Therefore, you must to make the code "less beautiful" in order, for example, to meet performance requirements, or to shrink the size of the program to fit in tiny memory.

But usually, these rules provide a reasonable criterion for judging whether you made a good architectural decision or not. If you break them, you should understand what you want to achieve this way.

Rules

We've prepared a list of rules for you to help you improve your code quality. When performing tasks in C, they must be observed.

1. Program structure

1. Functions must receive all the data they need through arguments.
2. Do not use mutable global variables (constants are Ok).
3. You can not mix the logic of calculations and I / O.
4. You cannot use `typedef` to define structures, except for structures from one field, which are analogous to `typedef`, but without implicit conversions (https://stepik.org/lesson/581681/step/5?unit=576402).
5. In the tasks, only the minimum required set of functions is indicated. You can add any number of helper functions for convenience.
6. Check the architecture. No decision will be made within a single file.
7. Write small functions, each does one thing.
8. For each function, ask yourself the question: what does it do? If the answer is longer than a few words, the function may need to be split into smaller functions.
9. Naming.

- Choose good names for functions and variables, as concise but informative as possible (this is not easy, good names do not immediately come to mind).
- Always name functions and variables consistently.

11. Make small functions even for those pieces of code that you don't plan to reuse. A small function has a name that quickly makes it clear what the code is used for.
12. Each function and variable should be accessed in the smallest possible part of the program. Consequently:

- Functions and globals that are only intended to be used in one module should be marked `static`

- No need to create index variables outside of loops (this was required in the C89 standard).
- It is useful to use opaque types.

13. File structures:

- Include guard is needed in header files need.
- Any header file `header.h` should be independent from others. This means that a file consisting of one line:

```
#include "header.h"
```

must be compiled.

- Sample structure of a header file:
    1. Includes
    2. Macros.
    3. Types definitions.
    4. Global variables.
    5. Functions.
- Sample structure of a `.c` file:
    1. Includes
    2. Macros (which should not be used in other files).
    3. Definitions of file-local types.
    4. Global variables.
    5. Static global variables.
    6. Functions.
    7. Static functions.
- In what order to include the header files for the `file.c` module?
    1. `file.h` (corresponded header file).
    2. Files from the standard library
    3. Other header files.

2. Types

1. Anything that can be marked `const` must be marked. An exception can be made for function arguments that are not pointers.

2. For indexes, use the `size_t` type.
3. Use only platform independent types such as `int64_t` or `int_fast64_t`. Numeric types marked `fast` are preferred because their existence is guaranteed on all platforms.
4. Use correct input and output specifiers.
5. Do not use the `PRI ...` specifiers for input instead of `SCN ...` and vice versa.

3. Using I / O

1. Error messages should be printed to `stderr`. General rule: the results of calculations are in `stdout`, information about how the calculations are performed in `stderr`.
2. You can not mix I / O and logic. One function counts, the other outputs.

4. Compilation

1. Don't forget to write a `Makefile`. It should allow, when changing one .c file, to rebuild part of the project without rebuilding everything else.

We will check your code using `gcc` and a `Makefile`.