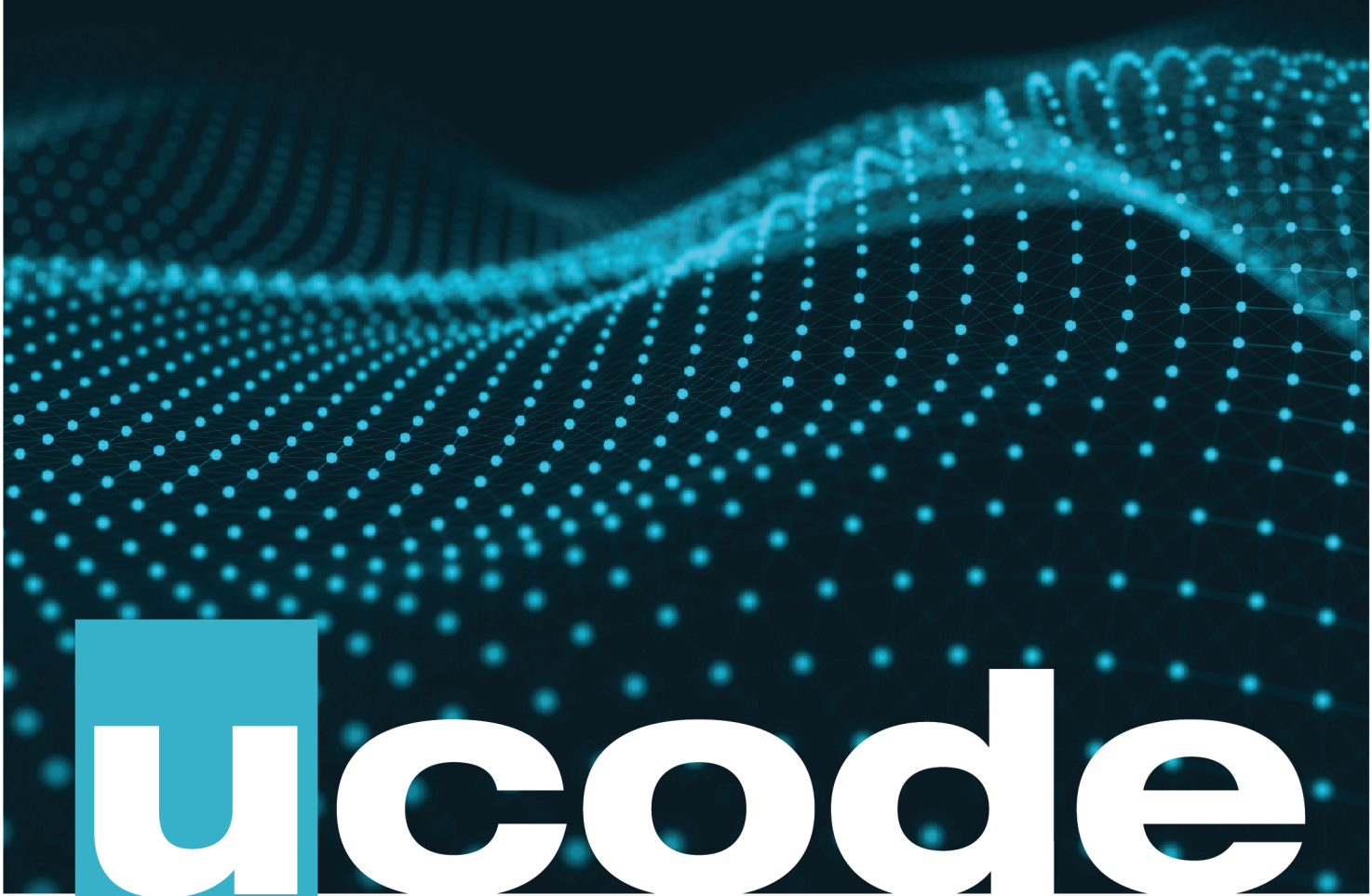
# The Auditor

## Style Guide

September 23, 2019



ucode

# Contents



ucode

# Introduction

## PREAMBULE

- One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of C code. Readability counts.
- A style guide is about consistency. Consistency with this style guide is important. Consistency within a challenge is more important. Consistency within one module or function is the most important.
- In particular: do not break backwards compatibility just to comply with the Auditor!

# The Auditor

## GLOBAL

- You should use the Auditor for every C challenges.
- You must use C11 standard for the C programming language.
- You should compile your file with next flags: `-Wall -Wextra -Werror -Wpedantic`.
- No compiler warnings!
- Objects (variables, functions, macros, types, files or directories) must have the most explicit or most mnemonic names as possible. Only 'counters' can be named to your liking.
- All that isn't allowed is forbidden.

## FORBIDDEN STUFF

- You're not allowed to use : `goto`, `do...while`.
- Any use of global variable are forbidden.

## LAY-OUT

- You cannot have more than 5 function-definitions in a .c file.
- All declarations must be at the top of a block (e.g. at the top of function), and must be separated by an empty line.
- One instruction per line.
- An empty line must be empty: no spaces or tabulations.
- No line should be longer than 79 characters, comments included. If this and the previous rule together don't give you enough room to code, your code is too complicated -- consider using subroutines.
- For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.
- There should be no space beside the semicolon, but single spaces on either side of the colon.

## HEADERS

- Preprocessor constants (or #define) you create must be used only for associate literal and constant values.
- All #define created to bypass the Auditor and/or obfuscate code are forbidden.
- You can use macros available in standard libraries, only if those ones are allowed in the scope of the given challenge.
- You must indent characters following `#if`, `#ifdef`, `#ifndef` or `#pragma once`.
- Multi-line macros are forbidden.

## FUNCTIONS

- All functions should be declared static unless they are to be part of a published interface.
- Function definition style: function name in column 1, outermost curly brackets in column 1, blank line after local variable declarations.

```
static int extra_ivars(t_mat_rx *rx, t_mat_qw *qw) {
    int soze = mx_bsize(rx);
    int ratz = mx_qwarz(qw);

    if (soze > ratz) {
    ...
    }
    else if (soze < ratz
            || mx_c4(soze, ratz)
            || mx_arch(ratz))
    {
    ...
    }
    else {
    ...
    }
    ...
    return 1;
}
```

- No functions should be longer than 20 lines. Functions own braces aren't counting.
- A function can take 4 named parameters maximum.
- `static inline` functions allowed.
- Designated initializers (especially nice for type declarations) allowed.
- You should follow intermingled declarations.
- Booleans ( `<stdbool.h>` ) are allowed.
- One single variable declaration per line.
- All function declarations and definitions must use full prototypes (i.e. specify the types of all arguments).

## SPACING

- Use 4 spaces per indentation level. Continuation lines should align wrapped elements either vertically using C implicit line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
var_three, var_four);

# Hanging indents should add a level.
foo = long_function_name(
var_one, var_two,
var_three, var_four);
```

- No line should end in whitespace. If you think you need significant trailing whitespace, think again -- somebody's editor might delete it as a matter of routine.
- You can’t declare more than 5 variables per bloc (per function too).
- You need to start a new line after each curly bracket or end of control structure.
- Each operator (binary or ternary) or operand must be separated by one - and only one - space.
- Each C keyword must be followed by a space, included keywords for types (such as int, char, float, etc.).
- The asterisks that go with pointers must be stuck to variable names.
- Code structure: one space between keywords like `if`, `for` and the following left paren; no spaces inside the paren; braces are required everywhere, even where C permits them to be omitted, but do not add them to code you are not otherwise modifying. All new C code requires braces. Braces should be formatted as shown:

```
if (mro != NULL) {
...
}
else {
...
}
```

- You should use curly braces if the condition in parens is more than one line. In the other case, curly brackets are optional.

```
if (mx_a != NULL
    && mx_b > 0) {
    mx_just_do_it();
}
...
if (mx_a != NULL && mx_b > 0)
    mx_just_do_it();
```

- The return statement should not get redundant parentheses:

```
Yes: return albatross;
No: return(albatross);
```

- Function call style: `foo(a, b, c)` -- no space before the open paren, no spaces inside the parens, no spaces before commas, one space after each comma.
- Always put spaces around assignment, Boolean and comparison operators.
- Breaking long lines: if you can, break after commas in the outermost argument list. Always indent continuation lines appropriately, e.g.:

```
has_you(knock_knock,
"Follow the white rabbit...",
mx->neo);
```

- Put blank lines around functions, structure definitions, and major sections inside functions.
- Use extra blank lines in functions, sparingly, to indicate logical sections. But no more than 1 additional blank lines per function.
- When you break a long expression at a binary operator, the operator goes at the start of the next line, and braces should be formatted from new line as shown. E.g.:

```
mx_s_user->age = mx_reallife_age(mx_s_user->name)
* mx_version(void)
/ mx_equation["eventuallity_of_anomaly"];

if (type->tp_dictoffset != 0 && base->tp_dictoffset == 0
    && type->tp_dictoffset == b_size
    || ((size_t)t_size == b_size + sizeof(mx *)))
{
    return 0;
}
```

- Avoid extraneous whitespace in the following situations:

  - Immediately inside parentheses, brackets or braces.

```
Yes: spam(ham[1], egg * 2)
No: spam( ham[ 1 ], eggs * 2 )
```

  - Between a trailing comma and a following close parentheses.

```
Yes: foo = {0,}
No: bar = {0, }
```

  - Immediately before a comma, semicolon, or colon:

```
Yes: foo = c > 0 ? 1 : 0;
No: if x == 4 : print x , y ; x , y = y , x
```

- Immediately before the open parenthesis that starts the argument list of a function call:

```
Yes: spam(1)
No: spam (1)
```

- Immediately before the open parenthesis that starts an indexing or slicing:

```
Yes: dct['key'] = lst[index]
No: dct ['key'] = lst [index]
```

- More than one space around an assignment (or other) operator to align it with another:

  Yes:

```
x = 1
y = 2
long_variable = 3
```

  No:

```
x  = 1
y =  2
long_variable  =  3
```

- Always surround these operators with a single space on either side:

  * arithmetic operators +, -, *, /, %
  * assignment operators =, +=, -=, *=, /=, %=
  * relational operators =, !=, >, <, >=, <=
  * comparisons ==, >, <, !=, >=, <=
  * bitwise operators &, |, ^  , >>, <<
  * logical operators &&, ||
  * ternary operator ?

  Yes:

```
i = i + 1
submitted += 1
x = !x * 2 - 1
hypot2 = x * !(x + y * y)
c = (a + b) * (a - b)
```

No:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x* ! ( x + y*y )
c = (a+b) * (a-b)
```

- Always surround these operators with a single space before the operator:

  * sizeof

  * pre increment and decrement operators ++, --

  * logical NOT !

  after the operator:

  * comma operator ,

  * post increment and decrement operators ++, --

  Yes:

```
x = !x * 2 - 1
hypot2 = x * !(x + y * y)
```

  No:

```
x = ! x * 2 - 1
hypot2 = x* ! ( x + y*y )
```

## NAMING CONVENTIONS

- Names that are visible to the user should follow conventions that reflect usage rather than implementation.
- Characters that aren't part of the standard ASCII table are forbidden.
- All identifiers (functions, macros, types, variables, etc) must be in English.
- Use a `mx_` prefix for public functions ; never for static functions. Specific groups of routines:
  - `s_` for structure's name
  - `t_` for typedef's name
  - `u_` for union's name
  - `e_` for enum's name
- Functions, variables, files and directories names use snake_case with lowercase and digits, like this: `mx_99_neo` , `mx_reloaded` , `mx_system_failure` .

- Macros should have a standard prefix and uppercase with underscores separating words, for example: `MX_ONE`, `MX_SUBWAY_STATION`.
- Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.

## COMMENTS

- Comments go before the code they describe.
- Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!
- Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).
- Block comments generally consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.
- You should use two spaces after a sentence-ending period in multi- sentence comments, except after the final sentence.
- Coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

## BLOCK COMMENTS

- Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. First line of a block comment starts with a `/*` and single space, next lines should start with asterisk and followed space at the same level of asterisk at first line, last line should be empty and looks like `*/` (asterisk at the same level of asterisk at first line).

```
int mx_get_foobang(int foo, int bang)
{
    mx_prepare_foobang(foo, bang);
    /*
    * Return a foobang
    * Optional plotz says to frobnicate the bizbaz first.
    */
    ...
}
```

- Paragraphs inside a block comment are separated by a line containing a single //.

## INLINE COMMENTS

- Use inline comments sparingly.

- An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a // and a single space.
- Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1; // Increment x
```

- But sometimes, this is useful:

```
x = x + 1; // Compensate for border
```

## MAKEFILE

- All source files you need to compile your challenge must be explicitly named in Makefile. Wildcards are forbidden.
- In the case of a challenge that calls a functions library, makefile must compile this library automatically.
- Makefile shouldn't relink. +
- It is forbidden to install or download additional files with the Makefile. You can use only mkdir, cp, mv, rm, clang, ar, touch and install_name_tool commands in the Makefile.
- The Makefile must manipulate only with the files under the project's root directory and its subdirectories. It is forbidden to interact (copy, move, etc.) with the files outside of your challenge directory.

## STANDARD MAKEFILE TARGETS

- Every Makefile should contain next targets:
  - all

```
Compile the entire program. This should be the default target.
```

  - install

```
Compile the program and copy the executables, libraries, and so
on to the project's root. If there is a simple test to verify that
a program is properly installed, this target should run that test.
```

  - uninstall

```
Delete all the installed files - the copies that the 'install'
targets create.
This rule should not modify the directories where compilation
is done, only the directories where files are installed.
```

- clean

```
	Delete all files in the current directory that are normally
	created by building the program. Also delete files in other
	directories if they are created by this makefile.
```

- reinstall

```
	Rebuilding the challenge.
	uninstall && install
```