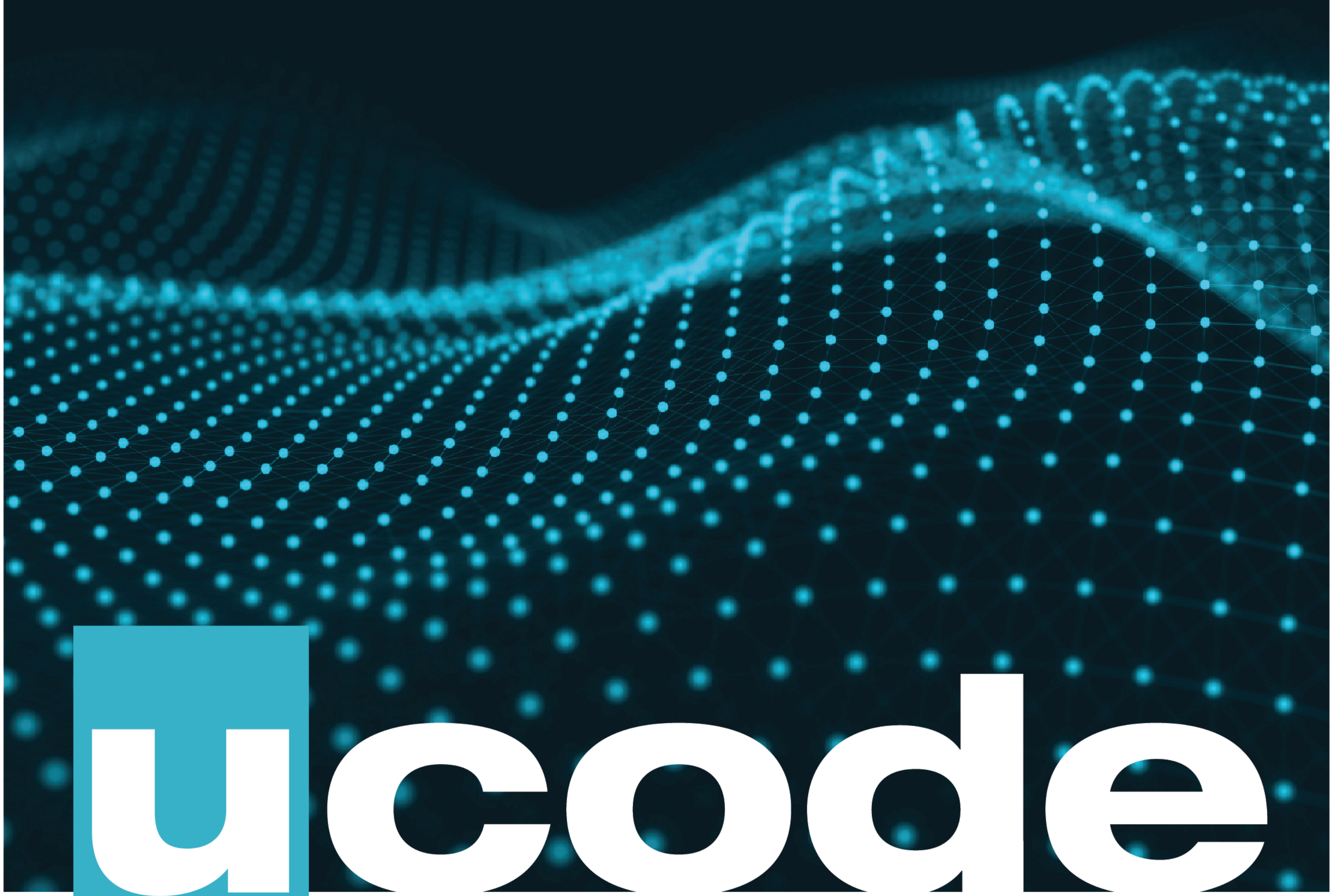
# Auditor

Style Guide

January 19, 2021



ucode

# Contents

## Preambule

This document is a code style guide for the ucode educational program. In the IT world, it is considered a best practice to follow a style guide. The quality of your code lies not only in how well it executes, but also in how clear, readable, and consistent it is.

A style guide presents a set of conventions. These conventions usually cover topics like file structure, naming, comments, indentation, code grouping, allowed tools, patterns, etc. A style guide is used to make the code within a certain project/company/school easy to follow for everyone involved (including the author).

Within ucode, it is vital to follow the Auditor for two key reasons. Firstly, you will help yourself and your peers by writing clear and concise code. And secondly, you will learn how to adapt to a set of rules and conventions, which is a helpful skill for your career.

# General rules

## GLOBAL

- Use the Auditor for all C challenges.
- Use C11 standard for code in the C programming language.
- Compile your files with the following flags: `-Wall -Wextra -Werror -Wpedantic`. Do not add more flags, unless stated otherwise in the story of a challenge.
- Name your objects (variables, functions, macros, types, files, directories, etc.) in the most explicit or mnemonic way possible. Only 'counters' can be up to you.

## FORBIDDEN STUFF

- `goto`
- `do...while`
- global variables

## LAY-OUT

- Maximum 5 function-definitions in a .c file.
- Forbidden to include .c files in other files.
- In .c files, you are allowed to include only you-defined header files. (unless stated otherwise).
- Preprocessor directives are forbidden, but you can use:
  - `#include` to include other files
  - `#define` to create your own macros
  - `#ifndef`, `#endif` and `#pragma once` to protect header files
- One instruction per line.
- In a `for` loop, there must be only one instruction per initialization-expression, condition-expression and loop-expression.
- Exactly one space after the semicolon in a `for` loop (but no spaces beside the semicolon in other cases).

```
Yes: for (int a; a < 3; a++)
No:  for (int a;a < 3;a++)

Yes: return;
No:  return ;
```

- Exactly one space on each side of a colon.
- An empty line must be empty: no spaces or tabulations.
- No line must be longer than 79 characters, comments included. If this rule does not 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 must be limited to 72 characters.

# Naming conventions

## GENERAL

- Names must 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, but never for static functions.
- Use the following prefixes:
  - `s_` for structure name
  - `t_` for typedef name
  - `u_` for union name
  - `e_` for enum name
- Functions, variables, files and directories names must be in snakecase, with lowercase and digits, like this: `mx_99_neo()`, `file_name.c`, `directory_name`, `system_failure_var`.
- Macros must have a standard prefix and uppercase with underscores separating words, for example: `MX_ONE`, `MX_SUBWAY_STATION`.
- Never use 'l' (lowercase 'L/l'), 'O' (uppercase 'O/o'), or 'I' (uppercase 'I/i') as single-character variable names.

# Headers

## GENERAL

- Header files must not contain function-definitions. Only prototypes are allowed.
- Protect your header files from double inclusion.
    - File with a name foo.h can be protected with `ifndef` in this way:

      ```
      #ifndef FOO_H
      #define FOO_H

      the file goes here

      #endif
      ```

    - Alternatively, a header file can be protected with the `#pragma once` preprocessor directive:

      ```
      #pragma once

      the file goes here
      ```

- You can declare a `typedef`, `struct`, `union`, `enum` only in a header file, unless stated otherwise.
- Place the opening curly bracket in a `struct`, `union` or `enum` on the same line as the previous keywords.
- When declaring a `struct`, `union` or `enum` with a `typedef`, all rules apply. Align the typedef's name with the struct/union/enum's name.

  ```
  typedef struct  s_structure_name {
      ...
  }               t_typedef_name;
  ```

- You are free to use anonymous `struct`, `union`, `enum`. If you use them with a `typedef`, there must be a single space between the closing parenthesis and the new type name.

  ```
  typedef struct {
      ...
  } t_typedef_name;
  ```

- Function prototypes and macro definitions are allowed only in header files.
- Multi-line macros are forbidden.
- Any `#define` created to bypass the Auditor and/or obfuscate code is forbidden.
- In terms of macros from standard libraries, only object-like macros are allowed.

# Functions

## GENERAL

- Declare all functions static, unless they are to be part of a published interface.
- Maximum 5 variable declarations per function.
- All declarations must be at the top of the block (e.g. at the top of the function or a loop).
- Function definition style: opening curly bracket at the last line of the function header.

```
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;
}
```

- If the function takes no arguments, the `void` keyword must be used as the argument.

```
Yes: void foo_bar(void);
No: void foo_bar();
```

- Maximum 20 lines per function. The function's own braces don't count.
- A function can take maximum 4 parameters.
- `static inline` functions are allowed.
- Designated initializers are allowed (particularly good for type declarations).
- Booleans ( `<stdbool.h>` ) are allowed.
- Maximum one variable declaration per line.
- All function declarations and definitions must use full prototypes (i.e. specify the types of all arguments).

ucode

# Spacing

## GENERAL

- Use 4 spaces per indentation level.
- Blank lines are forbidden, with the following exceptions:
  - blank lines around a function/struct/union/enum definition
  - one blank line after all the declarations inside a function
  - blank lines allowed inside header files (but no two blank lines in a row)
  - (optionally) one blank line to separate blocks on the same indentation level
    However, no blank line
    - if there is no another block after it
    - at the beggining of the block
    - between `if` and `else` blocks

```
void mx_func(void) {
    int var1 = 10;
    int var2 = 11;
                                //obligatory blank line after declarations
    if (var1) {
        if (...)
           function_call();
                                //optional blank line between blocks
        while (...) {
            ...
        }
                                //optional blank line, which separates block
        function_call();
        var1 = var2;
        another_function_call();
    }
    else {
        int var4 = 10;
                                //obligatory blank line after declarations
        ...
    }
}
```

- Continuation lines must be aligned with the opening delimeter (parenthesis, square bracket or brace), or using a hanging indent. When using a hanging indent there must be no arguments on the first line and further indentation must be used to clearly distinguish itself as a continuation line.

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

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

- 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);
```

- No whitespace at the end of a line.
- Start a new line after each curly bracket or end of control structure.
- Separate each operator (binary or ternary) or operand with exactly one space.
- Return value in function pointers must be followed by a space.

```
void mx_func(int a) {
    void (*func_pointer)(int) = &mx_func;
}
```

- No whitespace between a `sizeof` keyword and an opening parenthesis (if there is one).

```
Yes: sizeof(int);
Yes: sizeof something;
No:  sizeof (int);
```

- In cases of conversion and cast, no whitespace between the type and the operand.

```
Yes: int a = (int)3.5;
Yes: int b = (int)var_name;
Yes: int *c = (int*)ptr_name;
No:  int d = (int) var_name;
No:  int e = (int )var_name;
```

- No whitespace between a unary `*` operator and variable name.
  No whitespace between an adress-of unary `&` operator and variable name.
  No whitespace between multiple unary `*` and/or `&` operators.

```
void *b;
char a;

b = &a;
b = &*&*&a;
b = (void*)&a;
```

- Code structure:

- one space between the closing parenthesis and the opening curly bracket in `if`, `for`, `while`, `switch` blocks
- one space between keywords, such as `if`, `for`, `while`, `switch` and the following left parenthesis (no spaces inside the parentheses)
- braces must be formatted as shown:

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

- Curly brackets are optional if statement consists of only one instruction.

```
if (a != NULL
    && b > 0) {
    mx_just_do_it();
}
...
if (a != NULL && b > 0)
    mx_just_do_it();
...
while (a != NULL)
    mx_do_another_thing();
```

- Don't add redundant parentheses around the return statement.

```
Yes: return albatross;
No: return (albatross);
Yes: return a + b;
```

- Function call style: `foo(a, b, c)`
  no space before the opening parenthesis, no spaces inside the parentheses, no spaces before commas, one space after each comma.
- Always put spaces around assignment, Boolean, and comparison operators.
- When you break a long expression at a binary operator, the operator goes at the start of the next line. E.g.:

```
income = gross_wages
         + taxable_interest
         + (dividends - qualified_dividends)
         - ira_deduction
         - mx_magic_calculations()
         - student_loan_interest;
```

- Continuation of a condition begins on the next column after the opening bracket.

```
if (mx_func() == 1
    && mx_func2() == 1) {
    ...
}
for (int a = 0; a % 3 == 0
     && a < 10; a++) {
    ...
}
while (a > 1
       && a < 10) {
    ...
}
```

- No whitespace in the following cases:
  - immediately inside parentheses, square brackets or braces

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

  - between a trailing comma and a following close parentheses

    ```
    Yes: int foo[] = {0,}
    No: int bar[] = {0, }
    ```

  - immediately before a comma or a semicolon

    ```
    Yes: foo = mx_is_mid_num(10, 16, 4) == 1 ? 1 : 0;
    No: foo = mx_is_mid_num(10 , 16 , 4) == 1 ? 1 : 0 ;
    ```

  - immediately before an opening parenthesis that starts the argument list of a function call

    ```
    Yes: mx_spam(1)
    No: mx_spam (1)
    ```

  - immediately before an opening parenthesis that starts an indexing or slicing

    ```
    Yes: dct[0] = lst[index]
    No: dct [0] = 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
```

- around arrow and dot operators

```
Yes: s_struct->member
No: s_struct . member
```

- Always surround these operators with a single space on each side:
  - arithmetic operators `+`, `-`, `*`, `/`, `%`
  - assignment operators `=`, `+=`, `-=`, `*=`, `/=`, `%=`
  - relational operators `=`, `!=`, `>`, `<`, `>=`, `<=`
  - comparisons `==`, `>`, `<`, `!=`, `>=`, `<=`
  - bitwise operators `&`, `|`, `^`, `>>`, `<<`
  - bitwise assignment operators `&=`, `|=`, `^=`, `>>=`, `<<=`
  - logical operators `&&`, `||`
  - both elements of 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 if there is no parenthesis before them:

- pre increment and pre decrement operators `++`, `--`
- bitwise NOT `~`
- logical NOT `!`

after the operator:

- comma operator `,`
- post increment and post decrement operators `++`, `--` if there is no closing parenthesis or `;` after them

Yes:

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

No:

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

- Switch case usage:
  - the opening curly bracket goes on the same line as the `switch`
  - there must always be a `default` block
  - `case` labels go on the next identation level
  - `case` blocks do not have curly braces
  - each `case` (including the `default`) ends with a `break` or a `//fallthrough` comment
  - you may separate the cases with a newline
  - the cases' bodies go on the next indentation level after the `case` labels

```
switch (value) {
    case 1: //fallthrough
        mx_do_a();

    case 10:
        mx_do_b();
        break;

    case 11:
        mx_do_c();
        break;

    default:
        break;
}
```

# Comments

## GENERAL

- 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 must be complete sentences. The first word must 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.
- Write your comments in English.

## BLOCK COMMENTS

- Block comments generally apply to the code that follows them and are indented to the same level as that code. The first line of a block comment consists only of `/*`. Next lines must start with an asterisk followed by the space character. The asterisk must be at the same level as asterisk at the first line. The last line must consist only of `*/`, while the asterisk is at the same level as asterisk at the 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 must be separated by at least two spaces from the statement. They must start with a // and a single space.
- Don't add inline comments that are obvious (such comments are unnecessary and distracting). Don't do this:

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

- Sometimes, inline comments are useful

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

# Makefile

## GENERAL

- If a challenge calls a library of functions, your Makefile must compile this library automatically.
- Makefile must not relink.
- Build the project with `make` command in the root directory.
- Makefile must successfully build all targets.
- You can use only `mkdir`, `printf`, `cp`, `mv`, `rm`, `clang`, `ar`, `make`, `touch` and `install_name_tool` shell commands in the Makefile.
- Makefile functions are forbidden, except file name functions, text functions and wildcard function.
- A Makefile may 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.

## REQUIRED TARGETS

- Every Makefile must contain the following targets:
  - all

    ```
    Build the entire project. Default target.
    ```

  - NAME

    ```
    Target with the name of a library or an executable that you are building.
    Your Makefile must contain as many targets with names, as there are programs/libraries
    in your project.
    ```

    For example:

    ```
    CLIENT=client
    SERVER=server

    all: $(CLIENT) $(SERVER)

    $(CLIENT): ...
        ...
    ```

  - uninstall

    ```
    Delete all files and directories created during the building process.
    ```

  - clean

    ```
    Delete all files, excluding executables and libraries, that are normally
    created by this Makefile.
    ```

  - reinstall

    ```
    Rebuild the project.
    Delete all created files and build the project again.
    ```