General C Rules

July 13, 2021

Contents

1 Introduction

1.1 Standard language

1.1.1 Description

1. Project is developing in the well known C standard - C99.

1.1.2 Examples

1. ```
   /* Difference between GNU99 and C99 */

   /* Correct macro with standard C99 but danger */
   #define MIN(a,b) (a) > (b) ? (b) : (a)

   /* Incorrect. We do not support GNU99 */
   #define MIN(a, b) \
       ({ \
           typeof(a) _a = (a); \
           typeof(b) _b = (b); \
           (void)(&_a == &_b); \
           _a > _b ? _b : _a; \
       })

   /* Difference between C11 and C99 */

   /* Correct. Definition of strncpy from standard C99 */
   char *strncpy(char* restrict dst_p,
                 const char* restrict src_p,
                 size_t size);

   /* Incorrect. Definition of strncpy from standard C11. In standard C99
       we cannot ensure that array has at least one char */
   char *strncpy(char target[static 1],
                 char const source[static 1],
                 size_t size);
   ```

1.1.3 Reason

1. GNU99 is a great C standard with compiler features. By using this standard object oriented code is simpler to understand, because instead of C tricks we can use attributes and new keywords. Unfortunately this standard is supported only by GCC. Because we want to use clang static analyzer we cannot use GNU99. The last well known standard supported by all C compilers is ISO C99. That's why we chose C99.

1.2 Files

1.2.1 Description

1. All header files should be named as <class_name>.h, source files should be named in the same way with .c extension.

2. Header must have include guardian using
   #ifndef FILE_NAME_WITH_CAPITAL_LETTERS_H.

3. Always header file must have file doxygen documentation where destiny of file should be explained.
   ```

4. Do not use "" to import files (#include "file.h"). Always you should use <> operators (#include <file.h>). Makefile should take care of proper importing and linking files.

1.2.2 Examples

1. ```
   /* In this example we would like to implement pair - container. These
      names are correct */

   pair.h /* header file */
   pair.c /* source file */
   ```

2. ```
   /* Correct added file guardians */
   #ifndef PAIR_H
   #define PAIR_H

   /* some code */

   #endif /* PAIR_H */

   /* Incorrect */
   #pragma once

   /* some code */
   ```

3. ```
   /* Correct beginning of the file with doxygen rules */
   /** @file */
   ```

4. ```
   #ifndef PAIR_H
   #define PAIR_H

   /* Correct */
   #include <array.h>

   /* in Makefile */
   $(INCLUDES) += -I path_to_array.h

   /* Incorrect */
   #include "array.h"

   #endif /* PAIR_H */
   ```

1.2.3 Reason

1. This name convention is related to OOP in C file naming convention.
2. #pragma once is a compiler specific keyword. Since we want to be compatible with all C compilers, basic mechanism should be used.
3. File documentation with name convention tells user what is the file content and when user should use this file (include to project as well)
4. Using improper operators can cause linker error. Good build system should set compiler flag to header file's path before compiler will be run. That's why only <> operators should be used. If error occurs during compiling then build system should be fixed.

2 Style

Clang-format is a tool to format C code according to a set of rules and heuristics. Like most tools, it is not perfect nor covers every single case, but it is good enough to be helpful. Clang-format is specially useful when moving code around and aligning/sorting. Help you follow the coding style rules, specially useful for those new in this project or working at the same time in several projects with different coding styles.

2.1 Indentation

Only spaces should be used. Indentation should be set to 4 spaces.

2.2 Breaking long lines and strings

Long line is defined as line with more than 140. Every long line should be split into more lines using some rules.

2.2.1 Description

1. Break line after operator.
2. Do not break structure chain. If structure chain will be longer than 140 chars, split it and rewrite it as pointers.
3. Breaking function parameters should be aligned to each other.

2.2.2 Examples

1. ```
   /* Correct. Break after an operator */
   if (condition &&
       condition)
   {
       /* some code */
   }

   /* Incorrect if line with conditions is longer than 140 characters */
   if (condition && condition)
   {
       /* some code */
   }
   ```

2. ```
   /* Correct */
   a_p->b_p->c.d_p->g.f;

   /* Correct */
   ptr_p = a_p->b_p;
   ptr_p->c.d_p->g.f;

   /* Incorrect */
   a_p->b_p->c.
       d_p->g.f;
   ```

3. ```
   /* Correct. Break after comma. Function parameters should be aligned
      */
   void foo(arg1,
            arg2,
   ```
   ```

```
        arg3);

/* Incorrect if argument list is longer than 140 characters */
void foo(arg1, arg2, arg3);
```

2.2.3 Reasons

1. This style makes code more readable.
2. Broken structure chain makes code hard to read and understand.
3. Squashing code should be avoided. Aligned function arguments makes easier to analyze.

2.3 Placing braces and spaces

2.3.1 Description

1. Always after C keyword (if, switch, case, for, do, while), before and after operator (+, -, *, /), before comma space should be inserted. Exception is sizeof..
2. Structure operators (->, .) and array operator([ ]) should be used without spaces between them.
3. Braces should not be avoided in 1-line statements.
4. Braces should be always used in switch case.

2.3.2 Examples

1. ```
   /* Correct */
   for (size_t i = 0; i < ARRAY_SIZE(array); ++i)
   {
       printf("%4d", array[i]);
   }

   /* Incorrect */
   for(size_t i = 0; i < ARRAY_SIZE(array); ++i)
   {
       printf("%4d", array[i]);
   }

   /* Correct */
   if (condition)
   {
       do_this();
   }

   /* Incorrect */
   if(condition)
   {
       do_this();
   }

   /* Correct */
   a + b      /* addition operator */
   a - b      /* subtraction operator */
   a / b      /* division operator */
   a % b      /* modulus operator */
   ```
   ```

```
a ^ b      /* bitwise xor operator */
a & b      /* bitwise and operator */
a | b      /* bitwise or operator */
a < b      /* less than operator */
a <= b     /* less than or equal to operator */
a > b      /* greater than operator */
a >= b     /* greater than or equal to operator */
a == b     /* equal to operator */
a != b     /* not equal to operator */
a && b     /* logical and operator */
a || b     /* logical or operator */
a << b     /* shift left operator */
a >> b     /* shift right operator */
a = b      /* assignment operator */
a ? b : c /* ternary operator */
```

2. ```
   /* Correct */
   matrix[5][5] = 12;

   /* Incorrect */
   matrix [5] [5] = 12;

   /* Correct */
   list_p->head_p = NULL;

   /* Incorrect */
   list_p -> head_p = NULL;

   /* Correct */
   t[]        /* array operator */
   t()        /* function call operator*/
   t.a        /* member operator */
   t->a       /* arrow operator */
   *t         /* dereference operator */
   &t         /* address operator */
   ~t         /* bitwise negation operator (unary) */
   !t         /* logic negation operator */
   ++t        /* preincrementation operator */
   t++        /* postincrementation operator */
   --t        /* predecrementation operator */
   t--        /* postdecrementation operator */
   sizeof(t) /* sizeof operator */
   ```

3. ```
   /* Correct */
   for (size_t i = 0; i < ARRAY_SIZE(array); ++i)
   {
       printf("%4d", array[i]);
   }

   /* Incorrect */
   for (size_t i = 0; i < ARRAY_SIZE(array); ++i)
       printf("%4d", array[i]);

   /* Correct */
   ```

```
if (condition)
{
    do_this();
}

/* Incorrect */
if (condition)
    do_this();
```

4. ```
   /* Correct */
   switch (val)
   {
       case 1:
       {
           do_this();
           calculate();
           break;
       }
       case 2:
       {
           /* some code */
       }
       default:
       {
           ERROR("something bad has happened\n");
           break;
       }
   }

   /* Incorrect */
   switch (val)
   {
       case 1:
           do_this();
           calculate();
           break;
       case 2:
           foo();
           break;
       default:
           ERROR("something bad has happened\n");
           break;
   }
   ```

2.3.3 Reasons

1. It is easier to read special language keywords and operator with space before them.
2. It looks much better when array and structure operators are one sequence of characters.
3. Code has to be readable that is why braces cannot be avoided in one line statements. Instead of it, ternary operator can be use.
4. It allows to dismiss mistakes and allocate memory on stack if it is needed.
   ```

2.4 White characters

All trailing whitespaces should be deleted before commit.

2.5 Commenting

Do not use // as comment, even if comment has only 1 line. This is cpp specific. C comment operator should be used: /* */. To comment documentation, doxygen comment operator should be used: /** */

2.6 Casting

2.6.1 Description

1. Pointer to void should be never cast to and from
2. All compatible and incompatible types should be cast
3. Not used function parameters should be cast to void
4. Not used returned value where value describes error of function should be cast to void
5. Not used returned value from system functions should be always cast to void

2.6.2 Examples

1. ```
   /* Correct */
   int c = 0;
   void* vptr_p = &c;

   /* Incorrect */
   int b = 5;
   void* vptr_p = (void *)&b;

   /* Incorrect */
   int a = 3;
   int* ptr_p = &a;
   void* vptr_p = (void *)ptr;
   ```

2. ```
   /* Correct */
   char a = '0';
   int8_t b = (int8_t)a;

   /* Incorrect */
   char a = '0';
   int8_t b = a;

   /* Correct */
   int c = 2;
   long d = (long)c;

   /* Correct */
   int c = 2;
   long d = c;
   ```

3. ```
   /* Correct */
   void foo(int* a_p, int b, int c)
   {
       /* c in no longer needed */
       (void)c;

       /* some code */
   }

   /* Incorrect */
   void foo(int* a_p, int b, int c)
   {
       /* c in no longer needed but not case to void */

       /* some code */
   }
   ```

4. ```
   /* Correct */
   (void)strncpy(dst, src, size_of);

   /* Incorrect because printf returns number of
   characters , not error */
   (void)printf("Hello world\n") ;
   ```

5. ```
   /* Correct because write makes syscall and returns number of bytes
      written, which can be used to check success or -1 if failure */
   (void)write(fd, &buf, strlen(buf));

   /* Incorrect */
   write(fd, &buf, strlen(buf));
   ```

2.6.3 Reason

1. This implicit conversion is provided by compiler.
2. It makes code longer but easier to read.
3. This information tells clearly to programmer that input argument is no longer needed.
4. As above
5. As above

2.7 Allocating memory

2.7.1 Descripton

1. Do not use malloc to allocate memory. Instead of it, use calloc.
2. Structure allocated on stack should be always reset.
3. Do not use alloca to dynamically allocate memory on stack. C99 supports VLA and VLA should be used instead.
   ```

2.7.2 Examples

1. ```
   /* Correct */
   int64_t* ptr_p = calloc(5, sizeof(*ptr_p));

   /* Incorrect */
   int64_t* ptr2_p = malloc(sizeof(*ptr2_p) * 5);
   ```

2. ```
   /* Correct. Always initialize arrays on stack */
   void foo(void)
   {
       uint32_t array[10] = {0};

       /* some code */
   }

   /* Incorrect */
   void foo(void)
   {
       uint32_t array[10]; /* Not initialized array */

       /* some code */
   }

   /* Correct. Always memset VLA because VLA cannot be initialize as
      arrays of constant known size */
   void vla(size_t size)
   {
       int vla_array[size];
       (void)memset(&vla_array[0],
                    0,
                    sizeof(vla_array));

       /* some code */
   }

   /* Incorrect */
   void vla(size_t size)
   {
       int vla_array[size]; /* Not initialized VLA */

       /* some code */
   }
   ```

3. ```
   /* Correct */
   void foo(const size_t size)
   {
       /* Do not forget to memset VLA */
       int64_t array[size];
       (void)memset(&vla_array[0],
                    0,
                    sizeof(vla_array));

       /* some code */
   ```
   ```

```
}

/* Incorrect. Do not use alloca to allocating memory on stack.
   Instead of it, use VLA, which is supported by C99 */
void foo(const size_t size)
{
    int64_t* ptr_p = alloca(sizeof(*ptr_p) * size);

    /* some code */
}
```

2.7.3 Reason

1. Always use calloc which set memory to zero. This would avoid unnecessary mistakes in future.
2. Tis is often mistake. Dereference of memory allocated on stack, which has not been reseted. Always reset memory allocated on stack.
3. Try to avoid obsolete alloca for dynamic allocations on stack. Instead of it, use feature from C99, which is variable length array.

3 Naming

3.1 Descripton

1. Snake case should be used in every names (files, variables, functions, macros, macros like functions).
2. Function name should be started with class prefix.
3. Pointer should have suffix _p, double pointer _pp etc. (Array is not a pointer so should not have suffix).
4. Name should show programmer what function or variable do. But also name should be as short as possible.
5. New type of complex structure should start with capital letter, new types of fundamental types should end with _t prefix.

3.2 Examples

1. ```
   /* Correct */

   void foo(int val1, int val2, int val3);
   #define foo(val1, val2) foo(val1, val2, 0)

   static inline Rbt_node* rbt_min_node(const Rbt_node* node_p);

   static Rbt_node* sentinel_p = &sentinel_node;

   /* Incorrect */
   #define Foo(val1, val2) foo(val1, val2, 0)

   static inline RbtNode* __minNodeOfRbt(const RbtNode* node_p);

   static RbtNode* sentinel_p = &sentinelNode;
   ```

2. ```
   /* Correct */
   void* pair_create(int val1, int val2);

   /* Incorrect because pair is class name. We cannot do functions that
      begin with create, get, make etc. Instead of it we add prefix which
       contains class name */
   void* create_pair(int val1, int val2);
   ```

3. ```
   /* Correct */
   int* ptr_p;
   int array[10];

   /* Incorrect */
   int* ptr;
   int array_p[10];
   ```

4. ```
   /* Correct */
   int t[10];
   for (int i = 0; i < 10; ++i)
   {
   ```

```
    printf("%4d", t[i]);
}

#define SWAP(a, b, type) \
    do { \
        type tmp = a; \
        a = b; \
        b = tmp; \
    } while (0)

List* list = list_create();
List_node* head = list_get_head(list);

/* Incorrect */
int t[10];
for (int t_array_iterator = 0;
     t_array_iterator < 10;
     ++t_array_iterator)
{
    printf("%4d", t[t_array_iterator]);
}

#define SWAP(a, b, type) \
    do { \
        type new_temporary_variable_on_stack = a; \
        a = b; \
        b =  new_temporary_variable_on_stack; \
    } while (0)

List* l = list_create();
List_node* h = list_get_head(l);
```

5. ```
/* Correct */
typedef struct Pair
{
    int first;
    int second;
} Pair;

typedef uint32_t u32_t

typedef enum option
{
    OPTION_1,
    OPTION_2,
} option_t

/* Incorrect */
typedef struct Pair
{
    int first;
    int second;
} pair_t;
```
```

```
typedef uint32_t u32

typedef enum option
{
    OPTION_1,
    OPTION_2,
} Option
```

3.3 Reason

1. Snake case has been choosed because is easy to read and looks well.
2. Functions with class prefix tells clearly programmer where this function belongs.
3. Pointers are using another operators, so it's better to know if variable is a pointer or not.
4. Do not use cute and fancy names like this_variable_is_temporary_counter. A C programmer would call that variable tmp, which is much easier to write, and not the least more difficult to understand.
5. Custom types are really nice, but if we cannot find out which type is complex type or fundamental type we cannot optimize code also. When we will see type starting with capital letter we will know that this is structure or union and it's better to pass it via pointer. Also suffix _t tells us that this is only alias for another fundamental type.

4 Types

4.1 Description

1. When the variable may contains specific values (0 - off, 1 - on) new type should be created.
2. Never typedef arrays and pointers.
3. Enum, structure, union should create new types.
4. Always use type depending on your job.

4.2 Examples

1. ```
   /* Correct. This type tells exactly what this variable represents */
   typedef enum parser_state_t
   {
       PARSER_STATE_WORKING,
       PARSER_STATE_STOPPED,
   } parser_state_t;

   parser_state_t parser_state = PARSER_STATE_STOPPED;

   parser_state = parse(...);

   if (parser_state == PARSER_STATE_WORKING)
   {
       do_this();
   }
   else
   {
       do_this();
   }

   /* Incorrect */
   uint8_t parser_state = 0;

   parser_state = parse(...);

   if (parser_state == 0)
   {
       do_this();
   }
   else
   {
       do_this();
   }
   ```

2. ```
   /* Correct. Typedef structures and basic types */
   typedef uint8_t byte_t;
   typedef Rbt Rbt;

   /* Incorrect. Never do this ! */
   typedef void* void_ptr;
   typedef int int_array[CONSTANT];
   ```

3. ```c
   /* Correct */
   typedef enum option
   {
       OPTION_1,
       OPTION_2,
   } option_t;

   typedef struct Pair
   {
       int first;
       int second;
   } Pair;

   /* Incorrect */
   enum option
   {
       OPTION_1,
       OPTION_2,
   } option;

   struct Pair
   {
       int first;
       int second;
   } Pair;
   ```

4. ```c
   /* Correct, if you need to represent byte use uint8_t from stdint.h */
   typedef uint8_t byte_t;
   typedef char byte_t; /* Possible */

   /* Incorrect */
   typedef uint8_t byte;
   typedef char BYTE;


   /* Correct, if you need to represent size or length use size_t from
      stddef.h */
   const size_t array_size = sizeof(array) / sizeof(array[0]);

   /* Incorrect */
   const uint64_t array_size = sizeof(array) / sizeof(array[0]);


   /* Correct, if you need to return size or length from function, but
      '-1' is error message use ssize_t from sys/types.h */
   ssize_t darray_get_length(...);

   /* Incorrect */
   int64_t darray_get_length(...);


   /* Correct, if you need to subtract two pointer, use ptrdiff_t from <
      stddef.h> */
   int numbers[] = { /* lots of members */ };
   ```

```
int* p1_p = &numbers[18];
int* p2_p = &numbers[23];

const ptrdiff_t diff = p2 - p1;

/* Incorrect */
const int64_t diff = p2 - p1;
```

4.3 Reasons

1. It will make code more readable for other programmers.
2. This practices makes code really hard to understand.
3. By the new type we can represent new abstraction. It also helps to keep cohesion in code.
4. For representing:
   - unsigned numbers - use unsigned types.
   - signed numbers - use signed types.
   - unsigned numbers with possible error code (-1) - use ssize_t.
   - floating point - use double, long double types with proper precision.
   - etc.

5 Variables

5.1 Description

1. Always minimize the scope of variables and constants.
2. Always prefer function local variables than file local variables and file local variables than global variables.
3. Do not initialize with zeros file variables and global variables.
4. Do not initialize enum first value with 0.
5. Const key word should be used when variable is read only.
6. When defining an array variable, do not specify its size if is it possible.

5.2 Examples

1. ```
   /* Correct */
   void foo(void)
   {
       for (size_t i = 0; i < CONSTANT; ++i)
       {
           /* some code */
       }
   }

   /* Incorrect */
   void foo(void)
   {
       size_t i;

       for (i = 0; i < CONSTANT; ++i)
       {
           /* some code */
       }
   }
   ```

2. ```
   /* Correct */
   int f(int a)
   {
       static int carry;
       return a + (++carry);
   }

   /* Incorrect */
   static int carry;

   int f(int a)
   {
       return a + (++carry);
   }
   ```

3. ```
   /* Correct */
   typedef enum option
   {
       OPTION_1,
       OPTION_2,
   } option_t;

   typedef enum option
   {
       OPTION_1 = 1 << 0,
       OPTION_2 = 1 << 1,
   } option_t

   /* Incorrect */
   typedef enum option
   {
       OPTION_1 = 0,
       OPTION_2,
   } option_t;
   ```

4. ```
   /* Correct */
   parser_status_t global_status;
   static parser_status_t parser_status[CONSTANT];

   /* Incorrect. File and global variables will be initialize by zero by
      compiler. Do not take a compiler job */
   parser_status_t global_status = 0;
   static parser_status_t parser_status[CONSTANT] = {0};
   ```

5. ```
   /* Correct */
   const size_t list_length = list_get_length(list_p);

   if (list_length > CONSTANT)
   {
       do_this();
   }

   /* Incorrect. If length of list is read only, variable must be const.
      It is only information for programmer (Of course compiler will
      throw out const keyword) */
   size_t list_length = list_get_length(list_p);

   if (list_length > CONSTANT)
   {
       do_this();
   }
   ```

6. ```
   /* Correct */
   int64_t array[] = { 1, 2, 4, 8, 16, 32 };

   /* Incorrect. If possible do not create another define for
   array size */
   int64_t array[CONSTANT] = { 1, 2, 4, 8, 16, 32 };
   ```

```c
/* Correct. Remember, string with custom defined termination should be
    defined without size */
const char str[] = "Hello world\n";

/* Incorrect. What about '\0' ? */
const char str[13] = "Hello world\n";
```

5.3 Reasons

1. Smaller scope of variables makes code much easier to maintain.
2. Code with smaller number of static or global variables will be easier to maintain and develop.
3. ISO C99 tells that it is compiler job.
4. As above.
5. Const keyword should tells programmer that this variable is read-only.
6. If declare and initialize array from brace-enclosed list, do not add size of array, it is compiler job.

6 Functions

6.1 Description

1. Always prefer programming by contracts. Try to split function for preconditions, asserts for selected operations and post conditions.
2. Use const keyword for arguments passed by value and pointers (immutable pointer, data and pointer and data). Remember, add const keyword only in definition, not in declaration. We want be in accordance with MISC.
3. Never pass/return huge structures to/from a function by value. Always avoid unnecessary copying of data. Pointer to the structure has to be passed/returned to/from function.
4. Always prefer to write functions not larger than 100 lines (comments included) and input arguments not exceed 6.
5. Function definition must have a declaration. Exported, static, inline and static inline functions.
6. Put function body logic in following order: input, modify, output.
7. Put function parameters in following order: input, output.

6.2 Examples

1. ```
   /* Programming by contract helps to avoid a lot of errors */
   int foo(int input)
   {
       int output = 0;

       /* Precondition. Check input. */
       /* some code */

       /* Assertions for proper operations */
       /* some code */

       /* Postcondition. Check returned output */
       /* some code */

       return output;
   }
   ```

2. ```
   /* Correct declaration */
   void foo(size_t val);

   /* Incorrect declaration */
   void foo(const size_t val);

   /* Correct definiton. Immutable value at input */
   void foo(const size_t val)
   {
       /* some code */
   }


   /* Correct declaration */
   void foo(void* ptr_p)
   ```

```
/* Incorrect declaration */
void foo(const void* ptr_p)

/* Correct definiton. Mutable pointer to immutable data */
void foo(const void* ptr_p)
{
    /* some code */
}


/* Correct declaration */
void foo(void* ptr_p)

/* Incorrect declaration */
void foo(void* const ptr_p)

/* Correct definiton. Immutable pointer to mutable data */
void foo(void* const ptr_p)
{
    /* some code */
}


/* Correct declaration */
void foo(void* ptr_p)

/* Incorrect declaration */
void foo(const void* const ptr_p)

/* Correct definiton. Immutable pointer to immutable data */
void foo(const void* const ptr_p)
{
    /* some code */
}
```

3. ```
   /* Correct. Return huge structure by pointer */
   void* foo(void)
   {
       Huge_structure* ptr_p = calloc(1, sizeof(*ptr_p);
       process(ptr_p, data);

       return ptr_p;
   }

   /* Incorrect because returned huge structure by value */
   Huge_structure foo(void)
   {
       Huge_structure result = {0};
       process(&result);

       return result;
   }
   ```
   ```

```c
/* Correct. Pass huge structure to function by pointer */
void* foo(Huge_structure* ptr_p)
{
    /* some code */
}

/* Incorrect */
void* foo(Huge_structure huge_structure)
{
    /* some code */
}
```

4. ```c
/* Correct, code has been splitted into a few functions, and main
   function only calls another function */
void foo(void)
{
    Huge_structure* ptr_p = huge_structure_create();
    huge_structure_func1(ptr_p);
    huge_structure_func2(ptr_p);
    huge_structure_func3(ptr_p);
    huge_structure_destroy(ptr_p);
}

/* Neither correct nor incorrect. It is not always possible to write
   function to 100 lines and 6 arguments. But if possible, design
   function as good as can */
void foo(void)
{
    Huge_structure* ptr_p;
    /* code connected with allocation */
    /* code connected with func1 */
    /* code connected with func2 */
    /* code connected with func3 */
    /* code connected with deallocation */
}
```

5. ```c
/* Correct */
static inline void foo1(void);
inline void foo2(void);
void foo3(void);

static inline void foo1(void)
{
    /* some code */
}

inline void foo2(void)
{
    /* some code */
}

void foo3(void)
{
```

```
    /* some code */
}

/* Incorrect. There are no declarations. */
static inline void foo1(void)
{
    /* some code */
}

inline void foo2(void)
{
    /* some code */
}

void foo3(void)
{
    /* some code */
}
```

6. ```
   /* Correct */
   void foo(void)
   {
       /* create necessary variables */
       /* some code */

       /* process, modify */
       /* some code */

       /* output */
       /* some code */
   }

   /* Incorrect */
   void foo(void)
   {
       /* create necessary variables */
       /* some code */

       /* process, modify */
       /* some code */

       /* get output */
       /* some code */

       /* process output again */
       /* some code */

       /* output */
       /* some code */
   }
   ```

7. ```
   /* Correct */
   void foo(void* input1_p, void* input2_p, void* output_p);
   ```

```
/* Incorrect */
void foo(void* output_p, void* input1_p, void* input2_p);
void foo(void* input1_p, void* output_p, void* input2_p);
```

6.3 Reasons

1. If-statement should be added to exported functions, asserts to static functions and function with critical performance, when every cycle counts.
2. Const keyword should only tell programmer what is read-only in function, which helps to analyze and understand function.
3. Huge structures, passed/returned to/from functions will be copying by stack. Better idea would be coping address of pointer. Copying by stack may lead to stack overflow.
4. Large code = small readability. Function with many responsibilities spoils rule SOLID from OOP (section 8).
5. With declared function we will avoid linker errors. We do not need to worry about functions order.
6. This order make functions easier to read, understand and maintain.
7. This order make functions easier to analyze.

7 Macros

7.1 Description

1. Names of macros defining ICE (Integer Constant Expression) and labels in enums are always capitalized.
2. Capitalized macro names are appreciated for functionlike macros. Snake case names are only appreciated for functionlike macros which overlapping functions.
3. Generally, inline functions are preferable to macros resembling functions.
4. Macros with multiple statements should be enclosed in a do - while block.
5. Macros should not affect control flow. If do, describe it with details in code.
6. Whenever possible, use an enum definition instead of #define values.
7. Always put parentheses around the macro parameters and whole macro. But do not add parentheses around ICE which are not expressions.

7.2 Examples

1. ```
   /* Correct */
   #define CONSTANT 0x12345

   /* Incorrect */
   #define constant 0x12345
   ```

2. ```
   /* Correct. This is ICE */
   #define CONSTANT 0x12345

   /* Correct. This is functionlike macro */
   #define ASSIGN(dst_p, src_p, size_of) \
       do { \
           /* some code */ \
       } while (0)

   /* Incorrect. This functionlike macro not overlapping other function
      */
   #define assign(dst_p, src_p, size_of) \
       do { \
           /* some code */ \
       } while (0)

   /* Correct. This functionlike macro overlapping function pair_create.
      */
   #define pair_create() \
       do { \
           /* some code */ \
       } while (0)
   ```

3. ```
   /* Correct. This code need to print LINE in code, so need to be pasted
       into code directly */
   #define PRINT_LINE() printf("%d", __LINE__)

   /* Correct. This code need to works for all fundamental types */
   ```
   ```

```c
#define MIN(a, b) ((a) < (b) ? (a) : (b))

/* Correct. This is only wrapper, so should be inline */
static inline foo_wrapper(void)
{
    foo_func1();
    foo_func2();
}

/* Incorrect. This cannot be inline as functions, because this will be
    print LINE from function code instead of caller code line */
static inline print_line(void)
{
    printf("%d", __LINE__);
}

/* Incorrect. This code need as many functions as types */
static inline min_int(const int a, const int b)
{
    return a < b ? a : b;
}

static inline min_double(const double a, const double b)
{
    return a < b ? a : b;
}

/* the same for other types */

/* Incorrect. There is no advantages to use this wrapper as a macro */
#define FOO_WRAPPER(void) \
do { \
    foo_func1(); \
    foo_func2(); \
} while (0)
```

4. ```c
   /* Correct */
   #define FOO(ptr_p) \
       do { \
           free(ptr); \
           ptr_p = NULL; \
       } while (0)

   /* Incorrect */
   #define FOO(ptr_p) \
       free(ptr); \
       ptr_p = NULL; \
   ```

5. ```c
   /* Correct. This functionlike macro print error message and ending
      current process */
   #define ERROR(msg, val) \
       do { \
           fprintf(stderr, "%s.\t\t FILE=%s LINE=%d\n", \
                   msg, __FILENAME__,  __LINE__); \
   ```

```
        return val; \
    } while (0)

/* Incorrect. This functionlike macro affect control flow but affected
    control flow was not described */
#define FOO(x) \
    do { \
        if (blah(x) < 0) \
        { \
            return -EBUGGERED; \
        } \
    } while (0)
```

6. ```
   /* Correct */
   typedef enum parser_state_t
   {
       PARSER_STATE_WORKING,
       PARSER_STATE_STOPPED,
   } parser_state_t;

   /* Incorrect */
   #define PARSER_STATE_WORKING 0
   #define PARSER_STATE_WORKING 1
   ```

7. ```
   /* Correct */
   #define power(x) ((x) * (x))

   /* Incorrect */
   #define power(x) (x * x)

   /* Incorrect */
   #define power(x) (x) * (x)

   /* Incorrect. This is ICE */
   #define CONSTANT (NUMBER)
   ```

7.3 Reasons

1. Capitalized names tells only this is functionlike macro, define or enum. By keeping this order, code would be easier to read for static analyze.
2. In C99 overlapping is only possible by writing functionlike macro with the same name. This is why we allows for snake case for functionlike macros. In other ways, always prefer capitalized names for functionlike macros and ICE.
3. If you need generic function, always choose functionlike macros instead of writing many inline functions for specified types. If not, choose inline function which allows to check type at compile time.
4. Even if we have curly brackets, we will use do - while loop to protect against undesirable errors.
5. Usually we need wrappers for few operations e.g. write message and return. It does not make sens to copy-paste this operations. Better idea would be write functionlike macro. Inline function will not work because we return from incorrect function.
6. Enum has adjusted type and proper range for assign. Differ values are errors at compile time and this is why we do not want substitute enum by #define.

7. This helps to avoid many preprocessor mistakes.

8 Object oriented programming

The C programming language does not have native support for object oriented programming, but it is possible and indicate to write code object oriented. It makes code more readable and easier to maintain.
Some useful tricks and tips can be found here: OOP in C.

8.1 Description

1. Always follow OOP naming convection
   - class_create - allocating memory for object and setting fields if needed
   - class_init - only setting fields
   - class_destroy - deallocating memory
   - class_deinit - setting object fields to 0
2. Always hide structure members from users using forward declaration. This will force to use getters and setters instead of directly access to members.
3. Try to follow KISS principle (Keep it simple, stupid). Always prefer code simple, without C tricks, with straight logic.
4. Try to follow SOLID principles:
   - Single responsibility principle - one function = one job. Only wrappers and big box functions (like main) can omit this principle.
   - Open/closed principle - prefer small functions which can be reuse to build another bigger functionality.
   - Liskov substitution principle - this can be omitted, because this principle requires inheritance support by compiler.
   - Interface segregation principle - do not put everything to one interface, create instead whole interface hierarchy
   - Dependency inversion principle - when function needs more than 1 implementation of sub function, use pointer to functions.
5. When your project will need 2 or more implementation of class, create abstract class (structure with pointers to function). This will allow you to create a polymorphism
6. Hidden structure cannot be allocate on stack, but using VLA we can do that. If structure has not big memory allocating to pointers, more efficient way is to allocate object on stack using special macro from OOP in C.

8.2 Examples

1. ```
   /* Correct */
   void pair_init(Pair* const pair, const int first, const int second)
   {
       pair->first = first;
       pair->second = second;
   }

   Pair* pair_create(const int first, const int second)
   {
       Pair *pair = calloc(1, sizeof(*pair));
       pair_init(pair, first, second);
   ```
   ```

```
    return pair;
}

void pair_destroy(Pair* const pair)
{
    free(pair);
}

void pair_deinit(Pair* const pair)
{
    (void)memset(pair, 0, sizeof(*pair));
}

/* Incorrect */
Pair *pair_new(const int first, const int second)
{
    Pair *pair = calloc(1, sizeof(*pair));
    pair_init(pair, first, second);

    return pair;
}

void pair_delete(Pair* const pair)
{
    free(pair);
}
```

2. ```
   /* Correct */
   /* pair.h */

   typedef struct Pair Pair;

   Pair* pair_create(int first, int second);
   int pair_get_first(const Pair* pair);
   int pair_get_second(const Pair* pair);
   /* etc. */

   /* pair.c */
   struct Pair
   {
       int first;
       int second;
   };

   /* Incorrect */
   /* pair.h */

   typedef struct Pair
   {
       int first;
       int second;
   } Pair;

   Pair* pair_create(int first, int second);
   ```
   ```

```
int pair_get_first(const Pair* pair);
int pair_get_second(const Pair* pair);
```

3.
```
/* Correct */
ssize_t find_key(const int* const t, const size_t len, const int key)
{
  for (size_t i = 0; i < len; ++i)
  {
    if (t[i] == key)
    {
      return i;
    }
  }

  return -1;
}

/* Incorrect */
ssize_t find_key(const int* const t, const size_t len, const int key)
{
  if (len == 0) {return -1;}
  if (t[len - 1] == key) {return len - 1;}
  return find(t, len - 1, key);
}

ssize_t find_key(const int* const t, const size_t len, const int key)
{
  size_t i = 0;
  while (i < len && t[i] != key)
  {
    ++i;
  }

  return (i == len) ? -1 : i;
}
```

4.
```
/* Corerct (Single responsibility and open closed) */
int sum(const int* t, size_t len);
int create_socket(const char* host, int port);
int create_connection(int socket);
int send(int fd, const void* data, size_t len);
void close_connection(int fd);

static inline void init_connection(const char* const host,
                                   const int port)
{
  const int socket = create_socket(host, port);
  return create_connestion(socket);
}

static inline void send_sum(const int* const t,
                            const size_t len,
                            const int fd)
```

```
{
  int sum = sum(t, len);
  send(fd, &sum, sizeof(sum));
}

/* Incorrect */
void send_sum(const int* const t, const size_t len)
{
  int sum = 0;
  for (size_t i = 0; i < len; ++i)
  {
    sum += t[i];
  }

  int fd;
  fd = /* some code with connection establishing*/

  write(fd, &sum, sizeof(sum));
  close(fd);
}

/* Correct (Interface Segregation) */
/* Balanced or Unbalanced */
typedef struct Tree Tree;
/* Interface for Self Balancing Trees like RBT */
typedef struct SBTree SBTree;
/* Interface for Unbalanced Trees like BST */
typedef struct UBTree UBTree;

/* Incorrect */
/* What about tree_balance(Tree *tree)? RBT does not need this
   function */
typedef struct Tree Tree;

/* Correct (Dependency inversion) */
typedef int (*cmp_f)(int a, int b);

/* a < b -> a is before b */
int cmp(int a, int b);

/* a < b -> a is after b */
int cmp_rev(int a,int b);

void sort(int *t, size_t len, cmp_f cmp);

/* Incorrect */
/* a < b -> a is before b */
void sort(int *t, size_t len);

/* a < b -> a is after b */
void sort_rev(int *t, size_t len);
```

5.
```
/* Correct */
/* rbt.h */
```

```
typedef struct Rbt Rbt;

/* avl.h */
typedef struct Avl Avl;

/* sbtree.h */
typedef struct SBTree
{
    void* tree;
    bool (*key_exist)(const void *tree, const void *key);
    /* pointers to another functions */
} SBTree;

static inline bool sbtree_key_exist(const SBTree* tree,
                                    const void* key)
{
    return tree->key_exist(tree->tree, key);
}

/* Incorrect */
/* rbt.h */
typedef struct Rbt Rbt;

/* avl.h */
typedef struct Avl Avl;

/* RBT is compatible from functionality point of view, but interface
   is not created, this is incorrect */
```

6.
```
/* Correct */
void foo(const size_t payload)
{
    Object* ptr_p;

    OBJECT_ALLOCA(ptr_p, object_sizeof() + payload);
    object_init(ptr_p, "Alice play with Bob");

    /* some code */

    object_deinit(ptr_p);
}

/* Incorrect, main memory will be alloc on heap, there is no advantage
    to alloc list on stack, also by using special macro readability is
    decreased */
void foo(void)
{
    List *list_p;

    OBJECT_ALLOCA(list_p, list_sizeof());

    /* some code */
}
```

8.3 Reasons

1. Convection helps with implementation of interfaces and also improve readability
2. More private memory = more secure program. Forward declaration force compiler to produce an error when programmer uses directly access to structure member. This will bring encapsulation to project
3. KISS improves readability and maintainability. Also force user to create smaller functions which improves reusability.
4. SOLID has been created for full object oriented programming languages. But following those principles in 80-90% will improve OOP style in C.
5. Interface and abstract classes are very common strategy to create class (functionality) hierarchy. This can be done in pure C also by using simple structure with pointer to functions.
6. When structure is small and allocate huge chunk of memory inside (like vector) there is no advantage to allocate object on stack (from optimization point of view). That's why you need to be careful with allocating objects on stack. This is tricky and requires special macro. More simple and readable solution is to use class_create function.

9 Performance

"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming."

Donald Ervin Knuth

9.1 Description

1. Prefer aliasing using restrict keyword.
2. Inline functions results in each call to an inline function being substituted by its body, instead of normal call. This results in faster code bit it adversely affects code size, particularly if the inline function is large and used often. So use inline wisely.
   - All functions in headers should be inline.
   - Static functions called several times in loops should be inline.
   - Wrappers for a few functions should be inline.
   - Getter, setter should be inline.
   - Long functions should not be inline.
   - Complicated functions should be inline.
3. Aliasing using local variable.
4. Prefer stack than heap.
5. Prefer sequential memory operation instead of directly writing assignment.
6. Always try to factor out as much as possible.
7. Avoid bitwise operations. Prefer use bool array or a few bool variables if possible and if memory is not critical resource.
8. Use static memory in function when:
   - function will be used only in 1 thread
   - variable is huge, is an array or a structure
   - Memory is set by developer, like int t[] = {1, 2, 3, 4};
   - function will be call several times
9. Do not use constructor and destructor in loop if possible. Use them once and change structure value by using init function.
10. Prefer flexible array member instread of pointer to void.

9.2 Examples

1. ```
   /* Correct */
   void foo(int* restrict a_p, int* restrict b_p);

   /* Correct */
   void foo(void* restrict a_p, int* restrict b_p);

   /* Incorrect. Aliasing useless. Pointers to different base types are
      not supposed to alias */
   void foo(int* restrict a_p, double* restrict b_p);
   ```
   ```

2.
```c
/* Correct, getter should be inline */
static inline int pair_get_first(const Pair* const p)
{
    return p->first;
}

/* Correct, this is wrapper for 3 functions */
static inline void foo_wrapper(void)
{
    foo_func1();
    foo_func2();
    foo_func3();
}

/* Correct function sum is simple and is call several time in loop */
static inline int add7(const int a, const int b)
{
    return (a % 7 + b % 7) % 7;
}

int foo(const int* const t, const size_t size)
{
    int sum = 0;
    for (size_t i = 0; i < size; ++i)
    {
        sum = add7(sum, t[i]);
    }

    return sum;
}

/* Incorrect. Function is to long to inline */
static inline void foo(void)
{
    /* over 80 lines with a lot of code */
}

/* Incorrect. Function is too complicated to be inline  */
int foo(void)
{
    int res;
    /* allocation for structures */
    /* preprocessing */
    /* data processing */
    return res;
}
```

3.
```c
/* Correct */
void foo(int* const a_p, const size_t size)
{
    int a = *a_p;

    for (size_t i = 0; i < size; ++i)
    {
```

```
        a += bar(i);
    }

    *a_p = a;

    /* some code */
}

/* Incorrect */
void foo(int* const a_p, const size_t size)
{
    for (size_t i = 0; i < size; ++i)
    {
        *a_p += bar(i);
    }

    /* some code */
}
```

4. ```
   /* Let's say we need to process any container in loop. After loop this
       container is no longer needed */


   /* Correct. Container will be created on stack */
   Pair* pair_p = pair_create();
   pair_init(pair_p);

   for (size_t i = 0; i < CONSTANT; ++i)
   {
       process(pair_p);
       save(pair_p);
   }

   /* Incorrect. If main parts of list (like: head, tail, length) are on
      the stack, but every single node on heap, there is no sense. Still
      we process container allocated at heap */
   List* list_p = list_create();

   for (size_t i = 0; i < CONSTANT; ++i)
   {
       process(list_p);
       save(list_p);
   }
   ```

5. ```
   typedef struct Foo
   {
       int a;
       int b;
   } Foo;

   Foo foo1 = {2, 3};
   Foo foo2 = {0};

   /* Correct */
   ```

```c
(void)memcpy(&foo2, &foo1, sizeof(foo1));

/* This is also correct */
foo2 = foo1;

/* Incorrect */
foo2.a = foo1.a;
foo2.b = foo1.b;

/* Correct */
(void)memset(&foo2, 0, sizeof(foo2));

/* Incorrect */
foo2.a = 0;
foo2.b = 0;
```

6.
```c
/* Correct */
static void foo(int* const t, const size_t size)
{
    int val = calculateVal();
    for (size_t i = 0; i < size; ++i)
    {
        t[i] += val;
    }
}

/* Incorrect */
static void foo(int* const t, const size_t size)
{
    for (size_t i = 0; i < size; ++i)
    {
        t[i] += calculateVal();
    }
}


/* Correct */
static void foo(...)
{
    func1();
    if (cond1)
    {
        func2();
    }
    else
    {
        func3();
    }
}

/* Incorrect */
static void foo(...)
{
    if (cond1)
```

```
    {
        func1();
        func2();
    }
    else
    {
        func1();
        func3();
    }
}
```

7. ```
   /* Correct */
   bool cond1;
   bool cond2;
   bool cond3;

   cond1 = /* some code */
   cond2 = /* some code */
   cond3 = /* some code */

   /* Correct, To collect separable options using an int is ok */
   int options;

   options = O_WRITE | O_TRUNCATE;

   int fd = open("./file", options);

   /* Incorrect */
   int conds = 0;

   conds |= cond1 << 0;
   conds |= cond2 << 1;
   conds |= cond3 << 2;

   if (conds & (1 << 2))
   {
       /* cond2 is TRUE */
   }
   ```

8. ```
   /* Correct array is big and has constant values */
   int foo(const size_t idx)
   {
       static const t[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
       return idx < 10 ? t[idx] : 0;
   }

   /* Incorrect. Single variable is not a huge struct or array */
   int foo(void)
   {
       static int a;

       /* some code with a */

       return a;
   ```

```
}
```

9.
```
/* Correct. Allocation is moved out from loop */
void foo(void)
{
    Pair* pair_p = pair_create(0, 0);
    for (int i = 0; i < CONST; ++i)
    {
        pair_init(pair, i, CONST - i);

        /* some code with pair */
    }

    pair_destroy(pair_p);
}


/* Incorrect. Allocation in loop is too costly */
void foo(void)
{
    for (int i = 0; i < CONST; ++i)
    {
        Pair* pair_p = pair_create(i, CONST - i);
        /* some code with pair */

        pair_destroy(pair_p);
    }
}
```

10.
```
/* Correct, sizeof(List_node) = sizeof(next_p) + sizeof(size_of) */
struct List_node
{
    struct List_node* next_p;   /* pointer to next node */
    size_t size_of;             /* size of node */

    byte_t data[];              /* placeholder for data */
};

/* Incorrect. Always prefer FAM (Flexible Array Member) instead of
   pointer to void */
struct List_node
{
    struct List_node* next_p;   /* pointer to next node */
    size_t size_of;             /* size of node */

    void* data_p;               /* placeholder for data */
};

/* Incorrect. It is not C89, sizeof(List_node) = sizeof(next_p) +
   sizeof(size_of) + sizeof(data) */
struct List_node
{
    struct List_node* next_p;   /* pointer to next node */
    size_t size_of;             /* size of node */
```

```
    byte_t data[1];             /* placeholder for data */
};

/* Incorrect. It is not GNU99, sizeof(List_node) = sizeof(next_p) +
   sizeof(size_of) */
struct List_node
{
    struct List_node* next_p;   /* pointer to next node */
    size_t size_of;             /* size of node */

    byte_t data[0];             /* placeholder for data */
};
```

9.3 Reasons

1. The basic idea of restrict is relatively simple: it tells the compiler that the pointer in question is the only access to the object it points to.In other words, with restrict we are telling the compiler that the object does not alias with any other object that the compiler handles in this part of code. It should be used if applicable but remember restrict violation is undefined behavior. Restrict keyword is needed only for compatible types like: int* int*, double* double* etc. Remember that void* and char* is compatible with every type, and restrict keyword should be added.

2. There are several advantages to using inline functions:

   - Inline function allows to write real pure functions.
   - No function call overhead.
   - As the code is substituted directly, there is no overhead, like saving and restoring registers. Optimizer works for single function, so we have better scheduling registers and cache.
   - Lower argument evaluation overhead. The overhead of parameter passing is generally lower, since it is not necessary to copy variables. If some of the parameters are constants, the compiler can optimize the resulting code even further.
   - No function call means no branching of the code thus no pipeline flush.

   Also disadvantages:

   - Code size increase. Mainly if inline function is used in many places.
   - Large code makes cache misses in .code section and we must wait for reading code from RAM.
   - Using huge inline functions which are called once or twice, not in loop usually makes code slower. (Look at the two previous disadvantages).

3. It allows to generate better code, if during the execution we can reduce the number of memory accesses instead of reading the value from memory every time it is used.

4. Malloc is too costly on the other hand allocations on stack is only subtraction stack pointer. Additionally memory from stack will be automatically freed instead of heap and this is the most important advantage.

5. Sequential memory access is much faster than random access (one by one). RAM can be trigger into burst mode which enables transfer a few DWORDS in 1 cycle using dedicated long registers. Every functions from string.h have this property. So memset, memcpy, memmove, memcmp should be used to work with big memory chunk.

6. Factoring out as mach as possible use much better pipeline and generate smaller code on each branch.

7. This rule should be take into account after choosing design. Bitwise operation costs few cycles, always prefer single move. E.g. if we have POSIX options, use normally bitwise operations. On the other hand if we have got conditions, it would be better to save these operations in separately variables.

8. If array or structure is on stack and it is filled by values written manually, in every function call this array or structure will be setted again and again. If this array of structure is quite huge and we know thier usage is safety (only 1 thread), this memory should be static. (ELF, section .bss)

9. Always, first design and then think about performance. It would be better to make allocate memory on heap once and then changed many times it instead of repeatedly alloc - dealloc operations.

10. If we use FAM instead of pointer to void, we have got 1 calloc, memcpy, memset instead of 2. Less fragmentation, better SSE will decrease bottlenecks connected with memory overhead. Also prefer [ ] than [1] because you won't been calculate structure size manually.

10 Miscellaneous

10.1 Description

1. "do - while" as exception of { } placement.
2. When using sizeof(), always prefer using variable name instead of type.
3. Always write first scope (static, extern) of function.
4. Always add comma after enum option.
5. Always alias pointer to type, not to variable name.
6. Never use unsigned types to reverse loop (from N to 0).
7. Never play with C syntax.
8. Always use one variable statement in for loop.
9. Never play with bitwise operator when you want to make aryth.
10. Never use auto, register keywords.
11. Value in enum should contains name of enum as a prefix of value.
12. Function without argument list should have void as argument.
13. Functionlike macro shall not include the ending semicolon.
14. Functionlike macro without input argument must have brackets .
15. Use goto only for centralized exiting of functions.
16. Always add default statement to switch case.
17. Prefer switch case when a few point cases are possible instead of if else chain.
18. Global variables should be defined in header but declared only in source files.

10.2 Examples

1. ```
   /* Correct */
   do {
       /* some code */
   } while (0);

   /* Incorrect */
   do
   {
       /* some code */
   }
   while (0);
   ```

2. ```
   /* Correct */
   int* intptr_p = calloc(sizeof(*intptr_p), 5);

   /* Incorrect */
   int* intptr_p = calloc(sizeof(int*), 5);
   ```

```
Huge_struct foo = { /* Proper initialization */ };
Huge_struct bar;

/* Correct */
(void)memcpy(&bar, &foo, sizeof(foo));

/* Incorrect */
(void)memcpy(&bar, &foo, sizeof(Huge_struct));
```

3. ```
   /* Correct */
   static inline foo(void);

   /* Incorrect */
   inline static foo(void);
   ```

4. ```
   /* Correct */
   enum foo
   {
       OPTION1,
       OPTION2,
   };

   /* Incorrect */
   enum foo
   {
       OPTION1,
       OPTION2 /* <-- there is no comma */
   };
   ```

5. ```
   /* Correct */
   int* intptr_p = NULL;

   /* Incorrect */
   int *intptr_p = NULL;
   ```

6. ```
   /* Correct */
   for (ssize_t i = CONSTANT; i > 0; --i)
   {
       process();
   }

   /* Incorrect. This is infinite loop. Better idea would be us signed
      integer for loop counter */
   for (size_t i = CONSTANT; i > 0; --i)
   {
       process();
   }
   ```

7. ```
   /* Correct */
   while (ptr_p[i] != '\0')
   {
       ++i;
   }
   ```
   ```

```
/* Incorrect */
while (*ptr_p++ != '\0');
```

8.
```
/* Correct */
int b = 1;

for (size_t a = 0; a < 100; ++a)
{
    b *= 2
}

/* Incorrect */
for (int a = 0, b = 1; a < 100; a++, b *= 2);
```

9.
```
/* Correct */
int a = b % 32;
int c = b * 8;
int d = b / 16;

/* Correct we are sure that c = 2^k */
int a = b & (c - 1);

/* Incorrect */
int a = b & 31;
int c = b << 3;
int d = b >> 4;
```

10.
```
/* Correct */
int sum(const int* const t, const size_t size)
{
    int sum = 0;
    for (size_t i = 0; i < size; ++i)
    {
        sum += t[i];
    }

    return sum;
}

/* Incorrect */
int sum(const int* const t, const size_t size)
{
    auto int sum = 0;
    for (register size_t i = 0; i < size; ++i)
    {
        sum += t[i];
    }

    return sum;
}
```

11.
```
/* Correct */
typedef enum parser_state_t
{
    PARSER_STATE_WORKING,
    PARSER_STATE_STOPPED,
} parser_state_t;

/* Incorrect */
typedef enum parser_state_t
{
    WORKING,
    STOPPED,
} parser_state_t;
```

12.
```
/* Correct */
void foo(void); /* OK. There is no arguments list */

/* Incorrect */
void foo(); /* Is this variadic function ? */
```

13.
```
/* Correct */
#define PRINT1() error("error print\n")

if (ptr_p == NULL)
{
    ERROR1();
    return -1;
}

/* Incorrect. Do not add semicolon at the ending */
#define PRINT3() error("error print\n");

PRINT3();
```

14.
```
/* Correct */
#define FOO() print("fancy print\n")

FOO();

/* Incorrect. Functionlike macro must have brackets even if there is
   no arguments at input */
#define FOO print("fancy print\n")

FOO;

/* What's now ? Not an ICE ? */
int a = FOO;
```

15.
```
/* Correct */
int func(int a)
{
    int result = 0;
    char* buffer_p = calloc(sizeof(*buffer_p), CONSTANT);
```

```
    if (!buffer_p)
    {
        return -1;
    }

    if (condition)
    {
        while (loop)
        {
            process();
        }

        result = 1;
        goto out_free_buffer;
    }

    /* some code */

out_free_buffer:
    FREE(buffer_p);
    return result;
}

/* Incorrect*/
int func(int a)
{
    int result = 0;
    char* buffer_p = calloc(sizeof(*buffer_p), CONSTANT);

    if (condition)
    {
        while (loop)
        {
            if (second cond)
            {
                goto end;
            }
        }

        FREE(buffer_p);
        return 1;
    }

end:
    FREE(buffer_p);
    return -1;
}
```

16.
```
/* Correct */
void foo(int val)
{
    switch (val)
    {
```

```
        case 1:
        {
            /* some code */
            break;
        }
        case 2:
        {
            /* some code */
            break;
        }
        default:
        {
            fprintf("ERROR, Unsupported value:%d\n", val);
        }
    }
}

/* Incorrect */
void foo(int val)
{
    switch (val)
    {
        case 1:
        {
            /* some code */
            break;
        }
        case 2:
        {
            /* some code */
            break;
        }
    }
}

/* Incorrect, now default is not needed, but if we add some values to
   enum in future this switch case will safe us from bugs related to
   not checking supporting values */
typedef enum option
{
    OPTION_1,
    OPTION_2,
} option_t;

void foo(option op)
{
    switch (op)
    {
        case OPTION_1:
        {
            /* some code */
            break;
        }
        case OPTION_2:
```

```
        {
            /* some code */
            break;
        }
        default:
        {
            fprintf("ERROR, Unsupported value:%d\n", val);
        }
    }
}
```

17.
```
/* Correct */
int foo(int val)
{
    switch (val)
    {
        case 1:
        {
            /* some code */
            break;
        }
        case 2:
        {
            /* some code */
            break;
        }
        case 5:
        {
            /* some code */
            break;
        }
        case 100:
        {
            /* some code */
            break;
        }
        default:
        {
            fprintf("ERROR, Unsupported value:%d\n", val);
        }
    }
}

/* Correct, this is not point comparision, so if else is needed */
int foo(int val)
{
    if (val > 1 && val < 5)
    {
        /* some code */
    }
    else if (val > 6 && val < 100)
    {
        /* some code */
    }
```

```
    else if (val < 500)
    {
        /* some code */
    }
    else
    {
        /* some code */
    }
}

/* Correct, double is not compatible with int, long ... so if else is
   needed */
double foo(double val)
{
    if (val == 0.0)
    {
        /* some code */
    }
    else if (val == 1.0)
    {
        /* some code */
    }
    else if (val == 2.0)
    {
        /* some code */
    }
    else
    {
        fprintf("ERROR, Unsupported value:%lf\n", val);
    }
}

/* Incorrect */
int foo(int val)
{
    if (val == 1)
    {
        /* some code */
    }
    else if (val == 2)
    {
        /* some code */
    }
    else if (val == 5)
    {
        /* some code */
    }
    else if (val == 100)
    {
        /* some code */
    }
}
```

18. `/* Correct */`

```c
/* ascii.h */
extern const char new_line;

/* ascii.c */
const char new_line = '\n';

/* Incorrect */

/* ascii.c */
const char new_line = '\n';
```

10.3 Reasons

1. This is easier to read do while loop, when condition is in the same line as loop ending
2. We want to avoid problems when changing the type. After change of type, the sizeof operator can remain unchanged.
3. It is easier to read first scope of function (global, file) and then is inline or not.
4. We always add comma at the end of enum option because there is no difference after added new option in enum.
5. Pointer is a part of type, not a variable name.
6. This rules helps to avoid infinite loops and errors related to this.
7. We want to keep our code easy to read.
8. As above.
9. Keyword auto and register are obsolete and there are not applicable for modern C code.
10. Using deprecated keywords is not a good code style.
11. It is harder to static analyze code without prefix name of enum values.
12. We want to avoid variadic functions in code, which is not really variadic functions.
13. Functionlike macro must be called like normal function (with semicolon at the end of function call). So, do not interfere normal function call and never include semicolon at the end of functionlike macro.
14. If #define is functionlike macro, even without input arguments, it should be called like normal function. It's easy to make a mistake and assign functionlike macro to variable.
15. The rational for using gotos is:
    - Unconditional statements are easier to understand and follow.
    - Nesting is reduced.
    - Errors by not updating individual exit points when making modifications are prevented.
16. Adding default statement is safe in all cases, even if we write down all possible values in cases. We can't be sure that in future no values are added.
17. Switch case is an example of point condition, only 1 value can be matched with 1 condition. That's why compiler can change comparison chain to simple jump table. Of course you must remember that only types compatible with int, long ... can be used in switch case.
18. Header are copying into source file, so declaration will be pasted into a few source file with the same name and value. This is a symbol redeclaration error from linker. To avoid it declaration is only in 1 source file (memory is allocated only once), in header we have only "aliasing" to the variable.