



SPICE
Project

Spice project coding style and coding conventions

Draft 2

Table of Contents

1.C and C++ style.....	3
2.Source Files.....	3
2.1.Names.....	3
2.2.Line width.....	3
2.3.Tabs.....	3
2.4.White spaces.....	3
2.5.New Line.....	3
3.Comparing.....	3
4.TRUE, FALSE and NULL.....	3
5.Static storage initialization.....	4
6.Fixme and todo.....	4
7.ASSERT.....	4
8.sizeof	4
9.const.....	4
10.goto.....	4
11.Defining Constant values.....	4
12.void argument.....	5
13.Short functions.....	5
14.Return on if.....	5
15.Names.....	5
16.Optimization.....	6
17.Spacing.....	6
18.Function Indentation.....	6
19.Branching indentation.....	7
20.Types indentation	8
21.Vertical indentation.....	8
22.Multi line macro indentation.....	9
23.Multi line array indentation.....	9
24.C++.....	9
24.1.One super.....	9
24.2.Data members.....	9
24.3.Object reference.....	9
24.4.Templates.....	9
24.5. '*' and '&'.....	10
24.6.Class indentation.....	10
24.7.Constructor indentation.....	11
24.8.bool.....	11
24.9.Operator overloading.....	11
24.10.AutoRef and AutoPtr.....	11
25.Spice client.....	11
25.1.#ifdef PLATFORM.....	11
25.2.Use standard macros.....	11

1. **C and C++ style**

All of the following are applicable for both c and c++, except for section 25 which is c++ extension.

2. **Source Files**

2.1. Names

Use lower case and separate words using underscore (e.g., file_name.c, header.h).

Use cpp file extension for c++ source files and hpp extension for c++ template files. Use standard file extension for c source and header files.

2.2. Line width

No more than 100 character on a single line

2.3. Tabs

Tabs are not allowed, use 4 spaces instead

2.4. White spaces

Trailing white spaces are not allowed

2.5. New Line

Use Unix style line ending (i.e., LF)

New line (i.e., EOL) must be the last char in the file

3. **Comparing**

Use right-hand comparison style.

Examples:

use `(var == 7)` instead of `(7 == var)`

use `(function(var) > 7)` instead of `(7 < function(var))`

4. **TRUE, FALSE and NULL**

Use TRUE FALSE and NULL instead of 1 and 0 in order to improve code readability. TRUE FALSE is not relevant for c++, in c++ use the built-in bool type.

5. **Static storage initialization**

To improve code readability, explicitly initialize variables that depend on default initialization with zero/null.

6. **Fixme and todo**

Comments that are prefixed with “fixme” describe a bug that need to be fixed. Generally, it is not allowed to commit new code having “fixme” comment. Committing “fixme” is allowed only for existing bugs. Comments that are prefixed with “todo” describe further features, optimization or code improvements, and they are allowed to be committed along with the relevant code.

7. **ASSERT**

Use it freely. ASSERT helps testing function arguments and function results validity. ASSERT helps detecting bugs and improve code readability and stability.

8. **sizeof**

Apply function style to sizeof (i.e., use sizeof(x))

9. **const**

Use const keyword when applicable to improve code reliability and celerity.

10. **goto**

Using goto is allowed in c code for error handling. In any other case it will be used only as a special exception.

11. **Defining Constant values**

Use defines for constant values for improving readability and ease of changes. Alternatively, use global const variables.

12. **void argument**

Don't add explicitly void argument in functions without arguments. (i.e., void function_name() is OK)

13. **Short functions**

Try to split code to short functions, each having simple functionality, in order to improve code readability and re-usability. Prefix with *inline* short functions or functions that were splitted for readability reason.

14. **Return on if**

Try to return immediately on if in places that can reduce indentation level.

Example:

prefer

```
void function(int *n)
{
    if (!n) {
        return;
    }
    ...
}
```

on

```
void function(int *n)
{
    if (!n) {
        return;
    } else {
        ...
    }
}
```

15. **Names**

- Don't underestimate the importance of name choosing. Good names make the code more easy to understand and reduce the required level of code documentation. When choosing names, prefer long meaningful names over short vague name.

- Variable and Function names – use lower case and separate words using underscore (e.g., sample_variable_name)
- Structures, class and enum names – one or more words, each word start with upper case (e.g., Name, SampleStructName)
- Defines and enum items names – uppercase words separated using underscores (e.g., NAME, SAMPLE_DEFINE_NAME)

16. Optimization

Keep optimization to fast path code. Prefer safe, clear and easy to maintain coding for code that is not on the fast path.

17. Spacing

Use spacing for improving code readability.

```
for (i = 0; i < 10; ++i) {
    some_func(var, i * i + *ptr, &var, sizeof(var));
}
```

18. Function Indentation

- No spaces between function name to left bracket.
- Curly bracket start on new line.
- Functions must be padded with empty lines on both sides

```
void function(type1 arg1, type2 arg2, type2 arg3)
{
    ...
}
```

- In case of a new line in arguments list, align it to the first argument type.

```
void function(type1 arg1,
             type2 arg2,
             type3 arg3)
```

Or

```
void function(type1 arg1, type2 arg2,
             type3 arg3)
```

- New line is acceptable only in arguments list

19. Branching indentation

- Add space after a branch keyword and after the right bracket.
- Curly bracket starts on the same line the branch starts.
- Place curly brackets after all control flow constructs even where optional. This convention reduces branching bugs that are difficult to detect. It also makes it easier to add logging messages during debugging since it eliminates the need to add the brackets.

```
if (condition) {
    ...
} else if (condition) {
    ...
} else {
    ...
}
```

In case of long condition statement, prefer having additional temporary variable over multiple line condition statement.

In case of new line in condition statement.

```
if (long_name && very_long_name && very_long ||
    var_name) {
```

or indent using two tabs

```
if (long_name && very_long_name && long_name ||
    var_name) {
```

Break function arguments list in long condition statement according to section 18.

```
while (condition) {
    ...
}

do {
    ...
} while (condition);

for (i = x; i < y; i++) {
    ...
}
```

```

switch (x) {
case A: {
    ...
    break;
}
case B:
    ...
    ...
    break;
default:
    ...
}

```

20. Types indentation

```

struct StructName {
    type1 name1;
    type2 name2;

    ...
};

enum {
    A,
    B,
    C = 10,
    D,
};

union {
    type1 name1;
    type2 name2;
    ...
} u;

```

21. Vertical indentation

Use one space no tabs and no vertical alignment.

```

long var_name_1 = 7;
int var_name_2 = 11111;
unsigned long long_var_name_1 = 666;
char long_var_name_1 = 'a';

```

```

void f1(int a, char ch);
unsigned long f2(int a, char ch);

```

22. Multi line macro indentation

```
#define MACRO_NAME(a, b, c) {      \  
    int ab = a + c;                \  
    int ac = a + b;                \  
    int bc = b + c;                \  
    f(ab, ac, bc);                 \  
}
```

23. Multi line array indentation

```
char *array[] = {  
    "item_1",  
    "item_2",  
    "item_3",  
};
```

24. C++

C++ style extends C Coding style

24.1. One super

Avoid having more than one super class. Inherit from one super class and multiple interfaces (abstract class) for having multiple inheritance.

24.2. Data members

Prefix all protected and private data members with underscore (i.e., `_member_name`). Using public data members is allowed only as an exception. Use getters and setters instead.

24.3. Object reference

For code clarity use object reference (i.e., `Type& var`) in places where it is not allowed to have null pointer and is not permitted to release the object.

24.4. Templates

The use of c++ templates is limited to simple containers.

24.5. '*' and '&'

'*' and '&' , should be directly connected with the **type names**.

Example:

```
void function(int* i, int& n);
```

24.6. Class indentation

```
class ClassName: public SuperClass, public FirstInterface,
                public SecondInterface {
public:
    ClassName();
    virtual ~ClassName();

    type public_function_1(type var);
    type public_function_2();
    ...

protected:
    type protected_function_1(type var);
    type protected_function_2();
    ...

private:
    type private_function_1(type var);
    type private_function_2();
    ...

public:
    type public_member_1;
    type public_member_2;
    ...

protected:
    type _protected_member_1;
    type _protected_member_2;
    ...

private:
    type _private_member_1;
    type _private_member_2;
    ...
};
```

24.7. Constructor indentation

```
ClassName::ClassName(type1 arg1, type2 arg2, type3 arg3,
                    type4 arg4)
    : SuperClass(arg1, arg2)
    , _member_1 (arg3)
    , _member_2 (arg4)
    ...
{
    ...
}
```

24.8. bool

Use built-in bool 'true' and 'false' instead of TRUE and FALSE.

24.9. Operator overloading

Avoid using operator overloading. It is only allowed as an exception.

24.10. AutoRef and AutoPtr

Use AutoRef and AutoPtr style object for automatic object release. Using those objects simplify function cleanups and exception handling.

25. **Spice client**

25.1. #ifdef PLATFORM

Use #ifdef <platform> only in cases of different logic that depends on platform. In all other case add common interface (e.g., in platform.h) and keep specific platform implementation in platform directory (e.g., windows).

25.2. Use standard macros

```
LOG_INFO
LOG_WARN
LOG_ERROR
ASSERT
PANIC
DBG
THROW
THROW_ERR
```