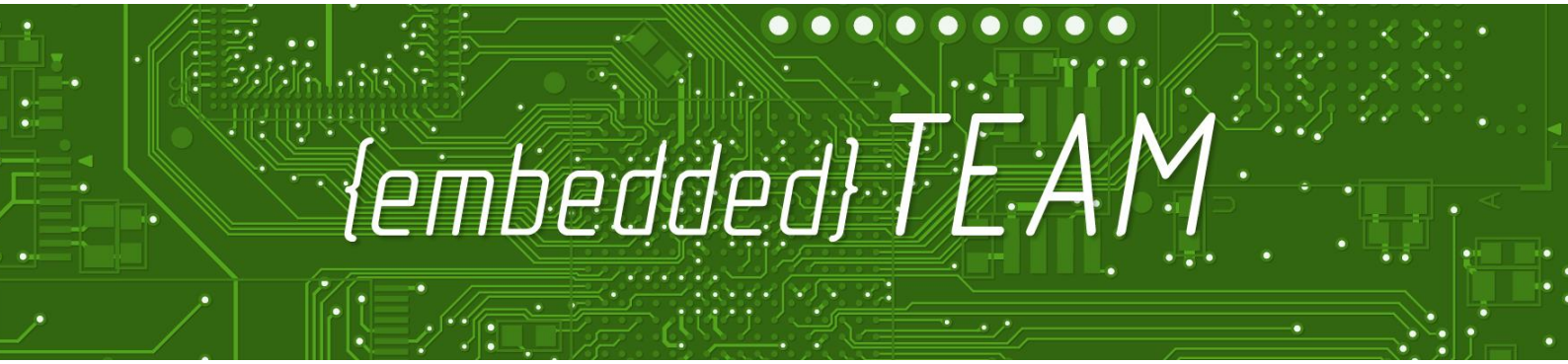
EMBEDDED TEAM C++ PROGRAMMING STYLE GUIDELINE



+7.812.907.64.32
info@embedded.team
www.embedded.team

Содержание

Введение

Этот документ определяет правила оформление исходных кодов программ на языке программирование C++ для разработчиков Embedded Team. Они основаны на общих рекомендациях сообщества разработчиков C++ и предназначены для написания программ в едином стиле.

Следование данному руководству настоятельно рекомендовано при разработке любого программного обеспечения Embedded Team, включая операционные системы, драйвера, утилиты и т.д. Также мы рассчитываем, что программное обеспечение, построенное на продуктах Embedded Team, будет соответствовать данным рекомендациям.

Общие правила

1. Все алгоритмы реализуются на языке C++ стандарта ISO/IEC 14882:1998.

2. Исходные коды программ должны безошибочно компилироваться компиляторами C++98, C++03, C++11 и C++14.

3. Компилятор не должен выдавать предупреждений о возможных ошибках.

4. Исходные коды программ должны быть разработаны с применением объектно-ориентированного подхода в программировании.

5. Базовые типы данных должны использоваться лишь, как их переопределения.

```
// Signed types
int8, int16, int32, int64, float32, float64

// Unsigned types
uint8, uint16, uint32, uint64

// Type returned by sizeof
size_t

// Type of minimal addressable memory cell
cell
```

6. Исходный код, комментарии, имена файлов должны быть написаны на английском языке.

Правила именования

Основные правила именования

7. Имена типов данных должны быть определены в смешанном регистре начиная с символа верхнего регистра.

```
Object, TimerInterrupt
```

8. Имена переменных должны быть определены в смешанном регистре начиная с символа нижнего регистра.

```
value, interruptResouce
```

9. Имена констант и значений перечислений должны быть определены в верхнем регистре с использованием символа подчёркивания для разделения слов.

```
MAX_RESOURCES, ERROR_VALUE
```

10. Имена методов и функций должны быть глаголами или начинаться с глагола, и определены в смешанном регистре начиная с символа нижнего регистра.

```
release(), unlock(), getPeriod(), isConstructed()
```

11. Имена пространств имен должны быть определены в нижнем регистре.

```
system, driver, driver::processor
```

12. Имена типов шаблона должны начинаться с символа верхнего регистра, а все остальные символы должны быть определены в нижнем регистре.

```
template <typename Type, class Alloc>
```

13. Все сокращения и аббревиатуры в именах не должны быть в верхнем регистре.

```
char* html;          // NOT: hTML
UsbDriver resource; // NOT: USBDriver
getUsbDriver();      // NOT: getUSBDriver()
testCpuFan();        // NOT: testCPUFan()
```

Использование символов в верхнем регистре при именовании конфликтует с правилами, описанными в предыдущих пунктах. Также определение имени, как hTML или uSB – не очевидно при чтении. Ещё одна проблема проиллюстрирована в примере выше, когда в имени соединены два слова – читабельность такого наименования сильно затруднена, так как слово, написанное за аббревиатурой, сливается с ней.

14. Приватные переменные классов должны иметь суффикс подчёркивания

```
class SomeClass
{
    setValue(int32 value)
    {
        value_ = value;
    }

private:

    int32 value_;
};
```

Даная практика позволяет легко отличить перемененные класса от локальных переменных, а так же, как показано в примере выше, это удобный способ для разрешения проблем именования аргументов методов, устанавливающих значения полей класса, или конструкторов классов.

15. Все имена должны быть написаны на английском языке.

```
fileName; // NOT: imyaFaila
```

Дополнительные правила именования

16. Префиксы *set* и *get* должны быть добавлены к именам методов, предоставляющих доступ к переменным класса.

```
getFrequency();
setFrequency(clock);
```

Все переменные объектов должны быть приватными, а доступ к ним, при необходимости, должен осуществляться по средствам методов объявленных в их интерфейсах.

17. Префикс *compute* может быть добавлен к именам методов, осуществляющих определенные вычисления.

```
computeAverage();
```

18. Префикс *find* может быть добавлен к именам методов, осуществляющих определённый поиск.

```
findElement();
```

19. Префикс *initialize* может быть добавлен к именам методов, осуществляющих дополнительную инициализацию объектов или инициализирующих определенный концепт.

```
printer.initializeFontSet();
```

20. Имена переменных, представляющих коллекции, должны быть во множественном числе.

```
vector<Configuration> configurations;
int32 values[];
```

Данное правило улучшает понимание того, какие операции могут быть совершенны над элементами коллекций.

21. Переменные циклов должны иметь имена *i, j, k* и т.д. в зависимости от глубины вложенности цикла.

```
for(int32 i=0; i<nElements; i++)
{
    for(int32 j=0; j<nElements; j++)
    {
        // ... Statements ...
    }
}
```

22. Префикс *is* должен быть добавлен к именам методов, возвращающих булевой тип, или булевых переменных.

```
isConstructed, isInitialized
```

Также могут быть использованы альтернативы *has, can, should*, если они подходят лучше в определённых ситуациях.

23. Префиксы, дополняющие друг друга, должны быть использованы при необходимости.

```
get/set, add/remove, create/destroy, start/stop, insert/delete,
increment/decrement, old/new, begin/end, first/last, up/down,
min/max, next/previous, open/close, show/hide, suspend/resume, и т.д.
```

24. Сокращение в именах не допустимо.

```
sendCommand(); // NOT: sendCmd();
```

Данное правило определяет требование к общим словам, присутствующим в словарях, которые не должны быть сокращены.

Например:
cmd должно быть *command*;
init должно быть *initialize*;
drv должно быть *driver*;
и т.д.

С другой стороны, общеизвестные аббревиатуры не должны быть раскрыты.

Например:
CentralProcessingUnit должно быть cpu;
UniversalSerialBus должно быть usb;
HypertextMarkupLanguage должно быть html;
и т.д.

25. Имена переменных не должны содержать дополнительных ссылок на типы их данных.

```
Driver* driver; // NOT: Driver* pDriver;
Driver& driver; // NOT: Driver& driverRef;
```

26. Отрицающие булевы имена не должны быть использованы.

```
bool isError; // NOT: bool isNoError;
bool isSet;   // NOT: bool isNotSet;
```

27. Значения перечислений могут иметь префикс их типа данных.

```
enum Led
{
    LED_RED,
    LED_GREEN,
    LED_BLACK
};
```

28. Имена классов исключений должны иметь суффикс *Exception*.

```
class InterruptException;
```

Оформление исходных кодов

29. Отступ должен быть в четыре пробела.

30. Каждый блок должен начинаться и заканчиваться на новой строке, если правила не определяют обратного.

```
void method()
{
    // ... Statements ...
    {
        // ... Statements ...
    }
    // ... Statements ...
}
```

31. Объявление пространства имён должно иметь следующую форму:

```
namespace identifier
{
    // ... Statements ...
}
```

Данное правило распространяется как на заголовочные файлы, так и на файлы с исходным кодом.

32. Объявление классов должно иметь следующую форму:

```
class SomeClass : public ::ns::BaseClass, public ::api::SomeInterface
{
    typedef ::ns::BaseClass Parent;

public:

    SomeClass();

    virtual ~SomeClass();

    virtual void publicMethod1() = 0;

    virtual void publicMethod2();

    // ... Other statements ...

protected:

    virtual void protectedMethod1() = 0;

    virtual void protectedMethod2();

    // ... Other statements ...

private:

    void privateMethod1();

    static void privateMethod2();

    static const int32 CONSTANT = 10;
```

```
    static int32 variable1_;

    int32 variable2_;

    int32 variable3_;

    // ... Other statements ...

};
```

- Все спецификаторы наследуемых классов должны быть объявлены явно.
- Имена наследуемых классов должны быть указаны относительно глобального пространства имён.
- Правила выше имеют полное отношение и к объявлению структур.
- Области видимости класса должны быть явно указаны, если они используются, и упорядочены в последовательности: *public*, *protected* и *private*.
- Каждая область видимость сначала должна включать описание чисто виртуальных методов, затем виртуальных, а после статических. После должны следовать описание констант, статических полей и переменных полей класса.

33. Объявление шаблонных классов должно иметь следующую форму:

```
template <typename Type, int32 CONSTANT=0, class Alloc=::Allocator>
class SomeClass : public ::Object<Alloc>
{
    typedef ::Object<Alloc> Parent;

    // ... Statements ...
};
```

- Если параметр шаблона может быть любого типа данных, то он должен быть специфицирован как *typename*.
- Если параметр шаблона может быть только класс, то он должен быть специфицирован как *class*.
- Целочисленные параметры не должны иметь ключевого слова *const*.
- Типы параметров шаблонов по умолчанию должны быть указаны относительно глобального пространства имён.

34. Объявление конструкторов должно иметь следующую форму:

```
SomeClass (bool value) : Parent(),
    isConstructed_ (false),
    object_        (value),
```

```
    value_          (value),
    address_        (NULL){
    // ... Statements ...
}
```

- Все конструкторы родительских классов должны быть явно вызваны.
- Все переменные класса должны быть явно проинициализированы до входа в тело конструктора.
- Последовательность инициализации переменных должна совпадать с последовательностью их объявлений, так как компилятор, в любом случае, будет инициализировать переменные согласно последовательности объявлений.

35. Объявление деструкторов классов должно иметь следующую форму:

```
virtual ~SomeClass()
{
    // ... Statements ...
}
```

Все деструкторы классов должны быть виртуальными.

36. Объявление деструкторов структур должно иметь следующую форму:

```
~SomeClass()
 {
     // ... Statements ...
 }
```

Как правило, структура описывает набор каких-то данных. Этот набор может быть строго определённого размера, например, для наложения структуры на область физической памяти. В таких случаях виртуализация метода в структуре приводит к увеличению её размера на объём, необходимый каждому конкретному компилятору для хранения таблицы виртуальных методов. Исходя из этого, структура, как правило, не должна имеет виртуальных методов, и её деструктор также не объявляется, как виртуальный.

37. Объявление структур битовых полей должно иметь следующую форму:

```
union SomeField
{
    SomeField(){}
    SomeField(uint32 v){value = v;}
   ~SomeField(){}

    uint32 value;
    struct
    {
        uint32 name0 : 1;
        uint32       : 1;
        uint32 name2 : 3;
        uint32       : 27;
```

```
    } bit;
};
```

Любая структура, описывающая битовые поля, должна быть объединена с полем, дающим возможность сразу установить значение всех полей.

38. Объявление методов должно иметь следующую форму:

```
virtual void somePublicMethod(const ::ns::Object& r, ::ns::Object v)
{
    // ... Statements ...
}
```

Тип аргумента всегда должен быть указан относительно глобального пространства имен. В подавляющем большинстве случаев, передача аргумента должна происходить по ссылке или по значению, тоже касается и возвращаемого значения. Также при разработке важно учитывать спецификатор *const*, так как его наличие говорит, что метод не будет изменять ссылающиеся данные, а его отсутствие этого не гарантирует.

39. Объявление публичных и защищенных методов должно иметь следующую форму:

```
virtual void somePublicMethod()
{
    // ... Statements ...
}
```

Любой метод, являющийся интерфейсом класса, должен быть виртуальный. Ключевое слово *virtual* должно быть всегда добавлено ко всем методам, переопределяющим виртуальные методы родительских классов.

40. Объявление приватных методов должно иметь следующую форму:

```
void somePrivateMethod()
{
    // ... Statements ...
}
```

Приватные методы класса реализуют его внутреннюю логику и не требуют их виртуализации. Виртуализация приватных методов приводит лишь к увеличению количества таблиц виртуальных методов или их размеров.

41. Объявление чисто виртуальных методов должно иметь следующую форму:

```
virtual void someVirtualMethod() = 0;
```

42. Объявление условий *if-else* должно иметь следующую форму:

```
if(value < MAX_VALUE)
{
    // ... Statements ...
}
```

```
if(value < MAX_VALUE)
{
    // ... Statements ...
}
else
{
    // ... Statements ...
}

if(value < MAX_VALUE)
{
    // ... Statements ...
}
else if(value == MAX_VALUE)
{
    // ... Statements ...
}
else
{
    // ... Statements ...
}
```

43. Объявление условия *switch* должно иметь следующую форму:

```
switch(value)
{
    case 0:
    {
        // ... Statements ...
    }
    break;

    case 1:
    case 2:
    {
        // ... Statements ...
    }
    break;

    default:
    {
        // ... Statements ...
    }
    break;
}
```

44. Объявление цикла *for* должно иметь следующую форму:

```
for(int32 i=0; i<MAX_VALUE; i++)
{
    // ... Statements ...
}
```

45. Объявление цикла *while* должно иметь следующую форму:

```
while(counter < MAX_VALUE)
{
    // ... Statements ...
}
```

46. Объявление цикла *do-while* должно иметь следующую форму:

```
do
{
    // ... Statements ...
}
while(counter < MAX_VALUE);
```

47. Объявление конструкции *try-catch* должно иметь следующую форму:

```
try
{
    // ... Statements ...
}
catch(Exception& exception)
{
    // ... Statements ...
}
```

Файлы исходных кодов

Имена файлов

48. Файлы с исходным кодом программ должны иметь расширение *cpp*.

```
SomeClass.cpp
```

49. Заголовочные файлы программ должны иметь расширение *hpp*.

```
SomeClass.hpp
```

50. Имя файла должно складываться из пространств имён и имени класса, разделенных точкой.

```
system.Thread.hpp
system.Thread.cpp
```

Общее правило присвоения имён файлам:
namespace.nestednamespace.SomeClassName.extension

51. Имена файлов имеющих разные расширения не могут быть одинаковыми.

```
system.Interrupt.cpp
system.Interrupt.ll.asm
```

Некоторые компиляторы, создавая объектные файлы, складывают их в одну директорию и заменяют их расширения. Если в приведённом примере оба

файла будут иметь одинаковое имя, то результатом компиляции будут объектные файлы *system.Interrupt.obj*. То есть один файл перепишет другой, что приведёт к отсутствию последнего на входе линковщика.

Содержание файлов

52. Каждый файл должен содержать объявление и/или реализацию единственного класса.

53. Заголовочный файл, содержащий реализацию, не должен иметь отдельного файла с исходным кодом.

    Другими словами, любой класс может иметь заголовочный файл с объявленным интерфейсом и файл с исходным кодом его реализации или реализация может быть целиком размещена в заголовочном файле. Второй вариант, как правило, используется для написания шаблонных классов или вспомогательных классов необходимых исключительно для разделения логики реализации.

54. Длина строки файла не должна превышать 120 символов.

55. Специальные символы, такие как символ табуляции, должны быть исключены.

56. Количество строк в файле не должно превышать 3000 строк.

    Как правило, большое количество строк в файле говорит о не структурности его функционала. Когда файл объединяет в себе не только реализацию собственного интерфейса, но и вспомогательного функционала. Понимание логики таких файлов затруднено.

57. Количество строк в реализации метода не должно превышать 100 строк.

    Большое количество строк в реализации метода приводит к затруднению его понимания. Как правило, всегда есть функционал, который можно вынести в отдельный метод, даже если он будет использован единожды.

58. Разнесённые на несколько строк выражения должны быть очевидны.

```
int32 sum = a + b + c
          + d + e;

void function(argument1, argument2,
              argument3);

SomeClass(int32* data, int32 length) : Parent(),
    data_   (data),
    length_ (length){
    // ... Statements ...
}

for(int32 i = 0; i < length;
    i += 5){
```

```
    // ... Statements ...
}
```

Подобные ситуации могут происходить, когда длина строки превышает 120 символов. Определить корректность того, как строки должны быть перенесены – непросто, но общие рекомендации следующие:

- Перенос строки после запятой;
- Перенос строки после оператора;
- Выравнивание новой строки от начала выражения в предыдущей строке.

59. Метод класса, по возможности, не должен быть вызван после его реализации в пределах одного файла.

```
void SomeClass::methodBefore()
{
    // Always available for calling
    method();
}

void SomeClass::method()
{
}

void SomeClass::methodAfter()
{
    // The best way is not to call
    method();
}
```

Данное правило позволяет облегчить поиск реализации вызванного метода в пределах файла и структурировать его. Разработчик, анализирующий файл, всегда знает, что реализация вызванного метода всегда будет определенна ниже.

Включение файлов

60. Любой заголовочный файл должен быть защищён от повторного включения в одну и ту же единицу трансляции.

```
/**
 * Header of system.TimerInterrupt.hpp
 */
#ifndef SYSTEM_TIMER_INTERRUPT_HPP_
#define SYSTEM_TIMER_INTERRUPT_HPP_

// ... Statements ...

#endif // SYSTEM_TIMER_INTERRUPT_HPP_
```

Общее определяющее правило:
NAMESPACE_NESTEDNAMESPACE_SOME_CLASS_NAME_HPP_

Таким образом, имя файла должно быть переведено в верхний регистр, все точки заменены на символы подчеркивания, имя класса разделено символами подчёркивания и в конце добавлено ещё одно подчеркивание.

61. Любой заголовочный файл должен включать в себя необходимое и достаточное количество внешних интерфейсных заголовочных файлов, подключенных в последовательности их использования.

```
/**
 * Some class.
 */
#ifndef SYSTEM_SOME_CLASS_HPP_
#define SYSTEM_SOME_CLASS_HPP_

// NOTE: The base class mentioned first.
#include "Object.hpp"

// NOTE: The interface class mentioned second.
#include "api.SomeClass.hpp"

// NOTE: The parameter of the class method mentioned third.
#include "api.OtherInterface.hpp"

// NOTE: The class variable type mentioned fourth.
#include "utility.LinkedList.hpp"

namespace system
{
    // NOTE: We have declared only pointer on Thread class
    //       and don't use it as interface.
    //       Thus, we must only declare the class for
    //       successfully compiling the file anywhere.
    class Thread;

    class SomeClass : public ::Object<>, public ::api::SomeClass
    {
        // NOTE: The best way is to redefine the type of
        //       the base class, as it suddenly might be changed.
        typedef ::Object<> Parent;

    public:

        // ... Statements ...

    protected:

        virtual void setInterface(::api::OtherInterface& param);

    private:

        /**
         * Some list.
         */
        ::utility::LinkedList<Thread*> list_;
```

```
    };
}
#endif // SYSTEM_SOME_CLASS_HPP_
```

Приведенный выше пример показывает общий принцип построения заголовочных файлов, когда они включают лишь необходимый и достаточный набор описаний, чтобы быть включенными в любую единицу трансляции, которая будет успешно скомпилирована и слинкована.

62. Любой файл с исходным кодом первым должен включать в себя собственный заголовочный файл.

    Данное правило позволяет убедиться, что собственный заголовочный файл не зависит от других заголовочных файлов и исправно компилируется в любой единице трансляции.

63. Любой файл с исходным кодом должен включать в себя заголовочные файлы, отсортированные согласно их уровню абстракции.

    Данное правило не определяет точного подхода к включению заголовочных файлов, а лишь описывает общие принципы. Любой заголовочный файл определяет интерфейс определенного уровня абстракции. Таим образом, чем выше уровень абстракции, тем позже файл должен быть включён в исходный код. Например, системные файлы должны быть включены первыми, далее могут быть включены файлы библиотеки утилит, а затем файлы необходимые для внутренней реализации.

64. Директивы *include* должны быть размешены только в начале файла, сразу после блока документации.

Комментирование исходных кодов

65. Все комментарии должны быть написаны на английском языке.

66. Двойной слеш должен использоваться для всех комментариев, включая многострочные комментарии.

```
// This is one comment line

// This is a multi-line comment,
// which must be ended with a point.
```

- Между двойным слешем и комментарием должен быть добавлен пробел.
- Все комментарии должны начитаться с заглавной буквы.
- Многострочные комментарии должны заканчиваться точкой.

67. Комментарий должен начитаться с той же позиции, что и комментируемый код.

```
void method()
```

```
{
    // Call some method
    someMethod();
}
```

68. Файлы, классы, методы и поля должны быть прокомментированы согласно соглашениям JavaDoc.

69. Формат *DocBlock* комментариев должен иметь следующую форму:

```
/**
 * Short one line description.
 */

/**
 * Short one line description.
 *
 * Full multi-line description if it is necessary.
 * This description may contain as many lines as it needs.
 */

/**
 * Short one line description.
 *
 * Full multi-line description if it is necessary.
 * This description may contain as many lines as it needs.
 *
 * @tag1 a tag one line description.
 * @tag2 a tag one line description.
 * @tag3 a tag one line description.
 */
```

Любой блок комментариев должен иметь короткое описание. Далее может следовать полное описание, а затем набор тегов.

70. Заголовочные файлы должны быть полностью прокомментированы.

Если класс имеет файл исходного кода и объявленный интерфейс в заголовочном файле, то заголовочный файл имеет приоритет. Он должен содержать полное описание, в то время как файл исходного кода может иметь только короткое описание. Это позволяет разработчику, изучающему класс, получить полную картину его интерфейса, просмотрев лишь один файл.

71. Комментарий к файлу-классу должен иметь следующую форму:

```
/**
 * Short one line class description of a file.
 *
 * Full class description.
 * This description may contain as many lines as it needs.
 *
 * @author    Name Surname, name.surname@embedded.team
 * @copyright 2017-2018, Embedded Team
 * @license   http://embedded.team/license/
```

```
 * @codepage  cp1251
 */
```

Каждый файл должен содержать один класс и начинаться с описания класса, реализованного в нём.

72. Комментарий к методу класса должен иметь следующую форму:

```
/**
 * Removes a first matched element from the list.
 *
 * The method scans the list starting from the beginning
 * to find a passed element. If the passed element and
 * a scanning element are equal, the passed counter is
 * decremented, and if the counter is zero, the element
 * of the list will be removed from it.
 *
 * @param element an element for searching.
 * @param number  a number of matched element for removing.
 * @return true if the element is removed successfully.
 * @throws ::util::ListException the list exception.
 * @throws ::Exception the global exception.
 */
virtual bool removeElement(const Type& element, int32 number) = 0;
```

- Короткий комментарий к методу должен начинаться с глагола, указанного в имени метода первым, в третей форме единственного числа.
- При описании параметров не следует ссылаться на тип параметра, его спецификаторы и методы передачи.
- Тип исключения должен быть указан относительно глобального пространства имён.
- Не используемые теги должны быть исключены из комментариев.

73. Комментарий к полю класса должен иметь следующую форму:

```
/**
 * Description of the variable.
 */
int32 value_;
```

74. Комментарий к вложенному классу, структуре или перечислению должен иметь следующую форму:

```
/**
 * Short one line class description of a file.
 *
 * Full class description.
 * This description may contain as many lines as it needs.
 */
```

Если класс содержит внутренние типы данных, то они должны быть также прокомментированы.

75. Комментарий к шаблонному классу должен иметь следующую форму:

```
/**
 * Short one line class description of a file.
 *
 * Full class description.
 * This description may contain as many lines as it needs.
 *
 * @param Type  data type of an element.
 * @param Alloc heap memory allocator class.
 */
template <typename Type, class Alloc>
```

76. Комментарий в *DocBlock* могут содержать следующие теги:

- @author [имя автора][, email адрес]
- @copyright [описание]
- @license [url]
- @link [url]
- @param [имя] [описание]
- @return [описание]
- @todo [описание]
- @throws [тип] [описание]
- @codepage [код страницы]

Дополнительные правила

Типы

77. Приведение типов всегда должно быть выполнено явно без использования Си стиля.

    Для явного приведения типов должны использоваться *static_cast*, *dynamic_cast*, *const_cast*, *reinterpret_cast* операции.

78. Базовые типы данных, используемые в объявлении интерфейсов, должны быть знаковыми.

79. Типы, используемые только в одном файле исходного кода, должны определяться только в нем.

Как правило, такие типы данных должны быть определенны в отдельном заголовочном файле и подключены в файл с исходным кодом директивой *include*.

80. Переопределённые типы, используемые только в одном файле исходного кода, должны переопределяться только в нем.

```
/**
 * Some class.
 */
#include "SomeClass.hpp"
#include "system.Mutex.hpp"

// NOTE: The rule usage
typedef ::system::Mutex Mtx;

/**
 * Does something.
 */
void SomeClass::doSomething()
{
    Mtx mtx;
    mtx.lock();
    //
    // ... Statements ...
    //
    mtx.unlock();
}
```

81. Переопределённые типы, используемые в одном классе, должны быть скрыты в его области видимости.

```
/**
 * Some class.
 */
#ifndef SOME_CLASS_HPP_
#define SOME_CLASS_HPP_

#include "Object.hpp"
#include "api.SomeClass.hpp"
#include "system.Mutex.hpp"

class SomeClass : public ::Object<>, public ::api::SomeClass
{
    // NOTE: The rule usage
    typedef ::Object<> Parent;
    typedef ::system::Mutex Mtx;

public:

    /**
     * Constructor.
     */
    SomeClass() : Parent(),
        mtx_ (){
```

```
        setConstruct( true );
    }

    /**
     * Destructor.
     */
    virtual ~SomeClass()
    {
    }

    /**
     * Does something.
     */
    virtual void doSomething()
    {
        mtx_.lock();
        //
        // ... Statements ...
        //
        mtx_.unlock();
    }

private:

    /**
     * Some mutex.
     */
    Mtx mtx_;

};
#endif // SOME_CLASS_HPP_
```

Переменные

82. Все переменные класса должны быть приватными.

    Данное правило позволяет ограничить доступ к инкапсулированным данным класса.

83. Все переменные структуры должны быть публичными.

    Структуры должны использоваться лишь с целью инкапсуляции данных и/или наложения на области физической памяти для предоставления доступа к регистровой модели аппаратуры, разбора принятой информации из каналов связи и т.д. В таких случаях объявление приватных переменных – не целесообразно, если только нет необходимости в намеренном сокрытии доступа к определённым полям, например, эти поля не доступны для чтения/записи на аппаратном уровне.

84. Символ указателя или ссылки должен стоять сразу после имени типа, а не перед именем переменной.

```
int32*  value; // NOT: int32  *value
int32&  value; // NOT: int32  &value
```

```
int32*& value; // NOT: int32* &value AND int32 *&value
```

85. Переменные, доступ к которым может осуществляться не явно для компилятора, должны иметь спецификатор *volatile*.

    Данное правило позволяет сократить отладку проектов при их сборке с включённым режимом оптимизации компиляторов.

86. Глобальные переменные должны иметь ссылку на глобальное пространство имён.

```
::systemContext.getCpuName();
```

Выражения

87. Новое выражение всегда должно начитаться с новой строки.

```
int16 a;          // Allowed
int16 a, b;       // Allowed
int32 c; int64 d; // Prohibited
```

Условия

88. Неявное сравнение с нулём должно быть исключено.

```
if(value != 0)    // NOT: if(value)
if(value != 0.0) // NOT: if(value)
```

    Стандарт C++ не определяет чётко, что целочисленные числа и числа с плавающей запятой должны быть бинарным нулём.

89. Сложные условия должны быть исключены. Используйте временные булевы переменные.

```
// Allowed
bool isError = (a > 5) || (b < 10);
bool isDone = c == 100;
if(isError || isDone)
{
    // ... Statements ...
}

// Prohibited
if(((a > 5) || (b < 10)) || (c == 100))
{
    // ... Statements ...
}
```

90. Ожидаемый результат всегда должен быть в *if* части, а исключения в *else* части условного выражения.

```
bool isOk = a < 8;
if(isOk)
```

```
{
    // ... Statements ...
}
else
{
    // ... Exception statements ...
}
```

Циклы

91. Бесконечные циклы должны быть *while(true)*.

```
// Allowed
while(true)
{
    // ... Statements ...
}

// Prohibited
for(;;)
{
    // ... Statements ...
}

// Prohibited
while(1)
{
    // ... Statements ...
}
```

Переходы

92. Оператор *goto* не должен использоваться.

    Оператор *goto* нарушает идеи структурированного подхода к программированию. Если необходимо выполнить определенный набор инициализаций и в случае ошибки на одной из стадий вернуть определенную ошибку и произвести деинициализацию того, что было инициализировано, то можно использовать следующий подход:

```
int32 SomeClass::main()
{
    int32 stage = 0;
    int32 error = -1;

    // Initialization
    do
    {
        // Stage 1
        stage++;
        if( not ::Class1::initialize () )
        {
            error = 1;
            break;
        }
```

```
        // Stage 2
        stage++;
        if( not ::Class2::initialize () )
        {
            error = 2;
            break;
        }
        // Stage 3
        stage++;
        if( not ::Class3::initialize () )
        {
            error = 3;
            break;
        }
        // Stage complete
        stage = -1;
        error = 0;

        // ... Statements after successful initialization ...

    }
    while(false);

    // Deinitialization
    switch(stage)
    {
        default:
        case  3: ::Class3::deinitialize();
        case  2: ::Class2::deinitialize();
        case  1: ::Class1::deinitialize();
        case  0: break;
    }
    return error;
}
```

История изменений

- 2017/08/30 – в примере правила 57 исправлена последовательность команд *ifndef* и *define* препроцессора.
- 2017/09/04 – в правило 35 внесена корректировка описания; в правило 55 внесена корректировка оформления; правило 21 исключено и список перенумерован.
- 2017/09/07 – добавлено 10 правил, список перенумерован.
- 2018/02/12 – в правиле 24 удалена повторяющаяся запись.
- 2018/06/09 – правило 5 о коневом классе *Object* исключено.

+7.812.907.64.32
info@embedded.team
www.embedded.team

Автор: Сергей Байгудин
Корректор: Игорь Кибальник