UNREAL MOJO

Поваренная книга нереального программиста

(Корпоративный стиль оформления исходного текста на языках программирования C/C++ и Objective C/C++ и прочие рекомендации специалисту)

[iOS & Mac OS X]

2012

Содержание

История

Версия	Дата	Автор	Комментарии
0.1	22.07.2012	Cyril	Первая черновая редакция: форматирование, общие положения, правила для “C”
	31.09.2012	Cyril	Вычитка, дополнение правил для «C» Общие положения, правила для «Objective-C»
	07.10.2012	Cyril	Вычитка, дополнение правил для «Objective-C» Заготовки новых глав
	16.10.2012	Antoxa	Вычитка
1.0	17.10.2012	Antoxa Cyril	Вычитка, дополнения Публичный релиз
		Cyril	Правка опечаток
	22.10.2012	Cyril	Правка опечаток, вынос справочной информации в сноски. Главы переименованы с учетом потенциальных дополнений документа правилами для других ЯП
	02.11.2012	Cyril	Правка опечаток, дополнения от Ksenks
	22.12.2012	Cyril	Правка опечаток, дополнения от Bealex
	15.03.2013	Antoxa Cyril	Явное указание на запрет пропуска ivar для синтезированных пропертей (было возможно непонимание без прямого запрета)
1.0.1	28.08.2013	Cyril	Форматирование блоков-замыканий

1 Вступление

1.1 Бла-бла часть

Здесь должно было быть доброе вступление и пожелание принять к сведению приведенные в этом документе мысли, бережно собранные для вас Капитаном Очевидность. Вместо этого здесь будет очередное перечисление забытых банальностей.

Итак, данный документ содержит набор правил и рекомендаций. Следование этим правилам и рекомендациям не есть каприз или самодурство. В большинстве случаев навязываемые правила призваны помочь вам совершить как можно меньше ошибок при написании программ, а так же в наиболее ясной форме донести свои мысли, выраженные в написанном коде, до ваших коллег, душевное спокойствие которых во многом зависит от того, как вы преподнесете “материал”. Да что там говорить, вы сами должны быть способны разобраться в своих же собственных конструкциях спустя достаточно долгое время, причем разобраться быстро и без риска попасть в психиатрическую лечебницу.

Так же хочется заметить, что единообразное оформление поможет вашим старшим коллегам быстро и без всякого отладчика найти проблемы в коде.

Существует множество легенд о том, что современные средства разработки решают за программиста множество тривиальных проблем и позволяют не думать о “мелочах”. Это весьма грустная тенденция и нам хотелось бы надеяться, что в мире остались люди, до сих пор считающие себя умнее компилятора, которым они пользуются.

В этом документе основной акцент сделан не только на оформлении текста программ, но так же на основные правила, призванные облегчить понимание кода, избежать банальных ошибок, а так же повысить производительность скомпилированной программы (оптимизация компилятором не панацея).

Так же стоит заметить, что свод правил из этого документа не является темой для дискуссии.

1.2 Отсылки на классику

В тексте встречаются отсылки к различным правилам стандарта разработки ПО «MISRA C++»[1] и аналогичных или на нем же основанных стандартов «JSF AV C++»[2] или «HI C++»[3].

Факультативное ознакомление с этими стандартами приветствуется, особенно, если оно затронет ваше сердце и вами будут сделаны правильные выводы.

1 http://www.misra.org.uk/
2 http://www.jsf.mil/
3 http://www.programmingresearch.com/high-integrity-cpp/

1.3 Благодарности

Антону Турову и **Юрию Каткову** за «UnrealMojo Reference Coding Standard», дедушки настоящей «**Поваренной книги нереального программиста**».

Антону Турову, **Павлу Мазурину**, **Александру Бабаеву** и **Ксении Покровской** за замечания и дополнения.

Коллегам за непротивление злу насилием и конструктивную критику.

2 Правила именования для языка «C» и производных

2.1 Именование лексем

2.1.1 Имена Classes, Protocols, Functions, Constants, Namespaces, Globals

Именование этих лексем должно осуществляться с использованием префиксов.

Префиксы лексем, использующихся во всем проекте (имена классов, протоколов, констант), должны соответствовать аббревиатуре капитализированного имени проекта или кодового имени проекта и состоять из двух или трех символов. Например: «Анонимные Братья» — AB, «Матка Боска» — MB, «Хон Гиль Дон» — HGD, и т.д.

Для лексем, использующихся в нескольких проектах, возможно отступление от этого правила и аббревиатуры могут отличаться.

Следующее за аббревиатурой имя должно отражать суть объекта и может состоять из нескольких капитализированных слов. Использование “_” так же возможно. Однако имя не должно быть слишком длинным и перегруженным — необходимо использовать разумный компромисс.

```
// Objective-C
@interface HGDBase : NSObject<HGDDataProviderDelegate>
{
//  …
}
@end

// C++
class HGD_SOAPParser : public HGD_XMLParser
{
//  …
};
```

Дополнительно, для имен переменных и констант, перед аббревиатурой должен стоять однобуквенный префикс, показывающий область действия:

“g” — глобальная переменная

“s” — статическая переменная

“k” — константа (может быть опущена)

```
extern HGDBase* gHGDBase;
static HGDBase* sHGDBaseInstance = nil;
extern NSString* const HGDApplicationDidEnterBackgroundNotification;
extern NSString* const kHGDApplicationDidEnterBackgroundNotification;
```


2.1.2 Имена Types, Structures, Enums, Enum Members

Именование этих лексем может осуществляться с использованием упомянутых префиксов.

Дополнительно к этому имя типа может содержать суффикс “_t”, имя структуры префикс “_”. Использования определения структуры или перечисления в определителе типа не разрешено. Как следствие, использование анонимной (неименованной) структуры запрещено, тем не менее использование анонимного перечисления допускается, но в редких оправданных случаях.

```
//  WRONG!
typedef struct
{
//  …
} HGDCoreData;

//  WRONG!
typedef struct _HGDCoreData
{
//  …
} HGDCoreData_t;

//  CORRECT
struct _HGDCoreData
{
//  …
};
typedef struct _HGDCoreData  HGDCoreData_t;
```

Именование членов структуры регулируется теми же правилами, что и именование членов класса C++ (см. 2.1.3.Имена членов класса).

Именование членов перечисления регулируется правилом формирования имен констант, но в качестве префикса может использоваться либо имя перечисления, либо аббревиатура имени перечисления.

```
//  WRONG!
typedef enum
{
    ABResultCode_OK = 0,
    ABResultCode_Exception = 1
} ABResultCode;

//  CORRECT
enum ABResultCode
{
    ABResultCode_OK = 0,
    ABResultCode_Exception = 1
};
typedef enum ABResultCode ABResultCode;
```



```
//  CORRECT
enum ABResultCode
{
    ABRC_OK = 0,
    ABRC_Exception = 1
};
typedef enum ABResultCode ABResultCode;

//  CORRECT
enum ABResultCode
{
    kRC_OK = 0,
    kRC_Exception = 1
};
typedef enum ABResultCode ABResultCode;
```

Замечание: запятая после определения последнего члена перечисления категорически не приветствуется.

2.1.3 Имена членов класса

- class members (C++)
- ivars (Objective-C)

Имена должны быть значимы и понятны при чтении. Имена начинаются с маленькой буквы и могут состоять из нескольких слов (первое с маленькой буквы, остальные капитализированные). Мы не пользуемся венгерской записью и не помечаем каждый члена класса префиксом “m”, тем не менее возможно использование следующих префиксов:

- “_” — если член класса приватный (или используется приватно).
- “p” — если член класса указатель

Замечание: формально говоря, в Objective-C у нас все члены используются приватно.

Замечание: в Objective-C используются только указатели на класс и префикс “p” излишен.

```
// Objective-C
@interface HGDBase : NSObject<HGDDataProviderDelegate>
{
    NSString* _identifier;
}
@end

// C++
class HGD_SOAPParser : public HGD_XMLParser
{
    const char* _pIdentifier;
};
```

Замечание: в C++ принят несколько иной подход к именованию членов класса, особенно это касается капитализации. Поэтому данное правило для C++ носит рекомендательный характер и будет отдельно рассмотрено в соответствующем разделе.

2.1.4 Имена методов

В Objective-C имена должны быть значимы и понятны при чтении. Имена начинаются с маленькой буквы и могут быть составными, это же правило верно для каждой компоненты имени метода. Приватные методы могут начинаться с “_”.

В C++ принят несколько иной подход к именованию методов, особенно это касается капитализации. Поэтому данное правило для C++ носит рекомендательный характер и будет отдельно рассмотрено в соответствующем разделе .

2.1.5 Имена локальных переменных

Имя должно отражать суть переменной и может состоять из нескольких слов (первое с маленькой буквы, остальные капитализированные). Использование “_” так же возможно. Однако имя не должно быть слишком длинным и перегруженным — необходимо использовать разумный компромисс.

Использование коротких “незначащих” имен переменных не приветствуется (например целочисленные индексы “i”, “j” и “k”, указатели “p” и “q”, значения с плавающей точкой “x” и “y”, и подобные устоявшиеся имена). Использования однобуквенных имен “o” и “l” строго запрещено. Используйте значащие имена — index, counter, ptrSrc, ptrDst и т.п.

2.1.6 Имена аргументов функций и методов

Выбор имен аргументов функций и методов соответствует правилам 2.1.5 Имена локальных переменных, но так же перед именем может стоять префикс “i”, “o”, или “io”, показывающий “направление” аргумента.

2.2 Именование файлов

Имя файла должно соответствовать имени класса в нем описанного (с точностью до префикса). Если файл содержит несколько классов (см. соответствующее исключение), используется имя основного или базового класса.

Для файлов, не содержащих описания классов, имя выбирается исходя из смысловой нагрузки и так же должно иметь префикс. Так же это правило справедливо, если в файле описано несколько классов и невозможно выделить основной или базовый.

3 Форматирование и оформления исходных текстов на языке «C» и производных

3.1 Табуляция

Мы используем табуляцию в четыре пробела. Символ табуляции недопустим, редактор должен быть настроен на замену табуляции пробелами.

3.2 Использование пробелов

- унарная операция не отделяется от операнда (так же указатель и референс)

```
*p
p--
```

- бинарная операция отделяется от операндов пробелами

```
a = b;
b = b * c;
```

- пробел всегда ставится после запятой, и никогда до

```
void f(int a, int b, int c);
```

- указатель и референс не отделяются от типа в декларациях переменных
- указатель и референс отделяются от имени переменной в декларациях минимум одним пробелом

```
int*       p;
int&*      p;
```

- между лексемой и “;” пробел не ставится

```
a = b;
```

- после имени оператора перед “(“ всегда ставится пробел

```
if (a < b)
```

- после имени функции перед “(“ пробел не ставится

```
void f(int a, int b, int c);
```

- после “(“ и перед “)” пробел не ставится (так же и для “[“ и “]”) (см. исключения)

```
a = b * (c + d / (a – f));
```

- в конце строки пробелов быть не должно

3.3 Выравнивание и операторы

- пустой оператор “;” никогда не используется (так же “;” не ставится после “}” если это не продиктовано синтаксисом)
- только один оператор (с заключительным символом “;”) может находиться в одной строке (исключение: аргументы оператора for и конструкция else if)
- операторные скобки “{“ и “}” всегда находятся на отдельной строке (исключе-

ние: оператор do … while, оператор for без блока, или если это блок инициализации переменной)

- операторные скобки “{“ и “}” выравниваются по левой стороне оператора (исключение: «блоки-замыкания», см. 5.8 Форматирование блоков, или если это блок инициализации переменной)
- подчиненный оператор смещен вправо на размер табуляции
- содержимое операторных скобок смещено вправо на размер табуляции
- операторные скобки ставятся всегда, если в блоке содержится более одного оператора (спорный случай switch/case), или имеются вложенный оператор (пусть даже выполняющий роль операторных скобок), или один длинный оператор, перенесенный на несколько строк.

3.4 Использование пустой строки

Для удобства чтения и визуального разделения логических “блоков” программы, допускается использование пустой строки.

Допускается использование пустой строки для разделения “блоков” case/default в операторе switch/case/default.

В пустой строке не должно быть пробелов.

3.5 Длинные строки

Следует избегать очень длинных строк.

Если строка не влезает по ширине в среднестатистический экран[4], ее имеет смысл принудительно перенести. Для выражений подходящей точкой “разрыва” может быть правая часть бинарного оператора с низким приоритетом. Не следует начинать перенесенную строку с оператора.

Если длинная строка получается в результате вызова функции с большим набором аргументов, можно переносить на новую строку аргументы, располагая и выравнивая их друг под другом. Подобная практика так же может применяться для комментирования каждого аргумента вызываемой функции. В этом случае каждая строка с аргументом заканчивается запятой — начинать строку с аргументом с запятой запрещено.

Если длинная строка получается из-за длинного строкового литерала, имеет смысл переносить ее по правилам языка «C» с использованием “\”.

4 Существует много мнений о максимальном количестве символов в одной строке. Это значение может быть равно как 80 (старозаветная перфокарта), так и 120. Или даже больше — 200 символов на практике так же вполне читаемы.


```
double superlength = sqrt((x1 – x2) * (x1 – x2) + (y1 – y2) * (y1 – y2)) +
                      (estimated_innacuracy(x1, y1) +
                       estimated_innacuracy(x2, y2)) / 2.;

double superlength = estimate_superlength(x1, /* 1st coordinates */
                                          y1,
                                          x2, /* 2nd coordinates */
                                          y2);

const char * superliteral = “that's a very long literal\n” \
                            “no joke, I have no idea how long could it be!\n”;
```

Иногда так же пользуются автоматическим «подворачиванием» длинных строк. Этот спорный подход пусть останется на вашей совести. Так же длинные строки — это обычная ситуация в Objective-C, поэтому см. Правила для Objective-C.

3.6 Использование комментариев

Определенно в коде необходимо пользоваться комментариями. Документировать код следует хотя бы в тех частях, где поведение программы не очевидно. Данный документ не регламентирует использование комментариев и оставляет это на совести программиста.

В любом случае необходимо помнить: переизбыток комментариев — это так же плохо, как и их полное отсутствие.

Не допускаются пространные комментарии или комментарии ни о чем[5] — если возникла необходимость прокомментировать кусок кода, то именно код и нужно пояснить, отметив нюансы, которые побудили вас написать комментарий.

Забытые и неактуальные комментарии могут причинить вред — будьте внимательны и удаляйте все, что не относится к текущему состоянию программы.

5 Иногда в тексте программы в комментариях можно встретить телефонные номера, записанные программистом во время беседы или напоминания себе в духе «не забыть купить поллитра и макароны».

4 Использование конструкций языка «C»

4.1 Константы

Старайтесь явно задавать тип констант в выражениях (включая размер) избегая «скрытых» приведений типа на этапе компиляции.

Капитализация символов шестнадцатеричных констант не имеет значения.

Использование восьмеричных констант запрещено.

Для констант с плавающей запятой без целой части можно отбрасывать ноль слева от точки: .5, .125, и т.д.

Для констант с плавающей запятой без дробной части можно отбрасывать ноль справа от точки: 1., 125., и т.д.

Для нулевой константы с плавающей запятой можно использовать форму .0

Помните, что по умолчанию в C/C++ константы с плавающей точкой имеют тип double[6].

4.2 Типы

Старайтесь не использовать стандартные типы языка «C» — int, short, long, long long, double, float и т.п. — без особой необходимости.

В приложениях для операционных систем Apple, используйте типы, унаследованные из Apple ToolBox — UInt8, SInt8, UInt32, SInt32, и т.п. — только в случае крайней необходимости, продиктованной использованием конкретных API.

Запрещено полагаться на размер слова и порядок байт в памяти.

4.3 Определение локальных переменных, структур, перечислений, констант и типов

Локальные переменные (а так же структуры, перечисления, константы и типы) декларируются в начале функции.

Так же декларация имеет право быть в начале логического блока и как часть конструкции оператора for в стандарте C99. Декларация локальной переменной в середине логического блока имеет право быть, но должна быть обоснована.

Одна декларация описывает только одну переменную (исключение для оператора for в стандарте C99). Это же относится к членам структуры.

Так же необходимо инициировать переменную в декларации.

6 Что иронично, потому как базовым типом CGFloat в 32-битных системах Apple явно задан float.

4.4 Выражения

Не следует перегружать арифметические и логические выражения. Выражения должны быть лаконичны и легко читаемы.

При оформлении выражений необходимо четко следовать правилам использования пробелов.

В сложных случаях в выражениях лучше явно использовать скобки даже если приоритет операций позволяет обойтись без оных.

При одновременном использовании в выражениях логических и арифметических операций для сложных конструкций использование скобок категорически приветствуется.

При получении в выражении логического значения (значение типа bool, BOOL, Boolean, и т.п.) из арифметического, операция сравнения необходима, так же по возможности имеет смысл использовать скобки, если выражение их не имеет или достаточно сложное.

```
//  WRONG
bool result = a + b;
bool result = a != 0;

//  CORRECT
bool result = (a + b) != 0;
bool result = (a != 0);
```

Присвоения внутри выражения запрещены (правило 6-2-1 MISRA C++)

```
//  WRONG
if ((self = [super init]) != nil) …
```

4.5 Условия

Если результат выражения условия не логическое значение (не значение типа bool, BOOL, Boolean, и т.п.), **операция сравнения необходима всегда**.

Так же запрещено использование унарного логического отрицания на арифметическое выражение.

```
//  WRONG!
if (!_controller)
    _controller = [[HGDController alloc] init];

//  WRONG!
if (_controller)
{
    [_controller release];
    _controller = nil;
}

//  WRONG!
while (--count)
{
//  …
}
```



```
//  CORRECT
if (_controller == nil)
    _controller = [[HGDController alloc] init];

//  CORRECT
if (_controller != nil)
{
    [_controller release];
    _controller = nil;
}

//  CORRECT
while (--count != 0)
{
//  …
}
```

Использование константы в левой части оператора сравнения (обычно оператора равенства) не найдет понимания и встречного отклика в душах ваших коллег[7].
Помните, что строгое сравнение переменных с плавающей запятой или с константой с плавающей запятой чревато. Даже если эта константа ноль.

7 Да, мы слышали обоснования, но это скорее демонстрирует, что вы выпендриваетесь, при этом не веря себе, а так же выключили соответствующие предупреждения компиляции. Последнее, надо заметить, наказуемо.

4.6 Использование оператора перечисления (запятая)

Использование оператора перечисления (запятая) запрещено.

```
//  WRONG
[obj release], obj = nil;

//  CORRECT
[obj release];
obj = nil;
```

Замечание (для справки): по правилу 5-18-1 MISRA C++ использование оператора запятая запрещено как вредное с указанием на возможность добиться аналогичного эффекта иными возможностями языка C/C++[8].

8 Тем не менее известен один пример мирного применения оператора перечисления в компании, однако о том, насколько это использование “прозрачно” можно судить самим из плохого примера:

```
#define _DEF_TIMEOUT(n, t)                    \
      struct timeval n##_tv;                  \
      double n##_check;                       \
      {                                       \
            gettimeofday(&n##_tv, NULL);      \
            n##_check = (double)n##_tv.tv_sec + 1E-6 * (double)n##_tv.tv_usec + (double)t / 1000.;     \
      }

#define _CHK_TIMEOUT(n)                       \
      (                                       \
            gettimeofday(&n##_tv, NULL),                          \
            (double)n##_tv.tv_sec + 1E-6 * (double)n##_tv.tv_usec < n##_check       \
      )
…
bool OK = true;
while (OK && bytesToSend())
{
      _DEF_TIMEOUT(TO, 10000);

      while (OK && !bytesAvailable())
            OK = _CHK_TIMEOUT(TO);

      if (OK)
            sendBytes();
}
```


4.7 Использование оператора goto

Умное использование оператора goto не возбраняется, а грамотное использование приветствуется. Слепые попытки написать сложную конструкцию без goto, вводя дополнительные признаки и условия может сделать код нечитаемым.

Предпочтительно ставить метку оператора goto в начале логического «блока» после определения переменных, метка в середине логического блока обрамленного операторными скобками не приветствуется. Метка, используемая в качестве точки входа в цикл не приветствуется категорически.

4.8 Использование условного оператора if…else

По правилам, указанным в 3.Форматирование, стандартный вид условного оператора в простейшем случае должен быть следующим:

```
if (a > b)
    result = a;
else
    result = b;
```

С учетом многострочных блоков и вложенных условных операторов:

```
if (a > b)
{
    result = a;
    libre = false;
}
else if (a < b)    // else if в одной строке
{
    result = b;
    libre = false;
}
else               // a == b
{
    result = 0;
    libre = true;
}
```

Замечание: для цепочки if…else if ()… конструкция else if должна находиться в одной строке
Замечание: для не очевидного финального else приветствуется комментарий, отмечающий неявное условие, по которому управление передается в финальный блок[9]

9 Для справки: например, по правилу 6-4-2 MISRA C++ каждый оператор if ()… должен иметь else (пусть даже пустой) или комментарий, подтверждающий, что else не нужен. Мы можем подобный случай комментировать в начальном условии, если есть желание и это поможет разобраться в коде.


```
//  WRONG!
if (a > b)
    if (a > c)
        max = a;
    else
        max = c;
else
    max = b;

//  CORRECT
if (a > b)
{
    if (a > c)
        max = a;
    else
        max = c;
}
else
    max = b;
```

Замечание: в примере выше, обе конструкции выполняют идентичную операцию, однако вторая конструкция однозначна и легко читаема. Это следствие выполнения последнего правила списка 3.3 Выравнивание и операторы.

4.9 Использование оператора цикла for

4.9.1 Общие правила

По правилам, указанным в 3.Форматирование, стандартный вид оператора цикла for в простейшем случае должен быть следующим:

```
for (i = 0; i < length; i++)
    result += i * 3;
```

С учетом многострочных блоков и вложенных операторов:

```
for (i = 0; i < length; i++)
{
    result += i * 3;
    if (result > threshold)
        raise = true;
}
```

4.9.2 Условие выхода из цикла

Условие выхода из цикла не должно быть чрезмерно сложным. Для цикла for обычно достаточно проверки счетчика цикла, но так же может присутствовать одно дополнительное условие.

```
//  DOUBTFUL
for (i = 0; i < length && j < bounds && isSane(i, j); i++)
{
//  …
}
```



```
//  CORRECT
done = false;
for (i = 0; i < length && !done; i++)
{
//  …
    if (j < bounds)
        done = !isSane(i, j);
    else
        done = true;
}
```

Если нет необходимости в обратном, вычисляйте границу счетчика цикла один раз, а не в условии оператора. Избегайте вызовов методов и функций в цикле, если их можно вынести «за скобки».

Строгое сравнение счетчика цикла, при шаге отличном от единицы недопустимо (правило 6-5-2 MISRA C++).

4.9.3 Использование break и continue

Не рекомендуется злоупотреблять большим количеством break и continue. Для выхода из цикла рекомендуется использовать проверку переменной. В некоторых отдельных случаях для сложных вложенных циклов имеет смысл использовать goto.

Не рекомендуется располагать break и continue глубоко во вложенных блоках конкретного цикла.

4.9.4 Оператор for без блока

Запрещено использовать пустой оператор “;”, вместо него следует использовать операторные скобки.

```
//  WRONG!
for (i = len - 1; i > 0 && p[i] != 0; i--) ;

//  CORRECT
for (i = len - 1; i > 0 && p[i] != 0; i--)  {}
```


4.10 Использование оператора цикла while

4.10.1 Общие правила

По правилам, указанным в 3.Форматирование, стандартный вид оператора цикла while в простейшем случае должен быть следующим:

```
while (i >= result)
    result += i * 3;
```

С учетом многострочных блоков и вложенных операторов:

```
while (i < length)
{
    result += i * 3;
    if (result > threshold)
        i++;
}
```

4.10.2 Использование break и continue

Не рекомендуется злоупотреблять большим количеством break и continue. В некоторых отдельных случаях для сложных вложенных циклов имеет смысл использовать goto.

Не рекомендуется располагать break и continue глубоко во вложенных блоках конкретного цикла.

Вообще оператор continue предназначен скорее для циклов for.

4.11 Использование оператора цикла do…while

4.11.1 Общие правила

По правилам, указанным в 3.Форматирование, стандартный вид оператора цикла while в простейшем случае с учетом многострочных блоков и вложенных операторов:

```
do {
    result += i * 3;
    if (result > threshold)
        i++;
} while (i < length);
```

или

```
do
{
    result += i * 3;
    if (result > threshold)
        i++;
} while (i < length);
```

Для случая с однострочным блоком возможно следующее использование:

```
do
    result += i * 3;
while (i >= result);
```

4.11.2 Использование break и continue

Не рекомендуется злоупотреблять большим количеством break и continue. В некоторых отдельных случаях для сложных вложенных циклов имеет смысл использовать goto.

Не рекомендуется располагать break и continue глубоко во вложенных блоках конкретного цикла.

Вообще оператор continue предназначен скорее для циклов for.

4.12 Использование оператора switch/case/default

4.12.1 Общие правила

Каждый «блок» case и default должен иметь соответствующий break. Использование break внутри «блока» не приветствуется.

«Блок» может содержать несколько case.

Каждый оператор switch/case/default должен содержать «блок» default, даже если входные данные подразумевают, что он «лишний».

«Блок» default рекомендуется ставить последним в конструкции switch/case/default.

Не приветствуется использовать case на один «блок» с default.

Использование инфернальной мощи оператора switch/case/default и конструкций с установкой case в середине логического блока другого вложенного оператора не допускаются[10]:

```
//  WRONG
int n = (count + 7) / 8;

switch (count % 8)
{
case 0:
    do {
        *to++ = *from++;
case 7: *to++ = *from++;
case 6: *to++ = *from++;
case 5: *to++ = *from++;
case 4: *to++ = *from++;
case 3: *to++ = *from++;
case 2: *to++ = *from++;
case 1: *to++ = *from++;
    } while (--n > 0);
}
```

4.12.2 Псевдооператор /* FALL THROUGH */

Допускается пропуск оператора break в конце блока case, если это действительно необходимо. Но такие моменты должны быть отмечены псевдооператором /* FALL THTROUGH */ (комментарий выравненный по левой стороне case/default):

```
switch (a)
{
case 0:
    if (b > c)
        break;

/* FALL THTROUGH */

case 1:
    break;
default:
    break;
}
```

10 Подразумевается, что не смотря на синтаксическую правильность и работоспособность данной конструкции, разобраться в ней без головной боли нереально. К тому же 6-3-1, 6-4-3 и 6-4-4 MISRA C++ как бы намекают.

4.12.3 Форматирование

По правилам, указанным в 3.Форматирование, стандартный вид оператора в простейшем случае должен быть следующим:

```
switch (a)
{
case 0:
    break;
case 1:
case 2:
    break;
default:
    break;
}
```

С учетом многострочных блоков и вложенных условных операторов:

```
switch (a)
{
case 0:
    {
    //  …
    }
    break;
case 1:
case 2:
    {
    //  …
    }
    break;
default:
    {
    //  …
    }
    break;
}
```

или:

```
switch (a)
{
case 0:
    {
    //  …
        break;
    }
case 1:
case 2:
    {
    //  …
        break;
    }
default:
    break;
}
```

Следующий вариант форматирования так же возможен (отступы перед case/default):

```
switch (a)
{
    case 0:
        break;
    case 1:
        break;
    default:
        break;
}
```

4.12.4 Варианты

Обсуждаемо использование содержимого «блока» оператора if в качестве «блока» case и default:

```
switch (a)
{
case 0:
    if (condition1)
    {
        b = 1;
        c = 2;
    }
    break;
case 1:
    {
    //  …
    }
    break;
default:
    break;
}
```

4.13 Использование оператора return

4.13.1 Общие правила

В идеале функция должна иметь всего один оператор return, который находится в конце функции.

Следующее по понятности и ожидаемости место, где может находиться оператор return — это, как ни странно, самое начало функции при проверке аргументов.

Большое количество операторов return в одной функции может сделать код сложно читаемым. Следует стараться избегать расположения операторов return глубоко во вложенных логических блоках функции.

4.13.2 Возвращаемые значения

Если результат функции указатель или референс, то переменная, на которую он указывает не должна быть локальной для данной функции[11].

11 Искренне ваш, К.О.

4.14 Мёртвый код

Финальный текст программы не должен содержать код, «выключенный» из компиляции комментариями, отбитый командами препроцессора или операторами условия:

```
//  WRONG
/*
    offset += sizeof(uint32_t);
*/
…
//  WRONG
#if 0
    offset += sizeof(uint32_t);
#endif
…
//  WRONG
if (0)
{
    offset += sizeof(uint32_t);
}
```

Это же относится к коду, который никогда не будет выполнен/вызван.

5 Использование конструкций языка «Objective-C»

Замечание: здесь и ниже идет речь о стандарте языка «Objective-C» ABI v2 в первой редакции, без всяких расширений, дополнений и разрешенных умолчаний.

5.1 Общие правила

Общие правила, описанные в 4. Использование конструкций языка «C» так же справедливы и для данного раздела. Здесь будут отдельно отмечены особенности оформления исходного текста на языке программирования “Objective-C”, особенно в свете обновленных стандартов и возможностей ABI v2.

5.2 Использование пробелов

Небольшие дополнения к правилам языка «C»:

- при описании метода между “+”/“-” и типом метода ставится пробел
- при описании метода между типом метода и именем пробел не ставится
- при описании метода между “:”, типом аргумента и самим аргументом пробелы не ставятся

5.3 Описание классов

Описание каждого класса должно находиться в отдельном файле-заголовке, кроме случаев упомянутых особо. Например когда класс является «подчиненным» и приватным для определенной части приложения, или описано несколько «родственных» классов, унаследованных от общего родителя (так же напр. см. 2.2. Именование файлов).

5.3.1 «Тело» класса

Правила именования классов и членов класса (ivars) описаны в 2. Именование. Описание классов должно иметь «тело» (даже, если класс не имеет ivars, «тело» должно быть обозначено фигурными скобками) — код должен быть совместим с первой редакцией ABI v2 и не иметь умолчаний.

Категорически приветствуется расстановка ключевых слов, управляющих доступом к ivars:

- @protected для ivars, доступ к которым планируется осуществлять из классов-наследников;
- @private для ivars, доступ к которым осуществляется только через проперти или не предусмотрен;

- @public для редкого случая, если планируется доступ по указателю (да, подобное использование ivars **не запрещено**, если вы отдаете себе отчет в том, что делаете);

5.3.2 Properties класса

После описания «тела» класса следует список пропертей. Никогда не забывайте тщательно продумать атрибуты каждой проперти.

Используйте atomic атрибут (**внимание!** Здесь: отсутствие ключевого слова nonatomic, само ключевое слово atomic является расширением синтаксиса «Objective-C» ABI v2 и его использование запрещено) только в том случае, если уверены, что к ним возможен конкурентный доступ.

Если доступ к проперти снаружи подразумевает только чтение, используйте атрибут readonly.

Используйте атрибут copy только в том случае, если уверены, что это необходимо.

В самой реализации класса, для каждой определенной в описании класса проперти, должен быть соответствующий @synthesize или @dynamic. Их пропуск запрещен. Для каждой @synthesize проперти обязан присутствовать ivar, полагаться на автоматическое синтезирование ivar запрещено — код должен быть совместим с первой редакцией ABI v2 и не иметь умолчаний. Таким образом любые умолчания при использовании @synthesize запрещены.

Пример, как общее следствие замечаний выше и правил именования:

```
@class HGDBase
{
@private
    NSString* _name;
}
@property (nonatomic, retain) NSString* name;
@end
```

Использование какого-либо умолчания одной из модификаций ABI v2:

```
@implementation HGDBase
//  WRONG

…
@end
```

```
@implementation HGDBase
//  WRONG
@synthesize name;

…
@end
```


Без использования умолчания:

```
@implementation HGDBase
//  CORRECT
@synthesize name = _name;
…
@end
```

Разрешено именование пропертей с префиксом “_”:

```
@class HGDBase
{
@private
    NSString* _name;
}
@property (nonatomic, retain) NSString* _name;
@end
```

Единственное возможное умолчание в @synthesize:

```
@implementation HGDBase
//  CORRECT
@synthesize _name;
…
@end
```

5.3.3 Методы класса

Примечание: методы класса — здесь не делается различия class methods vs. instance methods, для данного документа это безразлично кроме случаев рассмотренных отдельно.

Каждый метод класса, доступ к которому осуществляется снаружи, должен быть описан в заголовке в описании класса. Исключения могут быть сделаны для «перегруженных» методов (если вообще такое понятие корректно для Objective-C). При этом методы-инициализаторы описываются всегда, а dealloc никогда (так сложилось исторически).

Имена должны быть значимы и понятны при чтении. Имена начинаются с маленькой буквы и могут быть составными, это же правило верно для каждой компоненты имени метода.

5.3.4 Приватные проперти и методы класса

Приватные проперти и методы класса должны описываться в категории-расширении (анонимная категория) в файле-реализации класса. Использование именованной категории для описания приватных методов не приветствуется. Так же там описываются «перегруженные» public-проперти.

Имена приватных методов могут начинаться с “_”.

5.4 Реализация классов

Реализация каждого класса должна находиться в отдельном файле, кроме случаев упомянутых особо. Например когда класс является «подчиненным» и приватным для определенной части приложения, или описано несколько «родственных» классов, унаследованных от общего родителя (так же напр. см. 2.2. Именование файлов).

Первыми в реализации находятся методы инициализации класса и dealloc.

Все последующие методы группируются по логике работы, при этом каждая группа отделяется прагмами с формальным описанием группы:

```
#pragma mark -
#pragma mark ** Public accessors **
#pragma mark -
```

Рекомендуется начинать с группы внешних методов, затем методов доступа к пропертям класса. Далее следуют «перегруженные» методы класса-родителя (сгруппированные по родителю), реализации делегируемых протоколов (сгруппированные по протоколам). Последними группами являются приватные методы и прочие методы экшинов и нотификаций.

Каждый не-«перегруженный» метод (внешний или приватный) должен иметь описание в классе или в приватной категории.

5.5 Категории

В простейшем случае описание и реализация категории не отличается по своей сути от класса, с поправкой на то, что у категории не может быть ivars. За исключением использования анонимных категорий-расширений для описания приватных сущностей классов приложения (см. выше), использование категорий имеет смысл только для расширений стандартных классов API.

Здесь следует заметить следующее: никогда не переопределяйте поведение стандартного класса с помощью категории. Мало кто отдает себе отчет о последствиях подобного архитектурного решения.

В категориях-расширениях для стандартных системных классов используйте имена методов с уникальным префиксом, чтобы избежать возможных конфликтов с системными именами.

Замечание: для добавления псевдо-ivars в существующий класс через категорию, можно пользоваться пропертями и objc_getAssociatedObject/objc_setAssociatedObject.

Это замечание так же намекает на запрет использовать новейшую модификацию синтаксиса Objective-C ABI v2, которая позволяет вставлять ivars в анонимную категорию-расширение.

5.6 Протоколы

Правила именования протоколов описаны в 2. Именование. Имя протокола, содержащего делегированные методы, должно совпадать с именем класса-хозяина и иметь суффикс «Delegate» (или «DataSource», или иной, в зависимости от смысловой нагрузки).

Категорически приветствуется расстановка ключевых слов, управляющих обязательностью наличия метода протокола в реализации класса — @required и @optional. Для методов, которые обязательно должны быть реализованы, ключевое слово @required обязательно.

5.7 Вызов методов

5.7.1 Единое правило[12]

Очевидно, что можно вызывать только существующие методы. Проблема с пониманием этого правила начинается там же, где начинается использование пропертей. Поэтому имеет смысл перефразировать:

Разрешено вызывать только те методы, которые существуют в описании класса, категории, или протокола. Запрещено прямо вызывать метод, являющийся аксессором для описанной в классе или категории проперти.

12 Игнорирование 5.7.1 и 5.8.1 скорее всего обернется против вас жестокими карательными мерами со стороны руководства

Пример:

```
@class HGDBase
{
    int _state;
}
@property int state;
@end
```

```
@implementation HGDBase
@synthesize state = _state;

…
    //  WRONG
    [self setState:0];

    //  CORRECT
    self.state = 0;
…

@end
```

5.7.2 Форматирование и общие правила оформления

Единственное отличие форматирования вызова метода заключается в подворачивании длинной строки. В этом случае осуществляется жесткий перенос на новую строку, причем все аргументы должны быть выравнены по символу «:».

```
[self callingVeryLongMethodForIdentifier:identifier
                        withModificator:modificator
                              andParams:params
                                mutable:NO];
```

Для форматирования вызова метода со списком аргументов действуют правила, аналогичные форматированию вызова функции «C».

Вложенность вызовов не регламентируется. Однако всегда необходимо сохранить читаемость и структурированный перенос вложенных методов на новые строки может в этом помочь.

5.8 Обращение к проперти

5.8.1 Единое правило

Очевидно, что можно обращаться только к существующей проперти. Для особо одаренных следует повторить — **если проперти нет в описании класса, вызывать ее запрещено. Тем более спекулировать на вызове метода, совпадающего по имени с аксессором проперти.**

Пример:

```
    NSArray* array = [[NSArray alloc] initWithCapacity:0];
…
    //  WRONG
    if (array.count > 0) …
…
    //  WRONG
    array.release;
…
```

5.8.2 Обращения в инициализторе класса и dealloc

Необходимо раз и навсегда для себя запомнить: обращение к проперти в инициализаторе не рекомендуемы, а в dealloc строго запрещены.

Используйте обращение напрямую к ivars.

5.8.3 Обращение внутри класса

Необходимо осознавать, что каждое обращение к проперти формально говоря является вызовом метода-аксессора, что сказывается на производительности. Поэтому без особой необходимости (например см. ниже) избегайте обращения к пропертям и используйте ivars. Это практически всегда справедливо для чтения простых @synthesize nonatomic пропертей без «перегруженных» аксессоров. Вообще рекомендуется смотреть на синтаксис с точкой, как на своеобразный изощренный вызов метода и делать соответствующие выводы о производительности.

5.9 Форматирование блоков

К сожалению единого мнения о форматировании блоков-замыканий до сих пор не существует. Для блоков, являющихся аргументами при вызове метода, необходимо стараться сохранять читаемость кода находя компромисс между 5.6.2 Форматирование и общие правила оформления и форматированием содержимого блока по правилам из настоящего документа.

Предлагается следующий вид форматирования блоков внутри вызовов методов:

```
[UIView animateWithDuration:.3
                 animations:^
                       {
// …
                       }
                  completion:^(BOOL finished)
                       {
// …
                       }];
```


5.10 Управление памятью (классический случай)

Замечание: здесь и ниже идет речь о стандартной модели памяти без использования ARC и GC.

5.10.1 Единое правило

Все, что создано вами, вами же должно быть и удалено. Каждому использованию метода alloc, copy, copyMutable или retain должен соответсвовать вызов release. Вообще идеология работы с памятью неплохо описана в древней документации от Apple (а то и NeXT)[13].

Передавая/добавляя/устанавливая объект в некий класс, забудьте о его судьбе — правило то же: если вы его создали, релизните его. Класс сам должен позаботиться о том, чтобы сделать объекту retain. Простейший пример — добавление объектов в массивы или словари. Это же справедливо при установке объектов с помощью проперти с атрибутом retain или copy.

И тем не менее существуют нюансы. Вот некоторые из них:

- делегаты, провайдеры данных и прочие подобные объекты скорее всего не ретейнятся (ретейн такого объекта — потенциальный memory leak);
- различные спекуляции с autorelease (см. ниже);

5.10.2 Autorelease

Создавая локальные объекты, делайте им autorelease, чтобы не заботиться о их судьбе в текущем контексте. Однако следите за тем, чтобы их время жизни случайно не пересеклось в вызовом какого-нибудь NSRunLoop.

Метод, возвращающий созданный внутри себя объект, именованный без упоминания слов create, new или copy (обычно как префикс имени метода), всегда должен возвращать autorelease объект. В случае необходимости retain ему сделает вызывающая сторона, и это не является заботой метода.

Метод, возвращающий созданный внутри себя объект, именованный с упоминанием слов create, new или copy (обычно как префикс имени метода), всегда возвращает alloc, copy, copyMutable или retain (один раз) объект. В этом случае вызывающая сторона должна делать этому объекту release или autorelease. Создание и использование подобных методов является дурным тоном и категорически не приветствуется.

Если в определенном месте вам необходимо создать множество тяжеловесных объектов, жизнь которых вам в последствии не дорога, при этом связываться с alloc/retain нудно, а памяти жалко, используйте NSAutoreleasePool. Это так же может быть справедливо для «тяжелых» циклов.

5.10.3 Легкий путь попасть в ад

Легкий путь попасть в ад (в очень специальный ад) — это пытаться обмануть менеджмент памяти добавляя лишние retain и/или release, или делать какие-то выводы по содержимому retainCounter. Обычно явно хватает использования только одного retain

13 https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/MemoryMgmt.pdf

в конкретном контексте, остальное скрыто под 5.8.1. Единое правило.

Если на определенный момент выполнения программы, память объекта не освобождается по release, как вы ожидаете, это значит, что либо объект все же кому-то еще нужен, либо, что вы создали себе проблему с лишними retain'ами. Не следует пытаться судорожно освободить память волюнтаристическими методами, разбритесь, кто держит этот объект.

5.11 Управление памятью (использование ARC)

Мы не используем ARC и в силу определенных причин не будем его использовать вплоть до особого указания. В случае использования в приложении стороннего кода, написанного с использованием ARC, добавленные файлы просто помечаются для компиляции с ARC.

5.12 Управление памятью (использование GC)

Никто не пользуется CG.

6 Использование конструкций языка «C++»

В силу определенных причин[14], эта глава будет еще долго находиться в процессе написания.

6.1 Общие правила

Общие правила, описанные в 4. Использование конструкций языка “C” так же справедливы и для данного раздела.

6.2 Использование пробелов

Небольшие отличия от правил языка «C»:

- указатель отделяется одним пробелом от типа в декларациях переменных
- референс не отделяется от типа в декларациях переменных
- таким образом между референсом и указателем в декларациях переменных всегда ставится пробел
- указатель и референс отделяются от имени переменной в декларациях минимум одним пробелом

```
int *       p;
int& *      p;
```

14 Никто кроме автора настоящего документа не пишет на C++ для продакшина.

7 Использование расширения «Objective-C++»

Формально говоря — это ад. Но из-за ущербности Objective-C иногда имеет смысл архитектурно разделить приложение на две части: функциональную (написанную на C++ и осуществляющую основную обработку данных) и интерфейсную (написанную на Objective-C и осуществляющую отрисовку пользовательского интерфейса). Связывающим звеном между двумя этими частями как раз и будет являться прослойка, написанная на Objective-C++. Или как вариант, на Objective-C++ пишется интерфейсная часть.

Вообще, сильно упрощая, Objective-C++ — это Objective-C в котором, помимо всего прочего, можно ограниченно использовать доступ к C++ классам. Это как тени на стене — части складываются в общую картину, но друг с другом никак не взаимодействуют. Ад начнется, когда вы задумаетесь о том, как работают статические конструкторы C++. И деструкторы.

8 Среда разработки (для языка «C» и производных).

Для разработки под iOS и Mac OS X мы пользуемся Xcode. Вообще, за редким исключением, все пользуются Xcode. Это вынужденное зло и с этим ничего не поделать.

8.1 Настройка форматирования

Установки форматирования текста в Xcode должны быть поставлены в соответствие с 3. Форматирование и общие правила оформления.

8.2 Использование сниппетов и copy/paste кода

В Xcode при создании нового файла для класса или метода по автокомплиту, вставляется преформатированный код. Его всегда необходимо **вычитать** и **привести к единому стилю**, описанному в данном документе. Лишний код и комментарии должны быть удалены. Это же относится к copy/paste коду из других проектов, в особенно от 3rd party разработчиков.

8.3 Предупреждения

Для всех проектов в Xcode необходимо включать предупреждения (warnings). Причину получения тех или иных предупреждений необходимо осмыслить, исправить и больше так никогда не делать.

Чтобы было совсем понятно: в Xcode необходимо включить настройку «treat all warnings as error».

8.4 «Идеальный» случай

Ваш код должен собираться любым компилятором «Objective-C» ABI v2. Это значит, что он может быть собран не только в лунную безветренную ночь в Xcode 4.5 с последним сферическим clang для кухарок, но и в Xcode 4.2 с человеческим clang, а так же в llvm-gcc.

Для llvm-gcc можно сделать исключение, если в силу каких-то причин в приложение попал 3rd party код написанный с использованием ARC.

8.5 3rd party код и использование готовых библиотек

Решение, о возможности использовать тот или иной код стороннего производителя, а так же библиотек, принимается руководителем проекта. В сомнительных случаях лицензионной чистоты решение принимает руководство компании.

9 Общие слова

Assumptions are evil[15]**.** Полагаться на что-то без точного представления о механизмах работы — недопустимо. Именно поэтому мы не пользуемся правилами умолчания, ARC и аналогичными модными вещами навязываемыми нам Apple исключительно из рекламных соображений, и вызывающими оргастическое состояние у ПТУшников и «кодеров» от сохи. Мы взрослые и грамотные люди, давайте себя уважать.

Никогда не верьте компилятору. Ошибка в компиляторе редка и, положа руку на сердце, она практически невозможна (тем не менее это утверждение истинно только для конца прошлого века и первого десятилетия нынешнего). Но, доверяясь компилятору, вы рискуете, в лучшем случае, написать неоптимальный код[16]. Никакие алгоритмы оптимизации не исправят клинический идиотизм человека. Не пытайтесь обмануть компилятор усложнениями решения своей задачи и вбиванием костылей в алгоритмы. В конце концов, когда вам начнет казаться, что в компиляторе ошибка, это значит только одно — вы проиграли битву за звания альфа-самца бездушной машине и обманули сами себя.

Будьте проще. Простые решения эволюционируют и превращаются в сложный неподъемный код. Избегайте ненужных усложнений, пока это возможно. Если вы почувствовали, что завязли в болоте, лучше пересмотреть решение и все сделать заново. Именно сделать, а не переписать ущербное решение иными словами. Однако не злоупотребляйте переписыванием — программирование ради самого процесса никому не нужно. Нужен результат. Причем в срок.

Будьте самокритичными. Все, что мы пишем, мы можем написать лучше. Это аксиома, поэтому думайте о своем коде скептически и пообещайте себе в следующий раз сделать все лучше и правильнее. Самое главное — не забудьте об этом обещании.

15 В контексте программирования фраза принадлежит некоему Laurence Harris из SkyTag Software, Inc. В начале этого века он знатно раздавал пинки неофитам в CarbonDev рассылке от Apple.

16 AV Rule 216 запрещает попытки заранее найти и оптимизировать узкие места разрабатываемого алгоритма или куска программы. Но в то же время дает оговорку, что это не относится к фундаментальным алгоритмам и структурам данных. Похоже, что они сами не знают, что хотят запретить, а их цитата Д. Кнута о том, что мол де «ранняя оптимизация — корень всех бед», вообще не выдерживает никакой критики. При всем моем уважении к Кнуту, я поверю ему только в тот день, когда буду держать в руках TAOCP том V.

