Разработка стандартов кодирования для языка C++

А. А. Михайлов, Д. В. Соловьёв

Работа посвящена вопросам разработки стандартов оформления программных продуктов на уровне модуля. Показано, как на основе дерева требований был создан набор правил кодирования. Объяснено появление тех или иных правил в этом наборе. Приведен разработанный авторами стандарт кодирования компании Cognitive Technologies.

1. Введение

В течение жизненного цикла программы большинство файлов с исходным текстом часто «трогают». Их читают, отлаживают, профилируют, правят, добавляют новые строки и т. п. В этом легко убедиться, заглянув в систему контроля версий (SourceSafe, CVS и т. п.) и просмотрев историю версий большинства файлов с исходным текстом. Поэтому возможность удобной работы с кодом является даже более важной, чем удобство его создания.

В работе [1] была приведена методология создания стандарта кодирования. Здесь же будет показано, как на основе этой методологии был создан такой стандарт для использования в компании Cognitive Technologies.

2. Цель статьи

Целью статьи является описание набора правил кодирования, который позволяет соблюсти баланс между двумя условиями:

- облегчает работу с кодом, созданным с использованием этих правил;
- не сильно затрудняет создание нового кода.

3. Требования к стандарту кодирования

Требования к правилам кодирования распадаются на два набора. Это происходит потому, что цель состоит из двух условий. Первый набор требований касается качества получаемого кода. Он был рассмотрен

подробно в предыдущей работе авторов [1]. Эти требования будут кратко описаны в следующем разделе. Второй набор относится к эргономике написания кода по стандарту.

3.1. Требования к коду

Первое требование — **ясность**. Хороший код быстро читается и легко понимается. Текст программы должен претворять в жизнь любимый лозунг программистов: «Лучшая документация к программе — это ее код».

Это требование, при проецировании его на уровень модуля, распадается на два: **структурированность и информативность**. Под **структурированностью** в приложении к рассматриваемой модели авторы понимают создание такого кода, в котором хорошо просматривается его структура. Это означает, что все важные элементы видны, и их не приходится выискивать в тексте программы. Хорошая структурированность позволяет уменьшить сложность программы. **Информативность** кода — тоже важное требование. Оно означает, что можно легко узнать, зачем нужен каждый объект программы, как он взаимодействует с другими объектами, и что из этого получается.

Второе требование — **развиваемость.** Часто бывает необходимо добавить новую функциональность в уже написанную и отлаженную программу. Программа должна быть написана так, чтобы ради этого не требовалось переделывать ее всю. Решить эту проблему только на уровне кода не всегда возможно. Но к этому надо стремиться. Это требование состоит из трех требований более низкого уровня. **Изолированность** означает, что все логически самостоятельные части выделены, и связи между ними минимизированы. **Полнота (цельность)** предполагает, что любая идейно цельная часть программы является законченным объектом. Этот объект должен иметь достаточный набор выразительных или дедуктивных средств для описания всех его реальных свойств и интерфейсов взаимодействия. **Ограниченность** означает, что любая представляющая потенциальный интерес часть программы не должна быть слишком большой и сложной.

Следующее требование — **поддерживаемость.** Поддерживаемый код, во-первых, должен удовлетворять требованию ясности. Во-вторых, любое небольшое разумное действие разумного, но не слишком опытного человека не приводит к катастрофическим последствиям для программы. Требование поддерживаемости разделяется на два: **автоконтроль** и **точная трактовка. Автоконтроль** обеспечивает немедленную реакцию самой программы или средств разработки, отладки и тестирования на возникновение потенциально опасной ситуации. **Точная трактовка** означает, что код делает именно то, что предполагает читающий его разработчик.

И, наконец, последнее из рассматриваемых требований к коду — требование **стабильности** программы. Оно означает, что программа работает на заявленных операционных системах и компьютерах, поведение программы предсказуемо (нет плавающих ошибок) и т. п.

3.2. Требования к набору правил

Первое требование — **экономность стандарта**. Это требование составное, оно распадается на два:

- Список правил не должен превращаться в учебник по программированию. Не нужно прописывать нормы, очевидные всем разработчикам, для которых создается стандарт. Приведем пример правила, которое не прошло в стандарт из-за этого требования:

Используйте угловые скобки для включения системных заголовков, а кавычки — для включения проектных заголовков.

- Не должно быть легковесных правил. Все правила должны давать ощутимую пользу. Слабые правила — препятствие к внедрению. Одно из правил, исключенное по этому критерию, выглядит так:

Не используйте ключевое слово void, чтобы показать, что у функции нет параметров.

Второе требование — **удобство соблюдения стандарта в процессе написания кода**. И это требование тоже составное:

- Про любую часть программы можно легко понять, попадает ли она под действие какого-либо правила. Иначе разработчику придется затрачивать много усилий на соблюдение стандарта при написании кода. Это противоречит второй части цели.
- Исключения из правил должны быть описаны.
- Правила должны быть конкретными. Должно быть понятно, как их соблюдать. Благие пожелания вроде «*имена функций должны быть осмысленными*» не конструктивны.

Последнее требование — **необременительность комплексной проверки соблюдения стандарта в модуле**.

4. Правила стандарта кодирования

В этом разделе будут описаны правила, которые вошли в стандарт кодирования, а также рассмотрены причины, почему это произошло. Правила отсортированы по требованиям, к которым они относятся.

4.1. Структурированность

4.1.1. В структурах запрещены функции, private- и protected-секции.

Структура должна быть структурой. А если в ней есть функции-члены, секции сокрытия, то это уже класс. И оформлять такую структуру следует как класс. В программе не должно быть двусмысленностей и смешения понятий.

4.1.2. На строке не больше одной ';', кроме случая цикла 'for'.

Это формализация правила о том, чтобы на одной строке было не больше, чем одно предложение языка. Рассмотрим следующий фрагмент текста программы:

```
i = 0; j = 0; k = 0; DestroyBadLoopNames(i, j, k);
```

Аргументы в пользу того, чтобы писать несколько предложений языка в одной строке:

— Такой код занимает меньше строк и на экране и на бумаге. Это позволяет охватить больше кода одним взглядом.

— Это один из способов группировки связанных предложений языка.

Но аргументы в пользу выполнения данного правила нам кажутся более убедительными.

— Обеспечивается точный взгляд на сложность. Сложные предложения выглядят сложными, а простые — простыми.

— Код читается сверху вниз, а не сверху вниз и слева направо. Разыскивая нужный код, нам не надо вглядываться в строки программы.

— При отладке имеется возможность поставить точку прерывания на каждое предложение языка.

— При редактировании такой код легче удалять или комментировать.

4.1.3. Не используйте символ табуляции.

Авторы считали важным зафиксировать какой-нибудь один способ отступа и сделали выбор в пользу пробелов. Это было сделано потому, что символ табуляции интерпретируется разными приложениями по-разному, и это может привести к ошибкам выравнивания текста программы.

4.1.4. Ставьте пробел после каждой запятой.

Какой 4-й аргумент в следующем вызове функции?

```
ReadEmployeeData(MaxEmps,Team.EmpData,InputFile,
                                    Team.EmpCount,InpuError);
```

А теперь, какой 4-й аргумент в следующем вызове функции?

```
GetCensus(Team.EmpData, InputFile, Team.EmpCount,
                                    MaxEmps, InpuError);
```

Из примера видно, что код с пробелами легче читать.

4.1.5. Выделяйте пробелом справа и слева все бинарные операции.

Выражение

```
while((PathName[StartPath+Pos]!=';') &&
     ((StartPath+Pos) <= strlen(PathName)))
```

читается так же плохо, как Idforyoutoreadthis. Добавление же пробелов в это выражение делает текст программы более наглядным.

```
while((PathName[ StartPath + Pos ] ! = ';' ) &&
     ((StartPath + Pos ) <= strlen(PathName )))
```

4.2. Информативность

4.2.1. Все комментарии на русском языке.

Компания, для которой создавался описываемый стандарт, находится в России и работает на российском рынке. Использование родного языка облегчает разработчикам как понимание, так и составление комментариев. Да и тексты комментариев, составленных людьми, которые не сильны в используемом иностранном языке, бывает трудно понять даже людям, знающим этот язык (иногда, по прошествии времени, и самому автору).

4.2.2. Опишите файл — дату создания, назначение и автора.

4.2.3. Опишите функцию — решаемую задачу, параметры вызова, возвращаемое значение.

К этому правилу авторы сделали два важных дополнения. Первое касается того, где именно комментировать функцию. Было решено, что используемый интерфейс модуля и классов должен описываться в заголовочных файлах по месту объявления функций, составляющих этот интерфейс. Ведь для того, чтобы использовать эти функции, нужно к своим модулям подключить заголовочный файл с объявлениями.

Статические и private-функции — члены класса было решено описывать по месту их реализации. Не следует перегружать лишней информацией пользователя внешнего интерфейса модуля.

Второе дополнение вызвано тем, что некоторые функции и/или их параметры описывать бессмысленно и даже вредно. Были выделены следующие объекты, которые не требуют описания:

- Можно вообще не описывать короткие (менее 10 строк) статические функции или private функции-члены.
- Можно вообще не описывать функцию, переопределяющую стандартную, указав, где можно ознакомиться с ее описанием. Например: // см. MSDN.
- Можно не описывать назначение функции, если оно очевидно из ее названия.
- Можно не описывать параметр функции, назначение которого очевидно из ее названия и типа.
- Можно не описывать возвращаемое значение, если его назначение очевидно из его типа и/или имени.

4.2.4. Опишите управляющие структуры с телом больше одного предложения.

Управляющие структуры увеличивают сложность программы, так как они нарушают последовательное выполнение команд. Поэтому комментарии в таких местах важны для понимания алгоритмов работы программы.

4.2.5. Опишите все статические переменные.

4.2.6. Опишите все переменные внутри структуры.

4.2.7. Опишите все переменные — члены класса.

Все «долгоживущие» и многократно используемые данные должны быть описаны. Это важно как для понимания алгоритмов выполнения программы, так и во избежание нежелательного дублирования данных.

Далее будут приведены шесть правил, касающихся соглашений по именованию. Такие соглашения предоставляют несколько преимуществ:

— Принимается одно глобальное решение об именовании, а не много локальных. Это позволяет сосредоточиться на более важных характеристиках кода.

— Сходство в именах дает разработчику возможность легко угадать, для чего нужна незнакомая переменная или тип. И вообще понять, что это за объект.

— Соглашения уменьшают вероятность возникновения ситуации, когда одна и та же вещь называется разными именами.

И самое главное — любое соглашение лучше, чем отсутствие такового. Сила соглашения об именовании происходит не оттого, что выбрано какое-то определенное соглашение, а из самого факта его существования.

4.2.8. Все буквы в именах константных переменных — заглавные.

Выделяя имена констант, разработчики выделяют те переменные, которые не могут поменять свое значение в ходе выполнения программы (авторы не обсуждают различные трюки вроде const_cast). Использование в именах констант заглавных букв — это общая практика в сообществе «C++»-разработчиков.

4.2.9. Все буквы в именах макросов — заглавные.

Позволяет различать имена функций и макросов. Это важно, так как, несмотря на похожий синтаксис вызова, поведение функции и макроса в программе могут сильно различаться. Использование в именах макросов заглавных букв — это общая практика в сообществе «C++»-разработчиков.

4.2.10. Начинайте с 'C' имена классов.

4.2.11. Начинайте с 'p' имена указателей.

4.2.12. Начинайте с '`s_`' имена статических не константных переменных.

4.2.13. Начинайте с '`m_`' имена не статических не константных переменных — членов класса.

Эти правила об именовании были выбраны потому, что они распространены среди разработчиков организации, для которой создан данный стандарт.

4.2.14. Начинайте с префикса `::` вызовы всех глобальных (не статических) функций.

Это правило задумано для легкости идентификации глобальных функций и защиты их от перегрузки (overriding) аналогично названными функциями из локального пространства имен или функциями-членами.

4.2.15. Не используйте конструкцию '?:'.

Эта конструкция трудна для чтения. Приходится останавливаться на строке с таким оператором и задумываться о том, что и при каком условии произойдет.

4.2.16. Выделяйте скобками логические части логического выражения.

Это позволяет читателю легко понять порядок выполнения логического выражения.

4.3. Изолированность

4.3.1. Все функции, имеющие объявления в файлах c/cpp — статические.

Фактически здесь два правила. Первое гласит, что все функции, используемые только в данном модуле, должны быть объявлены как статические. Это позволит избежать конфликта имен.

Второе правило говорит о том, что все описания глобальных функций должны быть в заголовочном файле. Если требуется использовать глобальную функцию в модуле, следует подсоединить к этому модулю заголовочный файл с описанием этой функции. Запрещено явно вставлять в модуль (cpp-файл) описание глобальной функции. Это позволяет выделить внешний интерфейс модуля и облегчить работу в случае исправления этого интерфейса.

4.3.2. Все глобальные переменные — либо статические, либо константы.

Использование не константных глобальных переменных ведет к следующим проблемам:

— Случайные изменения глобальных данных. Вы изменяете глобальную переменную в одном месте и ошибочно предполагаете, что она остается неизменной где-то еще. Рассмотрим следующий фрагмент кода:

```
g_TheAnswer    = GetAnswer();
OtherAnswer    = GetOtherAnswer();
AverageAnswer  = (g_TheAnswer + OtherAnswer) / 2;
```

Вы, вероятно, предполагаете, что вызов GetOtherAnswer не изменяет значение g_TheAnswer. Если это не так, то третья строка программы даст неправильный результат.

— Забавные проблемы с псевдонимами (ссылками). Например,

```
int g_GlobalVar = 1;
void WriteGlobal(int & InputVar)
{
    g_GlobalVar = InputVar + 1;
    printf("InputVar =    %d \n", InputVar);
    printf("g_GlobalVar = %d \n", g_GlobalVar);
}
```

Теперь представим, что где-то есть такой вызов:

```
WriteGlobal(g_GlobalVar);
```

Глядя на реализацию функции `WriteGlobal`, можно предположить, что `g_GlobalVar` будет на 1 больше `InputVar`. На самом деле результат исполнения будет:

```
InputVar =    2
g_GlobalVar = 2
```

— Возможные проблемы при работе в многопоточном режиме. Представим себе, что один поток поменял глобальную переменную и собирается воспользоваться ее значением. В этот момент другой поток получает управление и меняет ее по-своему.

— Иногда глобальные данные мешают повторному использованию кода. Если мы хотим взять подпрограмму из одной программы и вставить ее в другую, то использование в ней глобальных данных может усложнить проблему.

Все вышеперечисленное затрудняет понимание логики исполнения программы.

С другой стороны, существуют случаи, когда использование глобальной переменной оправдано.

— У вас есть данные, которые относятся ко всей программе. Это может быть переменная, отражающая состояние программы. Например, отладочный или нормальный режим работы. Или какие-нибудь таблицы данных.

— Именованные константы.

— Иногда бывают переменные, ссылки на которые нужны в списке параметров почти каждой функции, которую вы пишите. Удобнее завести одну глобальную переменную, чем включать ее в каждый список параметров.

— Бывает, что нужно передать данные из одной функции в другую через длинный стек вызовов. Функциям в середине этой цепочки эти данные не требуются. Но они вынуждены передавать эти данные дальше. Глобальные переменные позволяют устранить это неудобство.

Поэтому использование глобальных переменных на уровне модуля допустимо. При этом рекомендуется использовать функции для доступа к таким переменным.

4.3.3. Не делайте переменные — члены класса public.

Обоснования аналогичны приведенным для предыдущего правила.

4.3.4. Не используйте в функциях числовые значения («волшебные числа»), кроме 0 и 1.

Использование значимых имен вместо числовых значений имеет следующие преимущества:

- Изменения могут быть произведены с большей степенью надежности. Если вы используете именованную константу, вы не проглядите одну из констант 100 или не поменяете число 100, которое имеет отношение к чему-то другому.
- Изменения легче сделать. Если вы хотите изменить 100 на 200, то вам надо найти все нужные сотни и поменять их. Кроме того, если вы используете $100 + 1$ или $100 - 1$, то надо не забыть поменять и их. В случае именованной константы вы меняете 100 на 200 в одном месте.
- Код становится более понятным. Можно только догадываться, что означает 100 в выражении

```
for (i = 1; i < 100; i++)
```

Но в случае

```
for (i = 1; i < MaxEntries; i++)
```

все понятно. Даже если число никогда не будет меняться, вы получите выигрыш в ясности кода.

Что касается чисел 0 и 1, то они часто используются как номер первого элемента массива, как инициализаторы начала цикла и т. п.

4.3.5. Не используйте механизм friend.

В большинстве случаев использование friend-функций — это признак неполноценного интерфейса класса. Этот механизм нарушает принципы сокрытия данных.

Но friend необходим, когда требуется получить доступ к скрытым данным класса:

- Для заранее определенного ограниченного круга пользователей. Это именно симбиоз, а не латание ошибок проектирования.
- Без пересборки кода, его реализующего.

4.4. Полнота

4.4.1. Всегда вставляйте умолчательные конструктор и деструктор.

Во-первых, разработчик лишний раз подумает, все ли в порядке с конструктором и деструктором. Ведь ошибки в инициализации могут привести к трудно находимым ошибкам в программе. Во-вторых, эти конструктор и деструктор могут понадобиться разработчику позже. То, что они уже есть, облегчит работу по поддержке.

4.4.2. Всегда вставляйте конструктор копирования.

4.4.3. Всегда вставляйте operator =.

Проблемы с конструктором копирования или оператором присваивания возникают, если побитного копирования недостаточно. Это всегда случается, когда класс сам берет память динамически. Потенциально опасно отсутствие конструктора копирования или оператора присваивания в классах, которые имеют в качестве своих членов указатели.

Если же какой-то класс не предполагается копировать, то следует создать пустые конструктор копирования и оператор присваивания, и поместить их в private секцию. Это защитит класс от несанкционированного копирования.

4.5. Ограниченность

4.5.1. Запрещены тройные указатели.

Использование таких «тяжелых» конструкций увеличивает сложность кода и, следовательно, сильно затрудняет его понимание. Даже с двойными указателями довольно сложно разбираться, но они часто используются для обеспечения реализации специальных интерфейсов (например, COM).

4.5.2. Максимальная вложенность управляющих структур не больше 3.

Исследования [2] показали, что людям трудно удержать в голове больше трех уровней вложенности if-ов. Поэтому им приходится часто возвращаться назад для того, чтобы понять логику выполнения программы. Кроме того, вложенные управляющие структуры увеличивают сложность программы, так как растет количество вариантов ее возможного выполнения.

4.5.3. Файл не длиннее 1 000 строк.

Файл должен быть обозримым. Но на практике бывают случаи, когда файл может получиться и большего размера. В данном стандарте авторы пошли по следующему пути: правило о 1 000 строк сделали рекомендательным, а в случае превышения размера потребовали, чтобы была описана причина этого. Так стандарт стимулирует разработчика создавать небольшие файлы.

4.5.4. Любая функция не длиннее 180 строк.

Исследования по поводу эффективного размера функций довольно противоречивы [2]. Но из них следует, что требовать, чтобы функции были маленькими почти так же плохо, как и разрешать им быть слишком большими. Поэтому для верхнего порога была выбрана цифра 180, которая приблизительно соответствует пяти страницам программного текста.

4.5.5. Строки не длиннее 100 символов («ширины экрана»).

Для просмотра строк большей длины приходится использовать горизонтальный `scrollbar`. А если код трудно читать, то его трудно понять.

4.5.6. Циклы, блоки `if` и блоки `else` не должны быть длиннее 36 строк («высоты экрана»).

Такой размер позволяет увидеть всю управляющую структуру целиком.

4.6. Автоконтроль

4.6.1. Каждое предупреждение компилятора (кроме 4786) с уровнем диагностики ошибок 'Level3' отметьте и объясните перед соответствующей строкой кода.

Предупреждение компилятора часто означает потенциальную ошибку в программе. Кроме того, в лесу безобидных предупреждений может затеряться важное.

4.6.2. Используйте `const`, а не `#define`, при описании констант.

Использование типизированных переменных вместо макросов позволяет компилятору осуществлять контроль типов. Это также гарантирует, что символьное имя переменной будет доступно во время отладки.

4.6.3. Используйте C++-преобразование типов, а не C.

С C-преобразованием типов существуют две проблемы. Во-первых, оно преобразовывает все, что угодно, во все, что угодно, используя одинаковый синтаксис. А ведь существует большая разница между преобразованием неконстантного объекта в константный объект и преобразованием указателя на базовый класс в указатель на наследованный класс. Вторая проблема состоит в том, что C-преобразование тяжело найти в коде программы. Синтаксически оно состоит из скобок и идентификатора, а скобки и идентификаторы используются в C++ везде. Иногда даже бывает трудно ответить на вопрос, а используется ли вообще преобразование типов в этой программе.

C++-преобразование типов нагляднее. Кроме того, наличие нескольких разноцелевых операторов позволяет более точно понять, зачем это преобразование было сделано. А наличие долгого в написании синтаксиса побуждает лишний раз подумать о том, нужна ли здесь вообще эта потенциально опасная операция.

4.6.4. Используйте `"const"` для защиты немодифицируемых параметров функций, передаваемых по указателю или ссылке.

Использование `const` информирует разработчика и компилятор о том, что значение параметра не может быть изменено. При попытке изменить этот параметр компилятор выдаст ошибку.

4.6.5. Если функция — член класса не вносит изменения в этот класс, используйте ключевое слово `"const"`.

Использование `const` информирует разработчика и компилятор о том, что объект данного класса не может быть изменен этой функцией-членом. За этим следит компилятор.

4.6.6. Всегда включайте в `switch` не пустой `default`.

4.7. Точная трактовка

4.7.1. Не используйте присвоения в логических выражениях.

Использование оператора присваивания в логических выражениях затрудняет понимание кода. Это связано с тем, что символ оператора присваивания (=) является частью оператора сравнения (= =). Поэтому, глядя на выражение вроде if (a = f()), не ясно, описка это или нет.

4.7.2. Используйте пару new и delete вместо пары malloc и free.

При использовании malloc и free не вызываются конструктор и деструктор. Кроме того, опасно смешение этих пар. new/delete и malloc/free используют различные методы выделения памяти и поэтому не взаимозаменяемы [3].

4.7.3. Не обращайтесь к глобальным переменным из конструктора.

Нарушение правила может привести к ошибкам. Например:

```
// Hen.h
   class CEgg;
   class CHen
  {
    public:
        CHen();
        ~CHen();
        void makeNewHen(CEgg*);
        // ...
  };
  // Egg.h
  class CEgg { };
  extern CEgg FirstEgg; // определено в Egg.cpp
  // FirstHen.h
  class CFirstHen : public CHen
  {
    public:
        CFirstHen();
        // ...
  };
  extern CFirstHen FirstHen; // определено
  в FirstHen.cpp
  // FirstHen.cpp
  CFirstHen FirstHen; // вызов CFirstHen::CFirstHen()
  CFirstHen::CFirstHen()
  {
    // В этом конструкторе может оказаться ошибка,
    так как // FirstEgg глобальный объект и возможно
    еще не существует когда // инициализи-
```

```
    руется FirstHen.
    // Так что же раньше~--- яйцо или курица?
    makeNewHen(&FirstEgg);
}
```

4.7.4. Не вызывайте виртуальные функции из конструкторов и деструкторов.

Так как не очевидно, функция какого класса будет вызвана, то лучше воздержаться

4.8. Стабильность

4.8.1. Преобразуйте указатель на базовый класс в указатель на наследованный класс с помощью `dynamic_cast`.

Во время компиляции не всегда известно о том, можно ли преобразовать указатель на базовый класс в указатель на наследованный класс. Оператор `dynamic_cast` — единственный преобразователь типов, который может осуществить такую проверку. Отказ же от проверки может привести к тяжелой ошибке.

4.8.2. Не инициализируйте глобальные переменные глобальными переменными.

В C++ не задан порядок инициализации глобальных переменных, определенных в разных модулях. Поэтому в момент инициализации одной переменной другая может быть еще не инициализированной.

4.8.3. Используйте виртуальные деструкторы в базовых классах.

В C++ в случае, если вы пытаетесь уничтожить объект наследованного класса через указатель на базовый класс, который не имеет виртуального деструктора, результат действия не определен.

4.8.4. Явно инициализируйте переменные.

4.8.5. Явно инициализируйте все члены класса во всех конструкторах.

4.8.6. Присваивайте значения всем членам класса в operator =.

Неинициализация или неправильная инициализация данных — один из самых распространенных источников ошибок в программировании. Дело в том, что неинициализированные переменные содержат значения, которых вы не ожидаете. Такие ошибки часто труднонаходимы, и профилактика может сэкономить много времени, которое пришлось бы потратить на отладку.

4.8.7. Всегда проверяйте значение, возвращаемое оператором new.

Память — это ограниченный ресурс. Его может не оказаться в самый неожиданный момент. Поэтому не стоит полагаться на авось.

5. Заключение

В данной статье авторы продемонстрировали, как, используя методологию стандартизации оформления программного кода, создавать стандарт кодирования. Были учтены не только требования к самим правилам, но и требования, позволяющие «настроить» набор правил для конкретной группы разработчиков. Эти требования, конечно, у других компаний могут быть другие. И в результате получится другой стандарт кодирования.

Приведенный в статье стандарт кодирования был разработан и используется в компании Cognitive Technologies.

Литература

1. *Михайлов А. А.*, *Соловьёв Д. В.* Принципы стандартизации оформления программного кода // Документооборот. Концепции и инструментарий: Сб. трудов ИСА РАН / Под ред. В. Л. Арлазарова и Н. Е. Емельянова. М.: УРСС, 2004. С. 58–70.
2. *McConnell Steve*. Code Complete. Redmond, Wash.: Microsoft Press, 1993.
3. *Scott Meyers*. Effective C++. 2nd ed. Addison-Wesley, 1997.
4. *Scott Meyers*. More Effective C++. Addison-Wesley, 1995.
5. *Todd Hoff*. C++ Coding Standard. [Electron. resource] // Access mode: http://www.possibility.com/Cpp/CppCodingStandard.htm.
6. Geotechnical Software Services, C++ Programming Style Guidelines. [Electron. resource] // Access mode: http://geosoft.no/style.html.
7. *Gabryelski Keith*. Wildfire C++ Programming Style. [Electron. resource] // Access mode: http://www.chris-lott.org/resources/cstyle/Wildfire-C++Style.html.