Как правильно оформлять программный код на C++

Категории >> ИТ >> IT-разработка >> Программирование

От переводчика. Искал в интернете простое и легко применимое руководство по оформлению программ на C++. Мне понравился один из вариантов, и я решил его перевести и опубликовать. — Dyzzet (http://habrahabr.ru/users/di_zed/,http://twitter.com/dyzzet).

Введение

Настоящий документ содержит рекомендации по написанию программ на языке C++.

Рекомендации основаны на установившихся стандартах, собранных из различных источников, личного опыта, частных требований и потребностей определённых проектов, а также почерпнутых из источников (см. ниже).

Но для появления ещё одного списка рекомендаций, помимо указанных источников, есть несколько причин. Основная причина — их излишняя обобщённость, поскольку зачастую требуется задать частные правила (в особенности правила именования). Данный документ содержит комментарии, что делает его более удобным в использовании при проведении ревизий кода, чем другие уже существующие документы. К тому же, рекомендации по программированию обычно вперемешку содержат описания проблем стиля и технических проблем, что не совсем удобно. Этот документ не содержит каких-либо технических рекомендаций по C++, делая упор на вопросах стиля.

Имеющиеся среды разработки могут улучшить читаемость кода с помощью отображения модификаторов доступа, подсветки кода, автоматического форматирования и прочего, но программисту не следует полагаться на эти инструменты. Исходный код должен рассматриваться не только в рамках используемой среды разработки и должен быть написан так, чтобы максимально улучшить читаемость независимо от среды.

Формат документа

Рекомендации сгруппированы по темам и пронумерованы, чтобы на них можно было ссылаться во время ревизий кода.

Рекомендации отображаются следующим образом:

n. Короткое описание рекомендации.

```
// Пример кода (если возможно)
```

Объяснение, происхождение и дополнительная информация.
Комментарии к рекомендациям особенно важны, поскольку стандарты написания кода и гайдлайны обычно разжигают «холивары», и важным моментом является объяснение рекомендации.

Важность рекомендаций

Рекомендации разделены по степени важности: обязательные, настоятельно рекомендуемые и общие.

Общие рекомендации

1. Допускаются любые нарушения рекомендаций, если это улучшает читаемость.

Основная цель рекомендаций — улучшение читаемости и, следовательно, ясности и лёгкости поддержки, а также общего качества кода. Невозможно дать рекомендации на все случаи жизни, поэтому программист должен мыслить гибко.

2. Правила могут быть нарушены, если против них есть персональные возражения.

Это попытка создать набор общих рекомендаций, не навязывая всем единый стиль. Опытные программисты обычно всё равно подгоняют стиль под себя. Подобный список рекомендаций, имеющийся под рукой (или хотя бы требование ознакомиться с ним), обычно заставляет людей задумываться о стиле программирования и оценке их собственных практик в этой области.

С другой стороны, новички и неопытные программисты обычно используют рекомендации по стилю для лучшего понимания жаргона программистов.

Соглашения об именовании

3.1 Общие соглашения об именовании

3. Имена, представляющие типы, должны быть обязательно написаны в смешанном регистре, начиная с верхнего.

```
Line, SavingsAccount
```

Общая практика в сообществе разработчиков C++.

4. Имена переменных должны быть записаны в смешанном регистре, начиная с нижнего.

```
line, savingsAccount
```

Общая практика в сообществе разработчиков C++. Позволяет легко отличать переменные от типов, предотвращает потенциальные коллизии имён, например: Line line;

5. Именованные константы (включая значения перечислений) должны быть записаны в верхнем регистре с нижним подчёркиванием в качестве разделителя.

```
MAX_ITERATIONS, COLOR_RED, PI
```

Общая практика в сообществе разработчиков C++. Использование таких констант должно быть сведено к минимуму. В большинстве случаев реализация значения в виде метода — лучшее решение:

```
  int getMaxIterations() // НЕЛЬЗЯ: MAX_ITERATIONS = 25
  {
    return 25;
  }
```

Эта форма более читаемая и гарантирует единый интерфейс к значениям, хранящимся в классе.

6. Названия методов и функций должны быть глаголами, быть записанными в смешанном регистре и начинаться с нижнего.

```
getName(), computeTotalWidth()
```

Совпадает с правилом для переменных, но отличие между ними состоит в их специфических формах.

7. Названия пространств имён следует записывать в нижнем регистре.

```
model::analyzer, io::iomanager, common::math::geometry
```

Общая практика в сообществе разработчиков C++.

8. Следует называть имена типов в шаблонах одной заглавной буквой.

```
template<class t style="box-sizing: border-box;"> ...
template<class c, class d style="box-sizing: border-box;"> ...
</class c, class d></class t>
```

Общая практика в сообществе разработчиков C++. Позволяет выделить имена шаблонов среди других используемых имён.

9. Аббревиатуры и сокращения в именах должны записываться в нижнем регистре.

```
exportHtmlSource(); // НЕЛЬЗЯ: exportHTMLSource();
openDvdPlayer();    // НЕЛЬЗЯ: openDVDPlayer();
```

Использование верхнего регистра может привести к конфликту имён, описанному выше. Иначе переменные бы имели имена dVD, hTML и т. д., что не является удобочитаемым. Другая проблема уже описана выше; когда имя связано с другим, читаемость снижается; слово, следующее за аббревиатурой, не выделяется так, как следовало бы.

10. Глобальные переменные всегда следует использовать с оператором разрешения области видимости (::).

```
::mainWindow.open(), ::applicationContext.getName()
```

Следует избегать использования глобальных переменных. Предпочтительнее использование синглтонов.

11. Членам класса с модификатором private следует присваивать суффикс-подчёркивание.

```
class SomeClass {
  private:
    int length_;
}
```

Не считая имени и типа, область видимости — наиболее важное свойство переменной. Явное указание модификатора доступа в виде подчёркивания избавляет от путаницы между членами класса и локальными переменными. Это важно, поскольку переменные класса имеют большее значение, нежели переменные в методах, и к ним следует относиться более осторожно.

Дополнительным эффектом от суффикса-подчёркивания является разрешение проблемы именования в методах, устанавливающих значения, а также в конструкторах:

```
  void setDepth (int depth)
  {
    depth_ = depth;
  }
```

Проблема заключается в том, что существует два варианта подчёркивания — в виде суффикса и в виде префикса. Оба варианта широко используются, но рекомендуется именно первый вариант, потому что он обеспечивает лучшую читаемость. Следует отметить, что определение модификатора доступа у переменных — иногда спорный вопрос. Хотя кажется, что рекомендуемая практика набирает сторонников и становится всё более распространённой в среде профессионалов.

12. Настраиваемым переменным следует давать то же имя, что и у их типа.

```
void setTopic(Topic* topic)      // НЕЛЬЗЯ: void setTopic(Topic* value)
                                 // НЕЛЬЗЯ: void setTopic(Topic* aTopic)
                                 // НЕЛЬЗЯ: void setTopic(Topic* t)

void connect(Database* database) // НЕЛЬЗЯ: void connect(Database* db)
                                 // НЕЛЬЗЯ: void connect (Database* oracle
DB)
```

Сокращайте сложность путём уменьшения числа используемых терминов и имён. Также упрощает распознавание типа просто по имени переменной.

Если по какой-то причине эта рекомендация кажется неподходщей, это означает, что имя типа выбрано неверно.

Не являющиеся настраиваемыми переменные могут быть названы по их назначению и типу:

```
  Point  startingPoint, centerPoint;
  Name   loginName;
```

13. Все имена следует записывать по-английски.

```
fileName;   // НЕ РЕКОМЕНДУЕТСЯ: imyaFayla
```

Английский наиболее предпочителен для интернациональной разработки.

14. Переменные, имеющие большую область видимости, следует называть длинными именами, имеющие небольшую область видимости — короткими.

Имена временных переменных, использующихся для хранения временных значений или индексов, лучше всего делать короткими. Программист, читающий такие переменные, должен иметь возможность предположить, что их значения не используются за пределами нескольких строк кода. Обычно это переменные i, j, k, l, m, n (для целых), а также c и d (для символов).

15. Имена объектов не указываются явно, следует избегать указания названий объектов в именах методов.

```
line.getLength();   // НЕ РЕКОМЕНДУЕТСЯ: line.getLineLength();
```

Второй вариант смотрится вполне естественно в объявлении класса, но совершенно избыточен при использовании, как это и показано в примере.

(Пункт № 16 отсутствует.— Примечание переводчика.)

3.2 Особые правила именования

17. Слова get/set должны быть использованы везде, где осуществляется прямой доступ к атрибуту.

```
employee.getName();
employee.setName(name);

matrix.getElement(2, 4);
matrix.setElement(2, 4, value);
```

Общая практика в сообществе разработчиков C++. В Java это соглашение стало более-менее стандартным.

18. Слово compute может быть использовано в методах, вычисляющих что-либо.

```
valueSet->computeAverage();
matrix->computeInverse()
```

Дайте читающему сразу понять, что это времязатратная операция.

19. Слово find может быть использовано в методах, осуществляющих какой-либо поиск.

```
vertex.findNearestVertex();
matrix.findMinElement();
```

Дайте читающему сразу понять, что это простой метод поиска, не требующий больших вычислений.

20. Слово initialize может быть использовано там, где объект или сущность инициализируется.

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

Следует отдавать предпочтение американскому варианту initialize, нежели британскому initialise. Следует избегать сокращения init.

21. Переменным, представляющим GUI, следует давать суффикс, соответствующий имени типа компонента.

```
mainWindow, propertiesDialog, widthScale, loginText,
leftScrollbar, mainForm, fileMenu, minLabel, exitButton, yesToggle и т. д.
```

Улучшает читаемость, поскольку имя даёт пользователю прямую подсказку о типе переменной и, следовательно, ресурсах объектов.

22. Множественное число следует использовать для представления наборов (коллекций) объектов.

```
vector  points;
int            values[];
```

Улучшает читаемость, поскольку имя даёт пользователю прямую подсказку о типе переменной и операциях, которые могут быть применены к этим элементам.

23. Префикс n следует использовать для представления числа объектов.

```
nPoints, nLines
```

Обозначение взято из математики, где оно является установившимся соглашением для обозначения числа объектов.

24. Суффикс No следует использовать для обозначения номера сущности.

```
tableNo, employeeNo
```

Обозначение взято из математики, где оно является установившимся соглашением для обозначения номера сущности.

Другой неплохой альтернативой является префикс i: iTable, iEmployee. Он ясно даёт понять, что перед нами именованный итератор.

25. Переменным-итераторам следует давать имена i, j, k и т. д.

```
for (int i = 0; i < nTables); i++) {
  :
}

for (vector::iterator i = list.begin(); i != list.end(); i++) {
  Element element = *i;
  ...
}
```

Обозначение взято из математики, где оно является установившимся соглашением для обозначения итераторов.

Переменные с именами j, k и т. д. рекомендуется использовать только во вложенных циклах.

26. Префикс is следует использовать только для булевых (логических) переменных и методов.

```
isSet, isVisible, isFinished, isFound, isOpen
```

Общая практика в сообществе разработчиков C++, иногда используемая и в Java. Использование этого префикса избавляет от таких имён, как status или flag. isStatus или isFlag просто не подходят, и программист вынужден выбирать более осмысленные имена.

В некоторых ситуациях префикс is лучше заменить на другой: has, can или should:

```
  bool hasLicense();
  bool canEvaluate();
  bool shouldSort();
```

27. Симметричные имена должны использоваться для соответствующих операций.

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

Уменьшайте сложность за счёт симметрии.

28. Следует избегать сокращений в именах.

```
computeAverage();   // НЕЛЬЗЯ: compAvg();
```

Рассмотрим два вида слов. Первые — обычные слова, перечисляемые в словарях, которые нельзя сокращать. Никогда не сокращайте:

```
cmd    вместо    command
cp     вместо    copy
```

```
pt     вместо    point
comp   вместо    compute
init   вместо    initialize
```

и т. д.

Второй вид — слова, специфичные для какой-либо области, которые известны по своему сокращению/аббревиатуре. Их следует записывать сокращённо. Никогда не пишите:

```
 HypertextMarkupLanguage  вместо    html
 CentralProcessingUnit    вместо    cpu
 PriceEarningRatio        вместо    pe
```

и т. д.

29. Следует избегать дополнительного именования указателей.

```
Line* line; // НЕ РЕКОМЕНДУЕТСЯ: Line* pLine;
            // НЕ РЕКОМЕНДУЕТСЯ: Line* linePtr;
```

Множество переменных в C/C++ являются указателями. Только в том случае, когда тип объекта в языке C++ особенно важен, имя должно отражать его.

30. Нельзя давать булевым (логическим) переменным имена, содержащие отрицание.

```
bool isError; // НЕЛЬЗЯ: isNoError
bool isFound; // НЕЛЬЗЯ: isNotFound
```

Проблема возникает, когда такое имя используется в конъюнкции с оператором логического отрицания, что влечёт двойное отрицание. Результат не обязательно будет отрицанием !isNotFound.

31. Константы в перечислениях могут иметь префикс — общее имя типа.

```
enum Color {
  COLOR_RED,
  COLOR_GREEN,
  COLOR_BLUE
};
```

Это даёт дополнительную информацию о том, где находится объявление, какие константы описаны в одном перечислении, а также какую концепцию являют собой константы.

Другим подходом является обращение к константам по их общему типу: Color::RED, Airline::AIR_FRANCE и т. д.

Обратите внимание, что имя перечисления обычно записано в единственном числе, например: enum Color {...}. Имя во множественном числе хорошо выглядит при объявлении, но не очень хорошо подходит для практического использования.

32. Классам исключений следует присваивать суффикс Exception.

```
class AccessException
{
  :
}
```

Классы исключений в действительности не являются частью архитектуры программ, и такое именование отделяет их от других классов.

33. Функциям (методам, возвращающим какие-либо значения) следует давать имена в зависимости от того, что они возвращают, а процедурам — в зависимости от того, что они выполняют (методы void).

Улучшайте читаемость. Такое именование даёт понять, что метод делает, а что нет, а также избавляет код от потенциальных побочных эффектов.

4 Файлы

4.1 Файлы исходных кодов

34. Заголовочным файлам C++ следует давать расширение .h (предпочтительно) либо .hpp. Файлы исходных кодов могут иметь расширения .c++ (рекомендуется), .C, .cc либо .cpp.

```
MyClass.c++, MyClass.h
```

Это расширения, одобряемые стандартом C++.

35. Класс следует объявлять в заголовочном файле и определять (реализовывать) в файле исходного кода, имена файлов совпадают с именем класса.

```
MyClass.h, MyClass.c++
```

Облегчает поиск связанных с классом файлов. Очевидное исключение — шаблонные классы, которые должны быть объявлены и определены в заголовочном файле.

36. Все определения должны находиться в файлах исходного кода.

```
class MyClass
{
public:
  int getValue () {return value_;}  // НЕЛЬЗЯ!
  ...

private:
  int value_;
}
```

Заголовочные файлы объявляют интерфейс, файлы исходного кода его реализовывают. Если программисту необходимо найти реализацию, он должен быть уверен, что найдёт её именно в файле исходного кода.

37. Содержимое файлов не должно превышать 80 колонок.

80 колонок — широко распространённое разрешение для редакторов, эмуляторов терминалов, принтеров и отладчиков; файлы передаются между различными людьми, поэтому нужно придерживаться этих ограничений. Уместная разбивка строк улучшает читаемость при совместной работе над исходным кодом.

38. Нельзя использовать специальные символы (например, TAB) и разрывы страниц.

Такие символы вызывают ряд проблем, связанных с редакторами, эмуляторами терминалов и отладчиками, используемыми в программах для совместной разработки и кроссплатформенных средах.

39. Незавершённость разбитых строк должна быть очевидна.

```
totalSum = a + b + c +
           d + e;

function (param1, param2,
          param3);

setText ("Long line split"
         "into two parts.");

for (int tableNo = 0; tableNo < nTables;
     tableNo += tableStep) {
  ...
}
```

Разбивка строк появляется, когда ограничение на 80 колонок, описанное выше, нарушается. Сложно дать жёсткие правила по разбивке, но примеры выше показывают общие принципы.

В общем случае:

- разрыв после запятой;
- разрыв после оператора;
- выравнивание новой строки с началом выражения на предыдущей строке.

4.2 Включения файлов

40. Заголовочные файлы должны содержать защиту от вложенного включения.

```
#ifndef COM_COMPANY_MODULE_CLASSNAME_H
#define COM_COMPANY_MODULE_CLASSNAME_H
  :
#endif // COM_COMPANY_MODULE_CLASSNAME_H
```

Конструкция позволяет избегать ошибок компиляции. Это соглашение позволяет увидеть положение файла в структуре проекта и предотвращает конфликты имён.

41. Директивы включения следует сортировать (по месту в иерархии системы, ниже уровень — выше позиция) и группировать. Оставляйте пустую строку между группами.

```
#include
#include

#include
#include

#include "com/company/ui/PropertiesDialog.h"
#include "com/company/ui/MainWindow.h"
```

Пути включения не должны быть абсолютными. Вместо этого следует использовать директивы компилятора.

42. Директивы включения должны располагаться только в начале файла.

Общая практика. Избегайте нежелательных побочных эффектов, которые может вызвать «скрытое» включение где-то в середине файла исходного кода.

5 Выражения

5.1 Типы

43. Локальные типы, используемые в одном файле, должны быть объявлены только в нём.

Улучшает сокрытие информации.

44. Разделы класса public, protected и private должны быть отсортированы. Все разделы должны быть явно указаны.

Сперва должен идти раздел public, что избавит желающих ознакомиться с классом от чтения разделов protected/private.

45. Приведение типов должно быть явным. Никогда не полагайтесь на неявное приведение типов.

```
floatValue = static_cast(intValue); // НЕЛЬЗЯ: floatValue = intValue;
```

Этим программист показывает, что ему известно о различии типов, что смешение сделано намеренно.

5.2 Переменные

46. Следует инициализировать переменные в месте их объявления.

Это даёт гарантию, что переменные пригодны для использования в любой момент времени. Но иногда нет возможности осуществить это:

```
  int x, y, z;
  getCenter(&x, &y, &z);
```

В этих случаях лучше оставить переменные неинициализированными, чем присваивать им какие-либо значения.

47. Переменные никогда не должны иметь двойной смысл.

Улучшайте читаемость, убеждаясь, что все представленные концепции не предполагают разночтений. Сокращайте возможность ошибки из-за побочных эффектов.

48. Следует избегать использования глобальных переменных.

Не существует причины использовать глобальные переменные в C++ (на самом деле существует.— Примечание переводчика). То же касается глобальных функций и (статических) переменных, область видимости которых — весь файл.

49. Не следует объявлять переменные класса как public.

Эти переменные нарушают принципы сокрытия информации и инкапсуляции. Вместо этого используйте переменные с модификатором private и соответствующие функции доступа. Исключение — класс без поведения, практически структура данных (эквивалент структур языка C). В этом случае нет смысла скрывать эти переменные.

Обратите внимание, что структуры в языке C++ оставлены только для совместимости с C; их использование ухудшает читаемость кода. Вместо структур используйте классы.

(Пункт № 50 отсутствует.— Примечание переводчика.)

51. Символ указателя или ссылки в языке C++ следует ставить сразу после имени типа, а не с именем переменной.

```
float* x; // НЕ РЕКОМЕНДУЕТСЯ: float *x;
int& y;   // НЕ РЕКОМЕНДУЕТСЯ: int &y;
```

То, что переменная — указатель или ссылка, относится скорее к её типу, а не к имени. Программисты на C часто используют другой подход, но в C++ лучше придерживаться этой рекомендации.

(Пункт № 52 отсутствует.— Примечание переводчика.)

53. Следует избегать неявного сравнения булевых (логических) переменных и указателей с нулём.

```
if (nLines != 0)  // НЕ РЕКОМЕНДУЕТСЯ: if (nLines)
if (value != 0.0) // НЕ РЕКОМЕНДУЕТСЯ: if (value)
```

Стандарт C++ не гарантирует, что значения переменных int и float, равные нулю, будут представлены как бинарный 0. Также при явном сравнении видно сравниваемый тип.

Логично было бы предположить, что также и указатели не следует неявно сравнивать с нулём, например, if (line == 0) вместо if (line). Последнее является очень распространённой практикой в C/C++, поэтому также может быть использовано.

54. Переменные следует объявлять в как можно меньшей области видимости.

Это упрощает контроль над действием переменной и сторонними эффектами.

5.3 Циклы

55. Нельзя включать в конструкцию for() выражения, не относящиеся к управлению циклом.

```
sum = 0;                          // НЕЛЬЗЯ: for (i = 0, sum = 0; i < 100; i+
+)
for (i = 0; i < 100; i++)                 sum += value[i];
  sum += value[i];
```

Улучшайте поддержку и читаемость. Строго разделяйте контроль над циклом и то, что в нём содержится.

56. Переменные, относящиеся к циклу, следует инициализировать непосредственно перед ним.

```
isDone = false;           // НЕ РЕКОМЕНДУЕТСЯ: bool isDone = false;
while (!isDone) {         //       :
  :                       //       while (!isDone) {
}                         //         :
                          //       }
```

57. Можно избегать циклов do-while.

Такие циклы хуже читаемы, поскольку условие описано после тела. Читающему придётся просмотреть весь цикл, чтобы понять его работу.

Циклы do-while вообще не являются острой необходимостью. Любой такой цикл может быть заменён на цикл while или for.

Меньшее число используемых конструкций улучшает читаемость.

58. Следует избегать использования break и continue в циклах.

Такие выражения следует использовать только тогда, когда они повышают читаемость.

(Пункт № 59 отсутствует.— Примечание переводчика.)

60. Для бесконечных циклов следует использовать форму while (true) .

```
while (true) {
  :
}

for (;;) {  // НЕТ!
  :
}

while (1) { // НЕТ!
  :
}
```

Проверка на единицу не является необходимой и бессмысленна. Форма for (;;) не очень читаема; также не является очевидным, что цикл бесконечный.

5.4 Условные выражения

61. Строго избегайте сложных уловных выражений. Вместо этого вводите булевы переменные.

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
  :
}

// NOT:
if ((elementNo < 0) || (elementNo > maxElement)||
     elementNo == lastElement) {
  :
}
```

Задание булевых переменных для выражений приведёт к самодокументированию программы. Конструкцию будет легче читать, отлаживать и поддерживать.

62. Ожидаемую часть следует располагать в части if, исключение — в части else.

```
bool isOk = readFile (fileName);
if (isOk) {
  :
}
else {
```

```
  :
}
```

Это позволяет убедиться, что исключения не вносят неясности в нормальный ход выполнения. Важно для читаемости и производительности.

63. Условие следует размещать в отдельной строке.

```
if (isDone)       // НЕ РЕКОМЕНДУЕТСЯ: if (isDone) doCleanup();
  doCleanup();
```

Применяется для отладки.

64. Следует строго избегать исполнимых выражений в условиях.

```
File* fileHandle = open(fileName, "w");
if (!fileHandle) {
  :
}

// НЕЛЬЗЯ:
if (!(fileHandle = open(fileName, "w"))) {
  :
}
```

Исполняемые выражения в условиях усложняют читаемость. Особенно это касается новичков в C/C++.

5.5 Разное

65. Следует избегать «магических» чисел в коде. Числа, отличные от 0 или 1, следует объявлять как именованные константы.

Если число само по себе не имеет очевидного значения, читаемость улучшается путём введения именованной константы. Другой подход — создание метода, с помощью которого можно было бы осуществлять доступ к константе.

66. Константы с плавающей точкой следует записывать с десятичной точкой и с указанием по крайней мере одной цифры после запятой.

```
double total = 0.0;    // НЕ РЕКОМЕНДУЕТСЯ:  double total = 0;
double speed = 3.0e8;  // НЕ РЕКОМЕНДУЕТСЯ:  double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

Это подчёркивает различные подходы при работе с целыми числами и числами с плавающей точкой. С точки зрения математики, эти две модели совершенно различны и не совместимы.

А также (как это показано в последнем примере выше) делается акцент на типе переменной (sum) в том месте, где это не является очевидным.

67. Константы с плавающей точкой следует всегда записывать, по крайней мере, с одной цифрой до десятичной точки.

```
double total = 0.5;  // НЕ РЕКОМЕНДУЕТСЯ:  double total = .5;
```

Система чисел и выражений в C++ заимствована из математики, и следует придерживаться традиционных форм записи, где это возможно. Помимо прочего, 0.5 — более читаемо, чем .5 (первый вариант никак не спутать с числом 5).

68. У функций нужно обязательно указывать тип возвращаемого значения.

```
int getValue()   // НЕЛЬЗЯ: getValue()
{
  :
}
```

Если это не указано явно, C++ считает, что возвращаемое значение имеет тип int. Никогда нельзя полагаться на это, поскольку такой способ может смутить программистов, не знакомых с ним.

69. Не следует использовать goto.

Этот оператор нарушает принципы структурного программирования. Следует использовать только в очень редких случаях (например, для выхода из глубоко вложенного цикла), когда иные варианты однозначно ухудшат читаемость.

70. Следует использовать «0» вместо «NULL».

NULL является частью стандартной библиотеки C и устарело в C++.

6 Оформление и комментарии

6.1 Оформление

71. Основной отступ следует делать в два пробела.

```
for (i = 0; i < nElements; i++)
  a[i] = 0;
```

Отступ в один пробел достаточно мал, чтобы отражать логическую структуру кода. Отступ более 4 пробелов делает глубоко вложенный код нечитаемым и увеличивает вероятность того, что строки придётся разбивать. Широко распространены варианты в 2, 3 или 4 пробела; причём 2 и 4 — более широко.

72. Блоки кода следует оформлять так, как показано в примере 1 (рекомендуется) или в примере 2, но ни в коем случае не так, как показано в примере 3. Оформление функций и классов должно следовать примеру 2.

```
while (!done) {
  doSomething();
  done = moreToDo();
}


while (!done)
{
  doSomething();
  done = moreToDo();
}
```

```
while (!done)
  {
    doSomething();
    done = moreToDo();
  }
```

Пример 3 использует лишние отступы, что мешает ясному отображению логической структуры кода.

73. Объявления классов следует оформлять следующим образом:

```
class SomeClass : public BaseClass
{
  public:
    ...

  protected:
    ...

  private:
    ...
}
```

Частное следствие из правила, указанного выше.

74. Определения методов следует оформлять следующим образом:

```
void someMethod()
{
  ...
}
```

Следствие из правила, указанного выше.

75. Конструкцию if-else следует оформлять следующим образом:

```
if (condition) {
  statements;
}

if (condition) {
  statements;
}
else {
   statements;
}

if (condition) {
  statements;
}
else if (condition) {
  statements;
}
else {
  statements;
}
```

Следствие из правила, указанного выше. Причём написание else на той же строке, где стоит закрывающая фигурная скобка первого блока, не является запрещённым:

```
  if (condition) {
    statements;
  } else {
    statements;
  }
```

Лучше каждую часть if-else помещать на отдельной строке. Это упрощает действия с кодом, например, перемещение блока else.

76. Цикл for следует оформлять следующим образом:

```
for (initialization; condition; update) {
  statements;
}
```

Следствие из правила, указанного выше.

77. Цикл for с пустым телом следует оформлять следующим образом:

```
for (initialization; condition; update)
  ;
```

Делает акцент для читающего на том, что тело пусто. Однако циклов, не имеющих тела, следует избегать.

78. Цикл while следует оформлять следующим образом:

```
while (condition) {
  statements;
}
```

Следствие из правила, указанного выше.

79. Цикл do-while следует оформлять следующим образом:

```
do {
  statements;
} while (condition);
```

Следствие из правила, указанного выше.

80. Конструкцию switch следует оформлять следующим образом:

```
switch (condition) {
  case ABC :
    statements;
    // Отсутствует "break"

  case DEF :
    statements;
    break;

  case XYZ :
```

```
    statements;
    break;

  default :
    statements;
    break;
}
```

Обратите внимание, что каждое слово case имеет отступ относительно всей конструкции, что помогает её выделить. Также обратите внимание на пробелы перед двоеточиями. Если где-то отсутствует ключевое слово break, то предупреждением об этом должен служить комментарий. Программисты часто забывают ставить это слово, поэтому случай нарочного его пропуска должен описываться специально.

81. Конструкцию try-catch следует оформлять следующим образом:

```
try {
  statements;
}
catch (Exception& exception) {
  statements;
}
```

Следствие из правила, указанного выше. Вопросы, касающиеся закрывающих фигурных скобок у конструкции if-else, применимы и здесь.

82. Если конструкция if-else содержит только одно выражение в теле, фигурные скобки можно опускать.

```
if (condition)
  statement;

while (condition)
  statement;

for (initialization; condition; update)
  statement;
```

Рекомендуется всё же не опускать фигурные скобки.

83. Возвращаемый функцией тип может располагаться над именем самой функции.

```
void
MyClass::myMethod(void)
{
  :
}
```

Так функции выровнены в одну колонку.

6.2 Пробелы

84. Пробелы

- Операторы следует отбивать пробелами.
- После зарезервированных ключевых слов языка C++ следует ставить пробел.
- После запятых следует ставить пробелы.

- Двоеточия следует отбивать пробелами.
- После точек с запятой в цикле for следует ставить пробелы.

```
a = (b + c) * d; // НЕ РЕКОМЕНДУЕТСЯ: a=(b+c)*d

while (true)   // НЕ РЕКОМЕНДУЕТСЯ: while(true)
{
  ...

doSomething(a, b, c, d);  // НЕ РЕКОМЕНДУЕТСЯ: doSomething(a,b,c,d);

case 100 :  // НЕ РЕКОМЕНДУЕТСЯ: case 100:

for (i = 0; i < 10; i++) {  // НЕ РЕКОМЕНДУЕТСЯ: for(i=0;i<10;i++){
  ...
```

Выделяет отдельные части выражений. Улучшает читаемость. Сложно дать всеобъемлющий набор рекомендаций относительно пробелов в языке C++. Рекомендации выше должны показать общие принципы.

85. После имён методов может идти пробел, если далее следует другое имя.

```
doSomething (currentFile);
```

Выделяет отдельные имена. Улучшает читаемость. Если далее нет никакого имени, пробел можно опускать (doSomething()).

Другим подходом является указание пробела сразу после открывающей скобки. Использующие его также обычно ставят пробел и перед закрывающей скобкой: doSomething(currentFile);. Это позволяет выделять отдельные имена; пробел перед закрывающей скобкой выглядит неестественно, но без него выражение выглядит несимметрично (doSomething(currentFile);).

86. Логические блоки в коде следует отделять пустой строкой.

```
Matrix4x4 matrix = new Matrix4x4();

double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

matrix.setElement(1, 1,  cosAngle);
matrix.setElement(1, 2,  sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2,  cosAngle);

multiply(matrix);
```

Улучшает читаемость.

87. Методы рекомендуется отделять тремя пустыми строками.

Это позволяет лучше их выделять.

88. Переменные в объявлениях можно выравнивать.

```
AsciiFile* file;
int        nPoints;
float      x, y;
```

Улучшает читаемость. Чётче видны пары тип — переменная.

89. Используйте выравнивание везде, где это улучшает читаемость.

```
if      (a == lowValue)    compueSomething();
else if (a == mediumValue) computeSomethingElse();
else if (a == highValue)   computeSomethingElseYet();

value = (potential        * oilDensity)   / constant1 +
        (depth            * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity)   / constant3;

minPosition     = computeDistance(min,     x, y, z);
averagePosition = computeDistance(average, x, y, z);

switch (value) {
  case PHASE_OIL   : strcpy(phase, "Oil");   break;
  case PHASE_WATER : strcpy(phase, "Water"); break;
  case PHASE_GAS   : strcpy(phase, "Gas");   break;
}
```

Есть множество случаев, когда код можно дополнительно выравнивать, даже если это нарушает установленные ранее правила.

6.3 Комментарии

90. Сложный код, написанный с использованием хитрых ходов, следует не комментировать, а переписывать!

Следует делать как можно меньше комментариев, делая код самодокументируемым путём выбора правильных имён и создания ясной логической структуры.

91. Все комментарии следует писать на английском.

В интернациональной среде английский — предпочтительный язык.

92. Используйте // для всех комментариев, включая многострочные.

```
// Комментарий, расположенный
// на нескольких строках.
```

Если следовать этой рекомендации, многострочные комментарии /* */ можно использовать для отладки и иных целей.

После // следует ставить пробел, а сам комментарий следует начинать писать с большой буквы завершать точкой.

93. Комментарии следует располагать так, чтобы они относились к тому, что они описывают.

```
while (true) {       // НЕ РЕКОМЕНДУЕТСЯ:  while (true) {
  // Do something                           // Do something
  something();                                 something();
}                                           }
```

Это делается с тем, чтобы избежать ситуацию, когда комментарии нарушают логическую структуру программы.

94. Комментарии к классам и заголовкам методов следует делать в соответствии с соглашениями JavaDoc.

Программисты на языке Java используют более развитый подход к документированию благодаря стандартному автоматическому средству Javadoc, которое является частью пакета разработки и позволяет автоматически создавать документацию в формате HTML из комментариев в коде.

Подобные средства есть и в C++. Они следуют тем же соглашениям о синтаксисе тегов, что и JavaDoc (см., например, Doc++ или Doxygen).