Министерство науки и высшего образования Российской Федерации
ФГБОУ ВО «Удмуртский государственный университет»
Институт математики, информационных технологий и физики
Кафедра теоретических основ информатики

А.Е. Анисимов

Требования и рекомендации по оформлению программного кода на языках C и C++

Учебно-методическое пособие

Ижевск
2020

УДК 004.424
ББК 32.973.26-018
А67

Рекомендовано к изданию Учебно-методическим советом УдГУ

Рецензент:
М. А. Клочков, к. ф.-м.н., доцент кафедры вычислительных систем и информационных технологий

Анисимов А.Е.
А67 Требования и рекомендации по оформлению программного кода на языках С и С++ / А.Е. Анисимов.– Ижевск: Издательский центр «Удмуртский университет», 2020. – 48 с.

ISBN 978-5-4312-0787-7

Следование принятым правилам оформления программного кода на языках С и С++ унифицирует внешний вид программ, делает их более читабельными и понятными. Процесс тестирования, отладки и изучения исходного кода программ сокращается по времени. Навыки правильного оформления кода и следования общепринятым единообразным принципам необходимо развивать на начальной стадии формирования компетенций программиста.

Предлагаемое учебно-методическое пособие предназначено для использования в рамках дисциплин начального обучения программированию программ высшего образования по программам бакалавриата «Прикладная информатика», «Информационные системы», «Фундаментальная информатика и информационные технологии» и ряда других.

УДК 004.424
ББК 32.973.26-018

ISBN 978-5-4312-0787-7

Содержание

Введение

Данное учебно-методическое пособие предназначено для использования в рамках дисциплин образовательных программ высшего образования «Информатика и программирование», «Практикум по программированию», «Технологии программирования» и других для ряда направлений подготовки "Прикладная информатика", "Информационные системы", "Фундаментальная информатика и информационные технологии".

Правила, стандарты и соглашения по оформлению программного кода на языке программирования (coding standard, coding convention, programming style, code style guide) необходимо осваивать одновременно с самим языком. Единый стиль написания текстов программ облегчает понимание, упрощает взаимодействие разработчиков программного обеспечения, значительно ускоряет процессы тестирования и отладки.

Для каждого языка программирования в той или иной форме существуют соглашения и правила по написанию кода: в некоторых случаях это формальные документы (как для языка Java), в других – сложившиеся полуофициальные традиции профессиональных сообществ разработчиков или корпоративные регламенты.

При начальном обучении программированию важно изучение правил оформления кода на самых ранних этапах для последующего формирования и закрепления профессиональных навыков по строгому следованию требованиям написания программ.

В данном пособии рассматривается набор требований и рекомендаций по оформлению кода на языках программирования C и C++ как элемент учебной дисциплины для выполнения обучающих упражнений. Эти правила сформированы на основе изучения различных регламентов и сложившихся традиций кодирования на этих языках.

Каждая тема снабжена контрольными материалами для проверки и самопроверки уровня полученных знаний, умений и навыков. Основной формой освоения материала является самостоятельная деятельность студента при наличии необходимого контроля со стороны преподавателя.

Правила, требования и рекомендации содержат сквозную для всего пособия нумерацию. Примеры, содержащие корректно оформленный код, забраны в рамку голубого цвета:

```
int sum = 0; // определение и инициализация переменной
```

Примеры, содержащие некорректно оформленный код или не рекомендуемые для применения конструкции, приводятся в рамке красного цвета:

```
for (i = 0; i < 5; i++) {...} // магическое число 5!
```

Не следует воспринимать последовательность строк в одной рамке всегда как полный фрагмент алгоритма, часто это просто разрозненные однострочные примеры для иллюстрации правил.

Программный код оформляется **черным цветом**, комментарии в коде – **зеленым**. Комментарии могут быть однострочными и обозначаться знаком `//`, либо – многострочными, обрамляемыми знаками `/*` и `*/`. Названия конструкций выделяются голубым цветом, например **операторы**.

1. Значение стиля оформления кода и его формализация

Болтовня ничего не стоит. Покажите мне код.
Линус Торвальдс

История и состояние вопроса

Умение правильно оформлять программный код – одна из профессиональных способностей разработчика программного обеспечения. Важно не только знать и уметь применять конструкции языка программирования, писать эффективные по времени программы, использовать алгоритмические приёмы и стандартные шаблоны программирования, но и «упаковывать» всё это в корректную и эффективную текстовую оболочку. Унификация и рациональная структура кода улучшают читабельность и понятность программ, ускоряют тестирование и отладку, упрощают взаимопонимание и взаимодействие в группе разработчиков. Таким образом, следование единым правилам оформления делается не столько для придания текстам программ красивого внешнего вида, сколько для повышения производительности разработки и качества программного продукта.

Для разных языков программирования существует разная история формирования и принятия правил оформления исходного кода. В большинстве случаев единых общемировых требований и рекомендаций не существует, но складываются определенные традиции и неформальные соглашения в сообществах разработчиков, авторами книг и статей используются общепринятые правила кодирования.

Например, для языка Java в 1995 году было принято и опубликовано соглашение по оформлению кода [1], следование которому является де-факто хорошим тоном для всего сообщества java-программистов. Для некоторых языков программирования источником стандартов оформления является компания-разработчик языка, например Microsoft выпущены рекомендации для языка C# [2]. Компании - производители программного обеспечения также вводят свои внутрикорпоративные требования по оформлению кода. В некоторых случаях такие требования формулируются для отдельных проектов.

Необходимо отличать принимаемые международные стандарты языков программирования (ANSI/ISO, ISO/IEC) от совокупности правил, соглашений,

стандартов оформления программного кода. В первом случае стандарт является основой для реализации языка как инструментального средства разработки, то есть создания транслятора и его инфраструктуры, во втором – это по сути соглашения и правила рекомендательного характера для разработчиков, следование которым не является обязательным с точки зрения синтаксиса или семантики языка, но значительно повышает качество и удобство разработки программных продуктов.

Язык программирования C (иначе - Си) был разработан на рубеже 1960-х и 1970-х годов сотрудником компании Bell Labs Деннисом Ритчи. Язык C++ создавался в начале 1980-х на основе C с использованием объектно-ориентированной парадигмы, его автором является Бьёрн Страуструп, разработчик той же компании.

Не существует принятых общемировых стандартов или рекомендаций по оформлению кода на C и C++. Есть корпоративные, проектные и университетские соглашения по стилю кодирования на этих языках, в частности можно назвать следующие документы:

1. русско-венгерская нотация (проект PBH); стиль использовался при разработке программного обеспечения на нескольких языках; документ рассматривает аспекты его применения при программировании на C и C++ [3];
2. руководство от факультета компьютерных наук Стенфордского университета по стилю оформления кода на C++ в рамках учебного курса CS 106B [4];
3. руководство от компании Google по стилю программирования на C++ [5].

Приводимые в данном пособии и используемые в рамках университетского курса по программированию требования и рекомендации по оформлению программного кода на C/C++ основаны как на сложившейся в компьютерной литературе и прикладном программировании практике, так и на указанных выше документах и соглашениях. Требования адаптированы для учебного процесса.

Так как синтаксис языка C++ основан на синтаксисе языка C, то в части требований к оформлению программ эти два языка практически не отличаются. Поэтому чаще всего под термином «язык программирования» будут

пониматься как C, так и C++. В отдельных случаях и если только будет в этом необходимость о различиях будет указано особо.

Термины и определения

Дадим определения некоторых терминов и понятий.

Выражение (expression) – комбинация операций, вызовов функций и объектов данных, вычисляемая в соответствии с правилами языка.

Идентификатор (identifier) – уникальное имя программного объекта (переменной, функции, класса, объекта и др.), позволяющее его идентифицировать в некоторой части пространства программы.

Интерпретатор (interpreter) – компьютерная программа, осуществляющая интерпретацию исходного кода программы, то есть его построчную обработку и исполнение.

Исходный код (source code) компьютерной программы– текст на языке программирования, который создается и может быть прочитан человеком. Является входом для компилятора или интерпретатора.

Класс (class) – определенный в программе именованный абстрактный тип данных, определяющий интерфейс и реализацию для своих экземпляров.

Комментарий (comment) – пояснение к исходному коду программы, которое игнорируется компилятором; комментарии включаются непосредственно в исходный код.

Компилятор (compiler) – программа, исполняющая трансляцию исходного кода программы в машинный код и последующую сборку кода исполняемой программы.

Компьютерная программа (computer program) – последовательность инструкций или деклараций на языке программирования для выполнения определённых вычислений, решения задач с помощью компьютера.

Массив (array) – встроенный тип данных языка, значения которого состоят из набора однотипных элементов, доступ к которым осуществляется по индексу.

Метод класса (class method) – инкапсулированный элемент класса, функция в классе.

Объект класса (object) – экземпляр класса, переменная типа класса. При исполнении программы каждый её объект находится в определенном состоянии (атрибуты) и характеризуется поведением (методы).

Оператор (statement) – наименьшая самостоятельная единица императивного языка программирования, команда на совершение определенного действия.

Операция (operator) – запись определенного действия над аргументами операции (операндами), выполнение которого, в отличие от оператора, оставляет в месте нахождения операции непустой вычисленный результат.

Переменная (variable) – объект данных программы, связанный с областью памяти, в которой хранится его значение. Переменная обеспечивает доступ к памяти для чтения или изменения значения.

Поле класса (class field, data member) – инкапсулированный элемент данных класса, переменная в классе, атрибут класса.

Препроцессор (preprocessor) – программа, осуществляющая предварительную перед компиляцией обработку исходного кода. Обработка препроцессором – особенность компиляции программ на C/C++.

Спецификация (specification) – законченное описание поведения разрабатываемой программы (подпрограммы); как правило, выполняется на естественном языке и имеет определенную структуру.

Строковый тип (string) – стандартный тип данных языка, значениями которого являются последовательности символов алфавита (строки);

Структура (struct) – встроенный тип данных языка, значения которого состоят из набора разнотипных именованных элементов (полей).

Тип данных (data type) – категория данных, представляющая множество значений данных и операций с ними.

Функция (function) – автономная, как правило – именованная единица программного кода (подпрограмма), которую можно вызвать в программе для исполнения.

Язык программирования (programming language) – формальный искусственный язык, предназначенный для записи компьютерных программ.

Все правила и рекомендации по оформлению кода будут представлены в следующем формате: краткое описание правила с номером (полужирным

шрифтом), более подробное описание правила, примеры (в рамке голубого цвета корректные, в рамке красного цвета – некорректные или не рекомендуемые).

Приведем первое правило.

Правило 1: Правила могут нарушаться, если для этого есть основания

Невозможно в правилах учесть все ситуации, возникающие при составлении кода, как невозможно дать рекомендации на все случаи жизни. Программист по мере накопления опыта формирует собственный стиль, ориентируясь, конечно, на существующие соглашения и стандарты.

```
i++; // увеличение переменной i на 1
// этот ↑ комментарий избыточен
```

Контрольные вопросы и задания темы 1

1. В чем заключается необходимость единого подхода в оформлении программного кода?
2. Каковы преимущества следования требованиям и соглашениям при оформлении программного кода?
3. Каковы недостатки следования требованиям и соглашениям при оформлении программного кода?
4. В чем сходство и различия между компилятором и интерпретатором?
5. В чем сходство и различия между оператором и операцией?
6. В чем сходство и различия между массивом и структурой?
7. Что такое класс, объект, поле и метод класса?
8. Является ли класс типом данных?
9. Для кого и с какой целью пишутся комментарии?

Упражнения по теме 1

1. Сформулируйте (кратко) на естественном языке спецификации следующих программ или функций:
    а). функция «модуль числа»;
    б). функция «максимум из двух»;
    в). функция «очистка экрана»;

г). функция «средняя зарплата»;
д). функция «проверка пароля»;
е). программа «калькулятор»;
ж). программа «мессенджер».

2. Нарисуйте схему, показывающую отношения «общее-частное» или «часть-целое» между следующими группами понятий:
   а). класс, тип данных;
   б). класс, объект класса, поле класса, метод класса;
   в). тип данных, массив, строковый тип, структура;
   г). выражения, операции, операнды.

Задания для самостоятельной работы по теме 1

1. Найдите в Интернет соглашения и рекомендации (международные, корпоративные, проектные и др.), помимо указанных в тексте, по оформлению программного кода для различных современных языков программирования.

2. Соглашения об именовании

Существует только две трудные вещи в информатике:
инвалидация кэша и именование сущностей.
Фил Карлтон

В программах используется именование различных программных объектов (сущностей) – переменных, констант, функций, классов, полей, методов и т.д. Кроме этого, именуются файлы с исходными кодами программ.

Правильно подобранные имена делают код более читабельным и понятным. Также имена могут дать информацию о том, чем является сущность: константой, переменной, функцией, классом и т.д.

Правило 2: Имена должны быть осмысленными

Имя должно не только идентифицировать объекты данных или функции в программе, но и содержать в себе краткое описание смысла и назначения сущности. Имена составляют из одного, двух, реже – трех слов.

Как правило, имена объектов данных (переменных, констант, полей, классов) являются существительными, имена блоков кода (функций, методов) – глаголами.

```
int countPositive;                    //переменная
double averageSalaryEmploees;         //переменная
Point xCenter, yCenter;               //объекты
int getDistance(Point x, Point y);    //метод
```

Неверным будет использование следующих имён:

```
int abcdef; // имя ни о чём не говорит
Line l12345;
short int i1, i2, i3, p2, j2, j3, j23; // можно
          //запутаться в переменных
```

Правило 3: Переменные, имеющие большую область видимости, рекомендуется называть длинными содержательными именами; имеющие небольшую область видимости – короткими.

Однобуквенных имен переменных следует избегать; допускаются однобуквенные имена, если область видимости таких переменных невелика (несколько строк кода). К переменным с короткой областью видимости относятся, например, индексы массива, указатели для перебора.

```
int countWords;// содержательное имя

for (int i = 0; i < n; i++) // индекс
  cout << a[i];

char * p = fileName; // указатель перебора
while (p != '/0') {
  cout << *p++;
}
```

Правило 4: Не рекомендуется сокращать слова в именах

Это не требование, а скорее рекомендация; однако следовать этому правилу желательно, тем более современные средства редактирования кода позволяют не затрачивать много времени на написание ранее определенных имён.

```
float arrayAverage;           // неверно arrAvg
string parseInputString();    // неверно prsInpStr();
bool getMessageBox();         // неверно getMsgBox();
```

Правило 5: Имена должны быть из английских слов

Английский язык де-факто стал языком международного общения в IT-сообществе. Кроме этого, синтаксис самых распространённых ЯП основан на английском.

```
string fileName;
int getAccountNumber;
```

Неверным будет использование следующих имён:

```
string imyaFaila;
int nomerScheta;
long abrufenListenGröße();
```

Правило 6: Составные имена должны набираться в стиле CamelCase

CamelCase («верблюжий» стиль) – это стиль написания фразы из нескольких слов без пробелов, когда каждое слово начинается с заглавной буквы.

lowerCamelCase – это стиль CamelCase, когда первое слово фразы в отличие от остальных начинается со строчной буквы. Так записываются имена *переменных, объектов, полей, методов, функций*.

```
string inputFileName;
Person personGroupLeader;
person.firstName = "Сергей";
cout << person.getFirstName();
```

Не рекомендуется именовать переменные так, как это показано в следующих примерах:

```
double A, B, C;
string FileName;
long int MAX_NUMBER;
```

UpperCaseName – это стиль CamelCase, когда все слова фразы, в том числе первое слово, начинаются с заглавной буквы. Так записываются имена *классов, структур и других типов*.

```
class PointOfPlane {
  double x, y;
};
PointOfPlane a, b, c;
```

Неверным будет именовать классы и типы, начиная со строчной буквы:

```
class lineplane{
  Point beginPoint, endPoint;
};
struct listArray{
/*...*/
};
```

Правило 7: Именованные константы записываются в верхнем регистре с нижним подчёркиванием в качестве разделителя

```
const int MAX_ITERATIONS = 10000;
const double HALF_PI = 3.1415926838 / 2.0;
```

Правило 8: Особые случаи именования

В некоторых случаях именования принято использовать устоявшиеся шаблоны имён и названий. Например, часто используются следующие префиксы для имён:

- get, set – методы доступа к полям класса;
- is – метод, возвращающего булево значение или логическая переменная; другие варианты: has, can, should;
- initialize – метод, инициализирующий сущность;
- run, compute – методы, запускающие процесс.

```
int getSize();    // метод, возвращающий значение
                  //поля size;
void setLength(int length); // метод задающий длину
bool isValidPassword();     // проверка правильности
void runMainForm();    // запуск некоторого процесса
bool canCompute;       // флаг возможности вычисления
```

Также часто применяют как часть имени общепринятые названия операций и алгоритмов, часто – парные [4]:

- 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 другие.

Правило 9: Зарезервированные слова или стандартные имена набираются в нижнем регистре

Так принято. Более того, компилятор не сможет правильно разобрать зарезервированные слова или стандартные имена, если они будут набраны как-то иначе.

```
using namespace std;
while (a != b)
  if (a > b)
    a -= b;
  else
    b -= a;
```

Неверным будет такой код:

```
While (!isEmpty(stack))    // правильно - while
  Std::Cout << stack.pop();// правильно - std::cout
IF (d > 0)                 // правильно if
  x1 = (- b - Sqrt(d)) / (2 * a); // правильно sqrt
```

Контрольные вопросы и задания темы 2

1. Какие цели преследует использование правил именования?
2. Какие преимущества даёт единообразие именования сущностей в программах?
3. Какие недостатки могут быть у строгого следования правилам именования?

4. Как зависит длина имени сущности от размера части программы, где она используется?
5. В чем отличие имени класса от имени члена этого класса?
6. Почему бы не использовать русский язык для именования программных сущностей? Или немецкий?
7. Каковы причины (кроме эстетических) использования правил именования сущностей программы?

Упражнения по теме 2

1. Правильно ли подобраны следующие имена переменных и констант? Если неверно – предложите свой вариант.

```
int a, a1, ABC, StudentCounter;
double setNewPerson;
string name_My_File;
const long double MyPi = 3.1415926;
for (int indexArrayOfRandom = 0; indexArrayOfRandom <
 a.size();   indexArrayOfRandom ++)
   cout << a[indexArrayOfRandom];
bool failChiselNayden;
```

2. Правильно ли подобраны следующие имена функций и методов? Если неверно – предложите свой вариант.

```
bool isValidValue(int value);
int findIndexArray(int * array);
double compAvg(int * array, int n);
double xyz(int a1, doble b12, string sss);
bool Array::isSorted();
Person Person::FindPerson(string name);
```

3. Правильно ли подобраны следующие имена классов? Если неверно – предложите свой вариант.

```
class Person {...};
class studentUdSU {...};
class Exception_Fail_Ne_Naiden {};
class PageHTTP {};
```

Задания для самостоятельной работы по теме 2

1. Проанализируйте код своей ранее написанной программы и приведите все имена в соответствие с требованиями именования.
2. Выясните самостоятельно, что такое «рефакторинг»? Какие инструменты сред разработки C/C++ для рефакторинга существуют? Как с помощью этих инструментов можно управлять именами в программе?

3. Соглашения по форматированию программных конструкций

Измерять продуктивность программирования подсчетом строк кода — это так же, как оценивать постройку самолета по его весу.
Билл Гейтс

Следующий набор правил определяет, как набирать и форматировать программные конструкции и их взаимное расположение. Цель правил – максимальная удобочитаемость текста и, как следствие, быстрое восприятие читающим его человеком смысла и логики программы.

Правило 10: Вложенные конструкции оформляются с помощью отступов (принцип «лесенки»)

Программа представляет собой не линейный текст, а структуру, состоящую из различных языковых конструкций, в том числе – вложенных друг в друга. Для быстрого понимания этой структуры используется принцип «лесенки», когда вложенная конструкция набирается с отступом относительно конструкции, её содержащей.

Лучше всего использовать отступ в два, три или четыре пробела, но размер отступа должен быть единообразным во всей программе. Использование для этих целей табуляции возможно, но может привести к искажениям при изменении настроек редактора среды программирования.

```
if (d > 0)
{
  x1 = (-b - sqrt(d)) / (2 * a);
  x2 = (-b + sqrt(d)) / (2 * a);
}
else
  if (d == 0)
    x1 = -b / (2 * a);
  else
    cout << "no roots" << endl;
```

Так, как указано в примере ниже, оформлять конструкции недопустимо:

```
// поиск максимума в списке
while (p != NULL)
if (p -> data >= max)
max = p -> data;
p = p -> next;
// другой пример
if (a > b)
if (b > c)
cout << "b в середине";
else cout << "c в середине";
else cout << "a в середине";
```

Правило 11: Каждая конструкция должна находиться (начинаться) в отдельной строке

Нельзя располагать разные конструкции в одной и той же строке.

К таким конструкциям относятся:

- описания переменных, констант;
- все заголовки (функций, методов, классов);
- операторы;
- операции присваивания;
- вызовы функций и методов;
- директивы препроцессора.

```
int counterPositiveNumbers;
void runEvaluate();
while (a[i] != findNumber)
{
  i++;
  j++;
}
a += d;
b *= q;
```

Неправильно располагать разные конструкции в одной и той же строке:

```
if (a[i] > max) max = a[i]; // условный и
                //присваивание в одной строке
i++; j++; k--; // изменения переменных в одной
                // строке
while (a != b) if (a > b) a -= b; else b -= a;
```

Правило 12: Условный оператор оформляется следующим образом:

```
// полный вариант
if (условие) {
  операторы;
}
else{
  операторы;
}

// сокращенный вариант
if (условие) {
  операторы;
}

// допустимый вариант
if (условие)
{
  операторы;
}
else
{
  операторы;
}
```

Последний (допустимый) вариант написания фигурных скобок может аналогичным образом применяться и в других операторах языка. Если во вложенном блоке кода размещается только один оператор, то фигурные скобки допускается не писать. Пример правильного оформления кода:

```
if (d > 0) {
  x1 = (- b - sqrt(d)) / (2 * a);
  x2 = (- b + sqrt(d)) / (2 * a);
}
else
  if (d == 0)
    x1 = - b / (2 * a);
  else
    noRoots = true;
```

Неверное оформление if-else:

```
if (d == 0) x = ( - b / (2 * a));
else x = 0;
if (a > b) max = a; else max = b;
```

Правило 13: Цикл for оформляется следующим образом:

```
for (инициализация; условие; модификация) {
  операторы;
}
```

Каждая из частей заголовка может отсутствовать. Пример правильного оформления цикла for:

```
for (int i = 0; i < n; i++) {
  s += a[i];
  m = max(m, a[i]);
}
```

Правило 14: Цикл while оформляется следующим образом:

```
while (условие) {
  операторы;
}
```

Правило 15: Цикл do-while оформляется следующим образом:

```
do {
  операторы;
} while (условие);
```

Правило 16: Оператор выбора switch оформляется следующим образом:

```
switch (условие) {
  case значение1 :
    операторы;
    break;

  case значение2 :
  case значение3 :
    операторы;
    break;

  default :
    операторы;
}
```

Оператор break в блоке case может отсутствовать – об этом лучше предупредить комментарием.

Правило 17: Оператор try-catch оформляется следующим образом:

```
try {
  операторы;
}
catch (ТипИсключения исключение) {
  операторы;
}
```

Правило 18: Определение функций и методов оформляется следующим образом:

```
ТипЗначения nameFunction(параметры)
{
  операторы;
}
```

Конструкторы и деструкторы оформляются аналогично за исключением того, что в заголовке не указывается тип возвращаемого значения. Если в определении имеется список инициализации, то он пишется в строке заголовка после двоеточия.

Примеры правильного оформления определения функции и конструктора:

```
int max(int a, int b)
{
  return a > b ? a : b ;
}

Complex::Complex(double re, double im) : Real(re)
{
  this -> im = im;
}
```

Правило 19: Использование пробелов в выражениях и операторах

Следует отбивать пробелами с двух сторон бинарные операции, двоеточие. Пробел ставится после запятой, точки с запятой, зарезервированных слов.

```
x = (- b + sqrt(d)) / (2 * a); //неверно x=(-b+
                               //sqrt(d))/(2*a);
if (d >= 0)                    //неверно if(d>=0)
  x = - b / (2 * a);           //неверно x=-b/(2*a);
```

```
while (true)                    //неверно while(true)
...;
for (i = 0; i < n; i++)         // for(i=0;i<n;i++)
...;
m = max(a, b, a + b);           // m = max(a,b,a+b);
```

Правило 20: Фрагменты кода, решающие отдельные подзадачи, рекомендуется разделять пустой строкой.

В этом случае программа будет выглядеть не сплошным текстом, а как разделённый на логически завершённые фрагменты программный код. Так он лучше читается.

Также пустой строкой, а иногда двумя или даже тремя, отделяются объявления или определения классов, методов, функций.

Пример использования пустых строк:

```
Matrix matrix = new Matrix(2, 2);

matrix.setElement(0, 0, 10.0);
matrix.setElement(0, 1, 20.0);
matrix.setElement(1, 0, -10.0);
matrix.setElement(1, 1, 15.0);

cout << "Matrix" << endl;
cout << matrix;

double det = matrix.computeDeterminant();
```

Контрольные вопросы и задания темы 3

1. Как влияет на эффективность программы форматирование кода «лесенкой»?
2. С какой целью форматируют исходный код программы «лесенкой»?
3. Какие недостатки у размещения двух или более операторов в одной строке?

4. Надо ли экономить на пробелах и когда?
5. Надо ли экономить на пустых строках и когда?
6. Что такое «список инициализации» конструктора?
7. Можно ли определить одну функцию как вложенную в другую?
8. Как вы думаете, почему отсутствие оператора break в какой-либо ветви оператора switch рекомендуется предупреждать специальным комментарием?

Упражнения по теме 3

1. Укажите, какие ошибки форматирования имеются в следующем коде? Как их исправить?

```
if (a <=b) {
  while (a != b)
    if (a > b) a -= b;
    else b -= a;
} else

swap(a,b);
```

2. Укажите, какие ошибки форматирования имеются в следующем коде? Как их исправить?

```
list* p = head;
while (p != NULL){
  if (p->data >0)
    s += p -> data;
  else
    count++; p=p->next;
}
```

3. Укажите, какие ошибки форматирования имеются в следующем коде? Как их исправить?

```
bool setFriend(Person& person1, Person& person2){
  if(isEnemy (person1,person2)) return false;
  setFriendship(person1, person2);
return true;
}
```

Задания для самостоятельной работы по теме 3

1. Отформатируйте код следующей программы в соответствии с требованиями. Объясните, какой алгоритм представлен этой программой?

```
double a,b;cin >>
 a >> b   ;if (a!= 0) {double x
= -b / a;cout << x << endl;} else if(b!=0)cout <<
"No roots" << endl;else cout <<
"Endless roots" << endl;
```

4. Документирование программ

Пишите вашу программу так, как будто парень, который будет ее поддерживать, жестокий психопат, который знает, где вы живете.
Мартин Голдинг

Исходный текст должен читаться так, чтобы у читающего его человека, как правило – специалиста, по возможности не должно остаться неразрешённых вопросов. Исходный код должен быть самодостаточным с точки зрения понимания его логики человеком. Кроме этого, смысл исходного кода должен быть понят человеком без избыточных затрат времени на его восприятие: в программировании тезис «время – деньги» приобретает вполне реальное воплощение.

Комментарии используются для описания отдельных строк, блоков кода или целого алгоритма. Комментарии должны содержать лишь ту информацию, которая необходима для чтения и понимания программы. Обсуждение нетривиальных вещей или неочевидных решений необходимо, но не нужно описывать то, что и так ясно из кода. Такие избыточные комментарии перегружают текст.

Документирующие комментарии – это особый вид комментариев, которые используются для описания *спецификации кода*, то есть требований к внешнему поведению, которые не зависят от реализации. Обычно документируют классы, функции, методы, интерфейсы и подобные элементы программы. Такие комментарии делаются для разработчиков, которые будут использовать ваши программы, не имея исходного кода.

Написание комментариев – такое же искусство, как и сам процесс программирования. Профессиональный писатель комментариев должен обладать чувством меры, находя баланс между полнотой и лаконичностью текста, уметь точно формулировать смысл, грамотно пользоваться естественным языком для описания формально-языковых конструкций. Это мастерство постигают не сразу, оно приходит с некоторым опытом. Но определённые правила, требования и рекомендации сформулировать можно.

Правило 21: Оформление комментариев

Для поясняющего комментария используют две наклонные черты //, после которых обязательно должен следовать пробел.

Обозначения /* и */ обычно применяются для временного отключения фрагментов кода при его отладке.

Комментарии блоков кода, спецификации классов, методов и функций размещают перед самим кодом, то есть в строках выше него.

```
// Функция производит поиск в массиве array элемента
// со значением desiredValue и возвращает его индекс.
// В случае отсутствия значения в массиве функция
// возвращает -1.
int findValue(Value* array, Value desiredValue)
{...
}
```

Для отдельного оператора допускается размещение комментария справа от него, если только такой комментарий не слишком велик. Желательно последовательность таких комментариев выравнивать относительно друг друга.

```
int countPositiv = 0; // счетчик положительных чисел
list<Book> books;     // список книг библиотеки
moveChess(knight, x, y); // ход конём в клетку x, y
```

Правило 22: Заглавный комментарий файла должен исчерпывающе описывать его назначение.

В заглавном комментарии (иначе – «шапке») файла должно быть описано, какую часть задачи решает код этого файла. В таком описании можно указать:

- имя файла;
- назначение содержимого файла: объявление или реализация класса, реализация функций, в том числе main(). и т.п.;
- постановка исходной задачи;
- фамилия автора;

- краткая техническая информация о состоянии кода, версии, программных интерфейсах и функциональности приложения;
- и другое.

Например, для учебного приложения, состоящего из одного исходного файла, «шапка» может выглядеть так:

```
// labwork03.cpp
// Лабораторная работа № 3 по предмету
// Практикум по программированию, 1 курс, 2 семестр
// ---- Задача: Дан случайный целочисленный массив из
// N элементов. Найти в массиве самую длинную цепочку
// подряд идущих положительных элементов -----
// Автор: студент группы ОАБ020302-11 Иванов С.А.
// Дата готовности : 15.10.2020 года
```

В многофайловых проектах не всегда каждый файл должен содержать заглавный комментарий: часто само название файла и структура проекта исчерпывающе определяют его назначение.

Правило 23: Каждый класс необходимо документировать.

При объявлении класса (или разновидностей класса: абстрактного класса, интерфейса, шаблона) необходима его спецификация – документирующий комментарий. В спецификации описывают назначение, цель класса, его поведение, другие технические моменты.

```
// Класса Point реализует понятие точки
// вещественного пространства произвольной размерности
// Автор : Иванов С.А.
// Последнее изменение : 15.10.2020
```

Правило 24: Каждую функцию, метод необходимо документировать.

В тексте спецификации функции, метода необходимо (но не избыточно) описать:

- поведение и/или цель функции, метода;
- семантику входных параметров;

- семантику выходных параметров;
- возвращаемое значение;
- исключения, если функция их порождает.

Спецификация функции, то есть точное описание её поведения, помогает разработчику сосредоточиться на соответствии её реализации внешним требованиям. Поэтому спецификацию необходимо оформлять *до начала* написания её кода.

```
// Функция sum возвращает сумму своих вещественных
// параметров. Количество параметров - три, два или
// один
// Вход:
//   a, b, c - вещественные
// Возвращаемое значение:
//   сумма a + b + c

double sum(double a, double b = 0.0, double c = 0.0)
{
  return a + b + c;
}
```

Если код читает другой человек, то ему спецификация должна давать полное понимание внешнего поведения метода или функции, даже если он не изучил их реализацию.

```
// Функция solveSquareEquation решает квадратное
// уравнение вида ax^2 + bx + c = 0 в вещественных
// числах.
// Вход:
//   a, b, c - коэффициенты квадратного уравнения
// Выход:
//   roots - множество корней квадратного уравнения
// Возвращаемое значение равно:
//    0, если уравнение имеет конечное число корней,
//    -1, если уравнение имеет бесконечное число
//        корней
int solveSquareEquation(double a, double b, double c,
                        set<double>& roots);
```

Правило 25: Советы по документированию программы.

- комментарии располагают так, чтобы было понятно, какой именно блок кода они описывают;
- программа должна иметь минимально необходимое количество комментариев, а в идеале не иметь их совсем; самодокументируемость программы достигается за счёт удачного подбора имён и ясной структуры кода;
- сфера разработки программного обеспечения преимущественно англоязычна, поэтому будет верным привыкать использовать для комментирования английский язык;
- существуют средства, автоматизирующие документирование исходного кода; наиболее известные из них – утилита Javadoc для языка Java и Doc++ для языка C/C++. Рекомендуется их самостоятельно изучить.

Контрольные вопросы и задания темы 4

1. Для кого нужны комментарии?
2. Объясните, как улучшает программу её комментирование?
3. В чём недостаток избыточного комментирования?
4. Что такое спецификация? Для чего она используется?
5. В чем отличие обычного комментария от комментария документирующего?
6. Почему документирующий комментарий необходимо размещать выше комментируемой сущности (класс, функция, метод), а не справа или ниже?
7. Почему документирующий комментарий рекомендуется писать раньше, чем комментируемая сущность, а не после того, как она уже определена?
8. Какие элементы функции (метода) должны быть описаны в её спецификации?
9. Что необходимо указывать в спецификации файла?

Упражнения по теме 4

1. Критически оцените приведенные комментарии. Приведите свой вариант комментирования.

```
// нахождение факториала числа n
int p = 1;   // переменная для накопления
             // факториала, начинается с 1
for(int i = 1; i <=n; i++) // перебор от 1 до n
  p *= i; // переменную накопления домножаем на i
```

2. Критически оцените приведенные комментарии. Приведите свой вариант комментирования.

```
// Печать
void printMatrix(int** a, int n, int m,
                 int color);
```

3. Критически оцените приведенные комментарии. Приведите свой вариант комментирования.

```
// Лаба 2.
// Дана строка. Найти самую длинную подстроку.
#include <iostream>
...
```

Задания для самостоятельной работы по теме 4

1. Сравните возможности утилит автодокументирования Javadoc Doc++.
2. Изучите вопрос о существовании других средств автоматизации документирования ПО, кроме указанных в предыдущем вопросе.

5. Рекомендации по использованию программных конструкций

Отладка кода вдвое сложнее, чем его написание. Так что если вы пишете код настолько умно, насколько можете, то вы по определению недостаточно сообразительны, чтобы его отлаживать.
Брайан Керниган

Правила этого раздела будет разумнее отнести не к требованиям по оформлению программного кода, а к рекомендациям по созданию более эффективных и ошибкоустойчивых программ. Однако в практике программирования эти вещи настолько связаны, что зачастую отделить одно от другого не только сложно, но и не нужно.

Правило 26: Переменные следует объявлять в как можно меньшей области видимости (принцип локализации)

Переменной предоставляется только та область программы, где она нужна и используется. Переменные, не используемые в части программы, но видимые в ней, образуют избыточный балласт, который может привести к ошибкам или неожиданным эффектам.

В следующем примере области видимости переменных x1, x2 и x ограничены блоками, в которых они объявлены: за пределами этих блоков эти переменные не нужны.

```
if (d > 0.0)
{
  double x1 = (-b - sqrt(d)) / (2 * a);
  double x2 = (-b + sqrt(d)) / (2 * a);
  cout << x1 << " " << x2 << endl;
}
else
  if (d == 0.0) {
    double x = -b / (2 * a);
    cout << x << endl;
  }
  else
```

```
    cout << "no roots" << endl;
```

Правило 27: Следует избегать глобальных переменных.

Это следствие предыдущего правила. Переменные должны быть локализованы в своих функциях, классах или даже блоках. Неявное изменение функцией не принадлежащих её данных является побочным эффектом. Побочный эффект нежелателен, так как поведение и результат работы функции сильно зависит от окружающей среды, а логика программы становится запутанной.

```
int x;  // глобальная переменная

void inc()
{
  x++;  // побочный эффект
}

void main()
{
  x = 1;               // значение 1
  inc();
  cout << x << endl;   // значение 2
}
```

Для обмена данными между функциями нужно использовать параметры, а не глобальные переменные.

```
void inc(int& x)  // явная передача переменной
{
  x++;
}

void main()
{
  int x = 1;    // значение 1
  inc(x);       // явное изменение переменной x
  cout << x << endl;   // значение 2
```

```
}
```

Глобальными допускается делать именованные константы, но даже в этом случае лучше поместить их в специальный класс. Глобальными в программе могут быть классы и некоторые функции, в том числе – main().

Правило 28: Следует инициализировать переменные перед их использованием и не полагаться на значения по умолчанию.

В языках C и C++ переменные *могут получать* значения по умолчанию, но полагаться на них в алгоритме является плохим стилем. Тем более, от версии к версии языка его механизм встроенной инициализации меняется.

Использование неинициализированных переменных может привести к ошибкам!

```
int n;
cin >> n;
int sum;   // переменная не инициализирована!
for (int i = 1; i <= n; i++)
  sum += i;
cout << sum << endl;
```

Среда программирования может предупредить об отсутствии инициализации, а может и не сделать этого, что может повлечь ошибки при исполнении программы. Правильным будет инициализировать переменные перед использованием.

```
int n;
cin >> n;
int sum = 0;    // переменная инициализирована
for (int i = 1; i < n; i++)
  sum += i;
cout << sum << endl;
```

Поля объектов могут инициализироваться разными способами.

```
class A {
private:
  int n = 0; // инициализация при определении
```

```
public:
  A() : n(1) // инициализация в списке инициализации
             // конструктора
  {
    n = 2;   // инициализация конструктором
  }
};
```

Правило 29: Если один и тот же код повторяется два или более раза, то его следует выделить в функцию (класс).

Подпрограммы (функции, процедуры, методы) были созданы в своё время (в том числе) для того, чтобы избежать дублирования идентичного кода.

В приведённом ниже примере такое дублирование наблюдается.

```
void main()
{
  const int n = 10;

  int a[n];
  for (int i = 0; i < n; i++)
    a[i] = rand() % 100;
  for (int i = 0; i < n; i++)
    cout << setw(4) << a[i];
  cout << endl;

  int b[n];
  for (int i = 0; i < n; i++)
    b[i] = rand() % 100;
  for (int i = 0; i < n; i++)
    cout << setw(4) << b[i];
  cout << endl;
}
```

Выделим код, решающий отдельные части задачи, в отдельные функции. При этом каждая функция – это не просто формально повторяющийся текст, а часть кода, решающая вполне определённую подзадачу.

```
// Заполнение массива случайными числами
void randomArray(int * array, int n)
{
  for (int i = 0; i < n; i++)
    array[i] = rand() % 100;
}

// Вывод массива на консоль
void printArray(int * array, int n)
{
  for (int i = 0; i < n; i++)
    cout << setw(4) << array[i];
  cout << endl;
}

void main()
{
  const int n = 10;
  int a[n];
  randomArray(a, n);
  printArray(a, n);

  int b[n];
  randomArray(b, n);
  printArray(b, n);
}
```

Правило 30: Для итерации нужно использовать наиболее подходящий вид оператора цикла.

В языке есть три вида операторов цикла – for, while и do-while (если не считать циклы по коллекциям). Они взаимозаменяемы, но не зря их три, а не один единый. Дело в том, что каждый из операторов предназначен для своего случая реализации циклического алгоритма:

Цикл for используют в тех случаях, когда количество итераций цикла известно заранее, ещё до начала его исполнения.

Цикл while используют тогда, когда количество итераций заранее неизвестно, а проверка того, нужно продолжать цикл либо его закончить производится до начала действия (тела цикла). То есть условие цикла – это ответ на вопрос «Нужно ли делать действие?»

Цикл do-while используют тогда, когда количество итераций заранее неизвестно, а проверка того, нужно продолжать цикл либо его закончить производится после выполнения действия (тела цикла). То есть условие цикла – это ответ на вопрос «Не получен ли в результате действия нужный результат?»

Выход из каждого цикла предусмотрен штатными средствами, поэтому использование для выхода из цикла оператора break или для выхода из очередной итерации оператора continue является плохим стилем.

Например, в следующем коде реализован алгоритм последовательного поиска значения в массиве. Однако, заранее неизвестно, на какой по счету итерации искомое значение будет обнаружено. Для принудительного прекращения дальнейшего поиска использован break, хотя цикл for в случае нормального исполнения должен естественно завершиться при достижении конечного значения индекса.

```
for (int i = 0; i < n; i++)
    if (a[i] == desiredValue) {
       cout << "Found index = " << i << endl;;
       break;
    }
```

Поэтому разумнее в данном случае использовать оператор while:

```
int i = 0;
while (i < n && a[i] != desiredValue)
  i++;
if (i < n)
  cout << "Found index = " << i << endl;
else
  cout << "Index not found" << endl;
```

Правило 31: Нельзя использовать goto.

Применение оператора goto противоречит принципам структурного программирования, так как нарушает естественный ход исполнения программных конструкций. Использование goto делает логику алгоритма запутанной и сложной для отладки.

Без него вполне можно обойтись: чтобы «перепрыгнуть ниже» по программе, то есть пропустить блок операторов можно использовать условный оператор, если нужно вернуться назад, то для этого есть цикл.

```
int n;
label:  cout << "Введите натуральное число" << endl;
cin >> n;
if (n < 1)
  goto label;
cout << "Вы ввели натуральное число " << n << endl;
```

Альтернативное решение без goto:

```
int n;
do {
  cout << "Введите натуральное число" << endl;
  cin >> n;
} while (n < 1);
cout << "Вы ввели натуральное число " << n << endl;
```

Правило 32: Следует избегать «магических» чисел.

«Магические» числа – это явно используемые в программе числовые значения. Когда такое число встречается в программе, то неочевидно, какой смысл оно несёт в её логике. Кроме этого, если от значения числа зависит поведение программы (а так чаще всего и бывает), то для изменения этого поведения придётся искать все вхождения «магического» числа в код и их синхронно править.

Типичные случаи использования «магических» чисел:

```
int a[16];   // 16 - магическое число!
...          // заполнение массива
int min = 100; // еще одно магическое число - 100
for (int i = 0; i < 16; i++)  // опять 16
  if (a[i] < min) // поиск min будет работать неверно,
```

```
min = a[i];// если в массиве все числа больше 100
```

Правильным будет определить и использовать именованные константы:

```
const int n = 16;
int a[n];    // размер массива задан константой
...          // заполнение массива
int min = INT_MAX;  // стандартная константа
for (int i = 0; i < n; i++)
  if (a[i] < min)   // поиск min будет работать
     min = a[i];    // правильно
```

Существуют исключения: 0 и 1 не считаются «магическими числами», их можно использовать явно, так как их смысл практически в любом контексте вполне понятен.

Правило 33: Код приложения разбивается на заголовочные h-файлы и cpp-файлы реализации.

Все компилируемые файлы с исходным кодом приложения компилируются отдельно и объединятся на этапе компоновки (сборки) исполняемого модуля. Файлы с кодом на C/C++, входящие в проект, бывают двух типов:

- заголовочные файлы (head-файл, h-файл, файл с расширением .h или .hpp) – содержит *объявления* используемых в программе сущностей (классов, типов, констант, функций и т.д.);
- файлы исходного кода (файл реализации, cpp-файл, файл с расширением .c или .cpp) – содержат *определения* сущностей (классов, функций и т.д.), то есть их реализацию.

Таким образом, заголовочные файлы объявляют интерфейс, файлы исходного кода его реализуют. Это не требование языка, это разумный способ организации программ, при котором программист всегда знает, где найти нужный ему код. Современные среды разработки по умолчанию предоставляют именно такой способ организации кода.

Заголовочные файлы должны включать в себя защиту от повторного включения. Ранее это делалось директивами **#ifndef** и **#define**, в современных версиях компиляторов такая защита реализована директивой **#pragma once**.

Пример заголовочного файла с объявлением класса Fraction:

```
// Файл Fraction.h
#pragma once
class Fraction
{
protected:
  int numerator, denominator;
public:
  Fraction(int numerator, int denominator);
  Fraction& operator= (const Fraction& right);
}
```

Реализация класса Fraction:

```
// Файл Fraction.cpp
#include "Fraction.h"

Fraction::Fraction(int numerator, int denominator)
{
  this-> numerator   = numerator;
  this-> denominator = denominator;
}

Fraction& Fraction::operator= (const Fraction& right)
{
  this->num = right.num;
  this->den = right.den;
  return *this;
}
```

Контрольные вопросы и задания темы 5

1. Почему необходимо минимизировать область видимости переменных?
2. В чем недостатки использования глобальных переменных?
3. Какие еще способы передачи данных функции (методу) существуют помимо использования глобальных переменных?
4. Почему константы можно определять глобально?
5. В чём заключается опасность побочного эффекта?

6. В чём заключается опасность использования значений переменных по умолчанию?
7. Когда переменную инициализировать не обязательно?
8. Какие ещё существуют причины выделения кода в отдельную функцию кроме сокращения текста программы?
9. А почему бы не использовать *один вид цикла* (например, while) во всех итеративных алгоритмах?
10. Укажите основные критерии для рационального выбора оператора цикла.
11. Может ли существовать алгоритм, который реализуется одним видом оператора цикла, но невозможно при этом использовать другой вид? Если существует – приведите пример.
12. Чем плох оператор goto?
13. Можно ли всегда обойтись без goto? Если нельзя – приведите пример.
14. Почему использование «магических» чисел является плохой практикой?
15. Можно ли обойтись без разделения исходного кода на .h и .cpp файлы? Обоснуйте.

Упражнения по теме 5

1. Напишите функцию swap, обменивающую два своих целочисленных параметра всеми (какими сможете) различными способами.
2. Напишите фрагмент программы, эквивалентный приведённому фрагменту (то есть работающий во всех случаях идентично), но с другим оператором цикла. Другой оператор цикла указан в комментарии.

```
// Нужен цикл while
for (int i = 0; i < n; i++) {
  if (i % 2 == 0)
    n++;
  if (i % 3 == 0)
    s += i;
}
// Нужен цикл do-while
max = INT_MIN;
cin >> n;
while(n != 0) {
```

```
  if (n > max)
    max = n;
  cin >> n;
}
// Нужен цикл for
while (a != b)
  if (a > b)
    a -= b;
  else
    b -= a;
// Нужен цикл for; нужен цикл while
int a = 1, b = 1, c;
do {
  c = a + b;
  a = b;
  b = c;
} while (c < n);
```

Задания для самостоятельной работы по теме 5

1. Выясните все возможные способы инициализации полей класса в языке C++ (помимо указанных выше).
2. Для каждого вида цикла сформулируйте правила эквивалентного преобразования его в другие виды циклов.

Заключение

Следование установленным правилам оформления программного кода – это не ограничение творческого начала в программировании. Напротив, это позволяет разработчику сконцентрировать внимание на смысле и эффективности алгоритма, а проблемы синтаксического и оформительского свойства уводит на второй план. Более того, стандарты и соглашения повышают эффективность разработки программного обеспечения, так как обеспечивают универсальный и общепонятный способ профессиональной коммуникации через внешнее единообразие программных конструкций.

Тема, изложенная в настоящем пособии, изложенным материалом не ограничивается, скорее здесь положено начало её освоения. Безусловно, во время дальнейшего изучения информатики, информационных технологий и программирования обучающийся должен развивать свои способности по

правильному применению и оформлению программных конструкций, документированию кода.

Список рекомендуемой литературы

1. Code Conventions for the Java TM Programming Language [Электронный ресурс]. Режим доступа https://www.oracle.com/technetwork/java/codeconvtoc-136057.html
2. Соглашения о написании кода на C# [Электронный ресурс]: руководство по программированию на C#. Режим доступа https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/inside-a-program/coding-conventions
3. РВН — Технология разработки программного кода [Электронный ресурс]. Режим доступа http://youinf.ru/standart-oformleniya-koda-na-yazyke-c
4. CS 106B: Programming Abstractions: [Электронный ресурс] Stanford University's Computer Science Department (C++) http://stanford.edu/class/archive/cs/cs106b/cs106b.1158/styleguide.shtml
5. Google C++ Style Guide [Электронный ресурс]. Режим доступа https://google.github.io/styleguide/cppguide.html
6. Mozilla Coding style [Электронный ресурс] Режим доступа https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style

Учебное издание

Анисимов Андрей Евгеньевич

Требования и рекомендации по оформлению программного кода на языках C и C++

Учебно-методическое пособие

Авторская редакция

Подписано в печать __.__.202_. Формат 60x84 1/16.

Усл. печ. л. ___. Уч.-изд. л. ___.

Тираж __ экз. Заказ № ____.

Издательский центр

«Удмуртский университет»

426034, Ижевск, Университетская, д. 1, корп. 4, каб. 207.

Тел./факс: + 7 (3412) 500-295. E-mail: editorial@udsu.ru