C. Требования к оформлению работ

Алексей Мартынов

28 января 2021 г.

Версия 3.0

Содержание

1 Оформление исходных текстов

1. Исходные тексты программы должны содержать не более одного выражения на одной строке.
2. Недопустимо писать функции в одну строку; сигнатура функции всегда должна идти на отдельной строке или нескольких, если сигнатура имеет большую длину.
3. Длина одной строки не должна превышать 120 символов в большинстве случаев и 140 символов всегда.
4. Строки не должны заканчиваться пробелами и/или символами табуляции.
5. Файл должен заканчиваться символом конца строки.
6. Все имена должны быть составлены из правильных английских слов или очевидных для читающего сокращений или аббревиатур.
7. Переменные определяются максимально близко к месту использования. Они должны иметь минимально возможную область видимости.
8. Все переменные должны быть инициализированы в точке определения.
9. Все неизменяющиеся переменные должны быть отмечены квалификатором **const**.
10. Аргументы функции, переданные по ссылке или по указателю и не изменяющиеся внутри функции, должны быть отмечены квалификатором **const**. Параметры, передаваемые по значению *могут* иметь квалификатор **const** в определениях функций, но он должен отсутствовать в объявлении функции.
11. Единицей отступа являются 2 пробела.
12. Отступы должны отражать логическую структуру программы:
    (a) операторы, выполняемые внутри циклов, должны иметь отступ на 1 больший, чем заголовок цикла;
    (b) операторы, выполняемые внутри **if** и **else**, должны иметь отступ больший, чем строки с **if** и **else**;
    (c) Метки операторов **case** внутри **switch** должны быть выравнены на тот же отступ, что **switch**.
    (d) Операторы внутри **switch** должны иметь на 1 больший отступ, чем оператор **switch**.
13. Все операторы, входящие в тела циклов и условий, заключаются в фигурные скобки, даже если это один оператор.
14. Фигурные скобки должны быть расставлены единообразно. Допускаются 2 варианта:
    (a) Открывающая фигурная скобка переносится на следующую строку и имеет тот же отступ, что и предыдущая строка, закрывающая скобка находится на отдельной строке, и ее отступ совпадает с открывающей. Операторы внутри скобок имеют дополнительный отступ. Например:

```
if (condition)
{
  c = a + b;
}
else
{
  c = a - b;
}
```

    Ключевое слово **else** должно находится на отдельной строке.
    (b) Открывающая фигурная скобка завершает строку с оператором и отделяется слева пробелом, закрывающая скобка находится на отдельной строке, и ее отступ совпадает с отступом строки, содержащей открывающую скобку (так называемые «египетские скобки»). Операторы внутри скобок имеют дополнительный отступ. Например:

```
if (condition) {
  c = a + b;
} else {
  c = a - b;
}
```

Ключевое слово **else** должно находиться на строке вместе с закрывающей скобкой.

Открывающая фигурная скобка в реализации функции *всегда* переносится на другую строку:

```
1 void foo()
2 {
3   ...
4 }
```

Способ расстановки фигурных скобок должен быть единообразен во всех работах.

15. Ключевые слова **if** и **else** могут быть объединены на одной строке для уменьшения отступов:

```
if (condition1) {
  c = a + b;
} else if (condition2) {
  c = a - b;
}
```

16. Ключевые слова **if**, **for**, **while**, **switch**, **catch** всегда отделяются пробелом от последующей открывающей скобки. В то же время список параметров при объявлении, определении и вызове функций никогда не отделяется пробелами.

17. Отдельные символы операторов в выражениях выделяются пробелами с 2-х сторон, кроме операторов:

    (a) индексации — квадратные скобки не отделяются пробелами от предшествующей переменной, а содержимое скобок не отделяется от них.

    ```
    a[1][index]
    ```

    (b) оператор последовательного выполнения следует грамматическим правилам английского языка — перед запятой пробел не ставится, а после — ставится, перенос строки выполняется строго после запятой;

    (c) запятая в списке параметров также следуют грамматическим правилам английского языка;

    (d) операторы доступа к полям составных объектов (`.` и `->`), а также к пространству имен `::`, не выделяются пробелами.

18. Двоеточие в метках **case** и переходах является частью синтаксической конструкции и не отделяется пробелом слева. Сразу за ним идет новая строка.

19. Двоеточие и точка с запятой в цикле **for** не должна отделяться пробелом слева, но после них обязательно идет пробел или перенос строки. В случае большого условия лучше определить λ-функцию или статическую функцию, чем переносить элементы заголовка цикла, так как перенос сильно ухудшает читаемость.

20. Имена переменных и параметров функций должны начинаться со строчной буквы и следовать одному из правил:

    (a) все слова пишутся строчными буквами и разделяются символами подчеркивания;

    (b) все слова, кроме первого, начинаются с заглавных букв и ничем не разделяются (аббревиатуры более 3 букв имеет смысл писать с заглавной строчными).

    Оформление должно быть однообразным во всех работах.

21. Имена классов должны начинаться с большой буквы. Если имя состоит из нескольких слов, они идут подряд без символов подчеркивания, каждое слово начинается с большой буквы (CapsCase). Например, `Queue` или `RoundedRectangle`. Исключением из правил могут являться функторы, имена которых могут быть в нижнем регистре. В этом случае отдельные слова в имени разделяются подчеркиваниями.

22. Имена классов должны представлять собой имена существительные с определениями и дополнениями. Исключения: имена функторов должны иметь в основе глагол, так как представляют собой действие.

23. Структуры в терминах C (без конструкторов и деструкторов, а также без методов) допустимо называть в нижнем регистре и разделять слова подчеркиваниями. В этом случае имя должно иметь суффикс «_t».

24. Секции внутри класса должны быть упорядочены в порядке убывания интереса к ним со стороны читающего код: первой должна идти секция **`public`**, так как это интерфейс класса для клиентов; затем секция **`protected`**, поскольку она представляет собой интерфейс для наследников; завершает класс секция **`private`**, так как эта информация интересна только разработчику класса.

25. Внутри каждой секции информация должна быть упорядочена следующим образом:

    (a) типы;
    (b) поля;
    (c) конструктор по умолчанию;
    (d) конструкторы копирования и перемещения;
    (e) все остальные конструкторы;
    (f) деструктор;
    (g) перегруженные операторы;
    (h) методы.

26. Имена функций и методов должны начинаться с строчной буквы, дополнительные слова должны начинаться с заглавной буквы и идти вместе с предыдущим (camelCase), например, `draw()` или `getArea()`.

27. Имена функций должны начинаться с глаголов, так как они представляют собой действие.

28. Имена параметров шаблонов должны начинаться с заглавной буквы, так как являются либо типами, либо специфическими константами.

29. Все методы, не меняющие объект логически, должны быть отмечены квалификатором **`const`**.

30. Имена полей класса должны следовать правилам для переменных и иметь отличительный признак поля, в качестве которого следует использовать либо префикс «m_», либо завершающий символ подчеркивания (предпочтительнее).

31. Недопустимо создавать имена, начинающиеся с подчеркиваний.

32. Имена макроопределений должны быть в верхнем регистре.

33. Перенос длинных операторов на другую строку должен выполняться следующим образом:

    (a) в начале следующей строки должен находиться перенесенный оператор (за исключением запятой);
    (b) отступ должен быть больше на 2 единицы отступа;
    (c) в случае переноса внутри сложного выражения с круглыми скобками, отступ должен отражать вложенность скобок, каждый уровень вложения добавляет 1 отступ на следующий строке.

    Пример переноса (ширина строки не соблюдена):

```
if (condition) {
  a = a + b
      * (c
        + d / e);
}
```

Исключение: в выражениях ввода и вывода с потоками допускается выравнивать операторы на перенесенных строка друг под другом, если это не создает слишком большого отступа:

```
std::cout << a << b
          << c << d;
```

34. Список инициализации в конструкторе должен инициализировать не более одной переменной или базового класса на строке. Двоеточие перед списком инициализации должно оставаться на строке со списком параметров, недопустимо переносить его на следующую строку. Оно не отделяется слева пробелом. Аналогично оформляются запятые, разделяющие элементы списка инициализации: запятая остается на строке с предыдущим элементом. Отступ в списке инициализации увеличивается на 1 по сравнению со строкой, где начинается определение конструктора (строки перенесены для демонстрации):

```
SampleClass::SampleClass(int a,
    int b):
  a_(a),
  b_(b)
{}
```

35. В списках инициализации массивов допускается применение «висящей» запятой после последнего элемента, так как это уменьшает «визуальный шум» в изменениях при увеличении списка инициализации.

36. Внутри выражений круглые скобки должны применяться для исключения неоднозначностей. За исключением очевидных из школьной арифметики приоритетов, все выражения оформляются так, чтобы при чтении не требовалось знать таблицу приоритетов операторов.

37. Не допускается применение «магических значений», вместо этого должны использоваться константы. Есть исключения: 0 и одноразово использованные строковые литералы.

38. Не допускается использование конструкций **`using namespace`** на верхнем уровне в заголовочных файлах.

39. Заголовочные файлы должны содержать header guards (разд. 5.2.3) независимо от наличия поддержки какого-либо другого механизма предотвращения повторного включения, так как все эти механизмы нестандартны и непереносимы.

40. Порядок включения заголовочных файлов важен:

    (a) внутри файла реализации первым включается соответствующий ему заголовок;

    (b) стандартные заголовки;

    (c) заголовки использованных библиотек;

    (d) свои заголовки.

41. Зависимости от заголовков должны быть минимизированы. Не допускается включение заголовков с реализациями, если достаточно объявлений (см. разд. 13.3.2).

42. В файлах реализации все функции должны иметь полные имена, в которых пространства имен и имена классов отделены от имени функции при помощи `::`. Например, для функции в пространстве имен `lab::detail` с именем `foo()` реализация должна выглядеть так:

```
void lab::detail::foo()
{
  ...
}
```

    Реализация функции в точке объявления допустима только для статических функций внутри единицы трансляции, шаблонных функций, методов шаблонных классов и однострочных методов, наподобие get-методов.

43. Форматирование λ-функций должно соответствовать оформлению обычных функций, с учетом того, что список захвата замещает название функции. Недопустимо писать их в одну строку.

```
[](const Object & obj) -> std::string {
  if (obj.getName().empty()) {
    return "<EMPTY>";
  } else {
    return obj.getName();
  }
}
```

44. Пустые строки должны использоваться разумно. Допускается использовать не более 1 пустой строки для:

    - разделения логических групп заголовков (п. 40);
    - отделения header guards;
    - разделения секций с различным доступом в классах (может быть вставлена *перед* ключевым словом, если оно не первое в определении класса;
    - разделения различных частей определения класса (п. 25);
    - отделения определений переменных от остального кода;
    - разделения определений классов, функций и т.п.;
    - отделения логически связанных фрагментов кода от других таких же фрагментов.

    Не следует отделять каждую переменную (или объявление функции и т.д.) пустой строкой, так как это излишне растягивает код.

    Для разделения классов и функций *не допускается* использование линеек из символов «минус» и «равно» для разделения блоков кода.

45. В программе не должно быть закомментированного кода. Весь неиспользуемый код должен быть удален. В случае необходимости, он может быть восстановлен из системы контроля версий.

Общий признак хорошего кода: его можно прочитать по телефону и он будет понятен на той стороне. Наиболее простым способом следовать этим правилам будет настроить редактор так, чтобы он выполнял такое форматирование автоматически.

2 Требования к дизайну

Общий дизайн выполненной работы должен удовлетворять следующим требованиям, в порядке убывания приоритета:

1. Использовать средства автоматического управления ресурсами (гл. 11)[1]
2. Удовлетворять как минимум базовой гарантии безопасности исключений, предпочтительно удовлетворять строгой гарантии (разд. 12.3.1)[1].
3. Каждый написанный класс должен перегружать все операторы, имеющие смысл для него. Например, класс матрицы должен перегружать операторы сравнения на равенство и неравенство, сложения, вычитания и умножения. Отношение матриц (больше или меньше) для матриц не могут быть однозначно определены и, соответственно, не могут определяться в виде перегруженных операторов.
4. Корректно обрабатывать ошибки при взаимодействии с внешним миром: ошибки ввода-вывода, некорректный пользовательский ввод и так далее.
5. Работы второго семестра прил. В должны корректно разбирать, а также выводить данные в соответствии с настройками пользователя (см.Потоки ввода-вывода и локализация).
6. Программы не должны содержать лишних сущностей. Например, создавать классы только для одного метода недопустимо. Дизайн должен быть минималистичным настолько, насколько это возможно.

   Также недопустимо создавать методы, идентичные создаваемым компилятором. Например, написание тривиального (пустого) открытого невиртуального деструктора недопустимо.

[1] Исключение: работы первого семестра А.1–А.3, так как соответствующий материал еще не прочитан.

7. Все детали реализации должны быть скрыты от клиентского кода. Если это шаблон, который невозможно скрыть, он должен быть помещен во вложенное пространство имен `detail`.

8. За исключением самых простых, работы не должны выполняться в рамках одной единицы трансляции (обычное исполнение типа «слабоотформатированный поток сознания»). Должно присутствовать разумное деление по файлам и правильно сформированные заголовочные файлы с минимальными зависимостями (гл. 13).

3 Общие требования к работам

Работы выполняются в рамках стандарта C++ 14. Это означает, что код должен компилироваться и работать на любой платформе: Microsoft Windows®, Linux, macOS. Это автоматически означает, что все платформенно-зависимые вещи, обычно добавляемые в проект Microsoft Visual C++®, должны быть исключены.

При компиляции программы должны отсутствовать предупреждения. Для GNU Compiler Collection (GCC)/Clang уровень предупреждений задается ключами -Wall -Wextra -Wold-style-cast. Часть предупреждений выключена, так как представляет собой несколько параноидальный подход (-Wno-missing-field-initializers).

Программы должны представлять собой консольные приложения, обрабатывающие параметры командной строки и использующие стандартные каналы ввода-вывода. Тестирование заданий *не предусматривает* интерактивного взаимодействия, поэтому все входные данные должны читаться либо до заданного разделителя, либо до конца файла (End Of File (EOF, конец файла)). Ошибки ввода должны немедленно обрабатываться и программа должна завершиться с ненулевым кодом возврата, так как исправлять ввод при неинтерактивном взаимодействии невозможно. Стандартный набор кодов возврата:

0 Успешное завершение программы.

1 Некорректные параметры. Обычно, в стандартный поток ошибок должно быть выведено описание причины ошибки.

2 Внутренняя ошибка программы. В стандартный поток ошибок также следует вывести описание ошибки.

Работы выполняются с использованием системы контроля версий [GIT]. Для работы с системой требуется зарегистрироваться на сайте https://gitlab.com/. После регистрации на сервере необходимо прислать преподавателю свой логин для добавления в проект (конкретное название проекта зависит от преподавателя).

Каждый студент размещает свои работы в директории со своим именем на английском языке, строчными буквами и разделив фамилию и имя точкой, например, lastname.firstname. Каждая работа должна находиться в директории, имя которого представляет собой объединение заглавной латинской буквы, обозначающей семестр («A» или «B») и номера работы, например, lastname.firstname/A1 для первой работы первого семестра или lastname.firstname/B3 для третьей работы второго семестра.

Каждый коммит должен:

- содержать логически единое изменение;
- обладать комментарием, из которого понятно что (в первой строке) и зачем (описание после пустой строки) было сделано (с обоснованием и деталями рекомендации можно ознакомится в статье https://chris.beams.io/posts/git-commit);
- успешно компилироваться и компоноваться;
- проходить unit-тесты, если они есть;
- крайне желательно проходить приемочные тесты.

Категорически не допускается «брать штурмом» систему, создавая множество коммитов путем подбора символов в исходных программах, которые смогут, наконец, скомпилироваться. Бесплатное использование инфраструктуры для тестирования допускает не более 2000 минут в месяц, поэтому лишние попытки завершатся исчерпанием лимита и пропуском сроков сдачи работ.

На проекте используется Continuous Integration (CI), соответственно, сборка и тестирование выполняются автоматически. Коммиты, не прошедшие CI (с ошибкой или пропустившие CI) не принимаются. Результаты тестирования доступны в виде архива артефактов сборки и могут быть получены по ссылке «Download acceptance artifacts» Web UI, указанной либо в коммите, либо в CI pipeline.

Категорически не допускается добавление в проект не относящихся к нему файлов, а также построенных программ, промежуточных файлов и файлов, создаваемых средами разработки.

Построение программ выполняется при помощи стандартного Makefile, находящегося в корне репозитория. Не допускается его изменение, а также любых файлов в корне проекта.

Идентификатором каждой работы является строка «lastname.firstname/labnumber», на которую в дальнейшем ссылаются при помощи «labid».

Все исходные тексты в каждой работе идентифицируются по расширению «cpp». Обнаруженные исходные тексты делятся на группы:

Исходные тексты работы все файлы, имена которых *не* начинаются с test-.

Исходные тексты тестов все файлы, исключая файл main.cpp.

Как видно, файл main.cpp стоит особняком: его назначение — функция `main()` выполненной работы.

Makefile автоматически строит программу для каждой работы. Так как имя программы в общем случае неизвестно, специальная цель выполняет запуск построенной программы с передачей параметров.

Общие файлы для нескольких работ должны размещаться в поддиректории common. Они будут автоматически добавлены при сборке работы. Внутри common допустимо создавать поддиректории для организации файлов.

Поддерживаемые цели:

build-labid построение лабораторной работы, например

```
$ make build-ivanov.ivan/2
```

run-labid запуск построенной программы, например

```
$ make run-ivanov.ivan/2
```

Для передачи дополнительных параметров используется переменная ARGS (при помощи GNU Make):

```
$ make run-ivanov.ivan/1 ARGS="1 ascending"
```

или (с использованием Bourne Shell):

```
$ ARGS="1 ascending" make run-ivanov.ivan/1
```

или (Bourne Shell, с сохранением в окружении процесса):

```
$ export ARGS="1 ascending"
$ make run-ivanov.ivan/1
```

test-labid сборка и запуск тестов работы:

```
$ make test-ivanov.ivan/3
```

Переменная TEST_ARGS используется для передачи параметров тестам аналогично ARGS.

labs список всех лабораторных в проекте.

Дополнительной возможностью является запуск динамического анализатора [VALGRIND] для запускаемых программ. Для этого необходимо указать в переменной VALGRIND параметры анализатора так, как это делается для ARGS/TEST_ARGS.

Библиотека [BOOST] может находиться в нестандартном месте. Для того, чтобы указать его, в корне проекта необходимо разместить файл .boost_location, состоящий из одной строки с путем к корню библиотеки. Путь не должен содержать пробельные символы.

Сдача работы осуществляется путем отправки Pull или Merge Request. Сроком представления считается дата отправки запроса. Все содержимое запроса принимается или отклоняется целиком. В случае успешного приема работы запрос объединяется с веткой «master».

Все работы выполняются строго последовательно от первой до последней. Представлять, например, 6 работу можно только после успешной сдачи первых пяти.

Так как каждая работа должна быть выполнена в отдельном Merge Request, не допускаются следующие ситуации:

- более одной работы внутри Merge Request, но, если работа требует исправления уже сданных работ, такие изменения *должны* быть выполнены в том же Merge Request, так как поддерживают корректность проекта;

- несколько Merge Request для одной и той же работы, за исключением предыдущего пункта, так как каждый Merge Request обеспечивает отслеживание истории изменений и приема работы.

Изменения в Merge Request должны быть выполнены относительно самого свежего коммита в ветке «master» основного проекта. Достигается это операциями «merge» и «rebase» (см. документацию по Git).

Условия, гарантирующие отклонение работы (преподаватель может даже не увидеть работы, так как отклонение происходит автоматически):

- неправильное именование директории с работой;
- наличие конфликтов при слиянии;
- наличие лишних файлов, создаваемых средой разработки и/или компилятором, не нужных для сборки и тестирования работы;
- невыполнение настоящих требований к оформлению работ;
- получение сообщений от [VALGRIND].

Список литературы

[BOOST] *Boost C++ Libraries*. URL: http://www.boost.org/.

[GIT] *Git*. URL: https://git-scm.org/.

[VALGRIND] *Valgrind*. URL: http://valgrind.org.