Методические рекомендации к практическим занятиям по дисциплине «Алгоритмы и структуры данных»

Осенний семестр 2017/18 уч. г.

С. А. Шершаков

11 сентября 2017 г.

В документе представлены методические рекомендации, договоренности по стилистическому оформлению программного кода, соглашения об именовании объектов, требования, предъявляемые к разрабатываемым программам, и др. полезная информация для участников дисциплины «Алгоритмы и структуры данных» образовательной программы бакалавриата «Программная инженерия».

Ред. 1.3.0 от 11.09.2017 г.

Является предметом дополнения и редактирования. Версия считается актуализированной в течение трех дней после опубликования на официальном ресурсе, ассоциированном с дисциплиной «Алгоритмы и структуры данных». В случае внесения в документ незначительных исправлений, касающихся описок, неточностей и других сведений, не имеющих временно́й значимости, вступает в силу немедленно после опубликования с соответствующей пометкой.

1 *Область применения*

Данный документ[1] является дополнением к программе дисциплины «Алгоритмы и структуры данных»[2], уточняющим и дополняющим программу в некоторой части организации семинарских занятий[3].

Методические рекомендации предназначены для преподавателей, учебных ассистентов и студентов, относящихся к дисциплине «Алгоритмы и структуры данных» (1 часть, 1–2 модуль 2017–2018 уч. года).

2 *Организация семинарского занятия*

Семинарские занятия по дисциплине проводятся согласно учебному расписанию еженедельно в количестве одной академической пары часов на подгруппу. Каждое семинарское занятие проводится в рамках единого среди всех подгрупп потока цикла изучения предметной темы[4].

Посещение студентами семинарских занятий не со своей подгруппой не допускается.

На семинарском занятии преподаватель предлагает студентам задание[5] на семинар и предоставляет необходимые пояснения по выполнению задания. Выполнение задания начинается во время семинарского занятия и должно быть завершено до окончания срока, указанного для каждого проекта индивидуально[6].

Семинарское занятие может сопровождаться соответствующим проектом в системе LMS[7]. Результатом выполнения студентом задания является один или более файлов и/или записей на бумажных носителях информации. Файлы в обязательном порядке прикрепляются к соответствующему проекту LMS, а бумажные носители сдаются в установленный срок. С целью упрощения организации проектов LMS студенты разных подгрупп могут быть прикреплены к единому проекту.

[1] Далее — Методические рекомендации

[2] Программа дисциплины «Алгоритмы и структуры данных» (часть 1, модули 1–2) для направления 09.03.04 «Программная инженерия» подготовки бакалавра // Дегтярев К. Ю., к.т.н., доцент ДПИ ФКН (далее Программа дисциплины)

[3] Документ разрабатывается так, чтобы не противоречить Программе дисциплины. В случае обнаружения несоответствий между Программой дисциплины и Методическими рекомендациями приоритет следует отдавать Программе дисциплины.

[4] По состоянию на 08.09.2017 г. цикл начинается с лекционного занятия в пятницу на неделе, предшествующей неделе семинарских занятий, которые приходятся на понедельник и вторник следующей недели.

[5] Может быть представлено в электронном и/или бумажном виде. Основное задание может включать ссылки на предыдущие задания, дополнительные материалы (стандарты, ремомендации, примеры и др.).

[6] Может выходить за рамки занятия.

[7] `lms.hse.ru`

В случае пропуска студентом срока сдачи результатов работы, работа считается невыполненной.

Срок выполнение каждого проекта устанавливается индивидуально. Разница в абсолютном времени, отведенном разным группам на выполнение проектов одной серии, не является предметом для аппеляции.

2.1 Текущий контроль знаний

Текущий контроль включает оценку за выполнение индивидуальных работ, назначаемых студентом на семинарских занятиях.

Базовой оценкой — в случае верного выполнения студентом задания в полном объеме, в указанные выше сроки и с учетом удовлетворения базовым требованиям к программного коду — является **8 (отл)**. В дополнении к базовой оценке преподавателем могут быть начислены дополнительные баллы при выполнении следующих условий[8], но не ограничиваясь ими:

1. работа сдана студентом до окончания семинарского занятия[9];
2. работа удовлетворяет дополнительным требованиям к программному коду, не являющимся обязательными к исполнению на момент сдачи[10];
3. работа включает выполнение дополнительного необязательного задания, если таковое предложено[11];
4. при решении задачи одновременно выполнены следующие условия: разработанный код является хорошо декомпозированным на подзадачи, лаконичным (необходимо кратким), хорошо читаемым, оптимальным в некотором поле оптимальных решений;
5. на основании иных причин, позволяющих с достаточной степенью уверенности считать работу заслуживающей дополнительных баллов.

Безусловными основаниями для снижения оценки вплоть до неудовлетворительной являются следующие[12]:

1. задача выполнена неверно или не в полном объеме;
2. на некоторых наборах входных значений из верного множества допустимых значений программа демонстрирует некорректное поведение;
3. разработанная программы генерирует **непользовательскую** исключительную ситуацию, указывающую на ошибки в проектировании;
4. разработанный интерфейс программы не соответствует предъявляемым требованиям к тестопригодности;
5. разработанная программа предъявляет объективно завышенные требования к ресурсам для выполнения задачи со стандартным набором выходных данных[13];

[8] Список **не является** исчерпывающим. Предложенные критерии **не являются** достаточным основанием для повышения оценки.

[9] Данный факт определяется по временной отметке загрузки работы в LMS. В случае обнаружения студентом неточностей в первоначальной редакции, исправленной в последующих редакциях, временная отметка загрузки исчисляется **моментом загрузки последней редакции** работы вне зависимости от объема и важности сделанных изменений. Таким образом, необходимо признать приоритет за тщательной проверкой работы перед загрузкой работы в максимально ранние сроки.

[10] Такие требования индивидуально указываются на семинарах/доп. материалах к ним.

[11] Индивидуально для каждого семинара.

[12] Список не является исчерпывающим.

[13] Симптомом может является выход за диапазоны по времени исполнения/затрачиваемой памяти по сравнению с эталонным решение и/или решениями других студентов.

6. иные причины, позволяющие с достаточной степенью уверенности считать работу заслуживающей снижения оценки по ней.

Данные критерии не учитывают *организационные причины снижения оценки*, к которым в частности относятся *пропуск сроков сдачи* и *плагиат*.

Проставление оценки по текущему контролю знаний осуществляется преподавателем семинарских занятий, его/её коллегой в случае замены и/или ответственным за курс преподавателем. Оценка по текущему контролю считается объявленной после опубликования ее с использованием соответствующего информационного ресурса[14].

[14] Будет указан дополнительно.

2.1.1 Коэффициент сложности работы

В зависимости от сложности и объемности предполагаемого решения для каждого индивидуального задания вводится *коэффициент сложности*, учитываемый при расчете итоговой оценки за индивидуальные задания в виде простой взвешенной суммы.

Коэффициент сложности для текущего проекта объявляется в начале очередного цикла семинарских занятий и фиксируется в сводной ведомости.

2.2 Плагиат

Каждый студент выполняет свою работу **полностью самостоятельно**. Недопустимо: обмен кодом в каком бы то ни было виде с использованием любых инструментов (форумы, тематические группы, репозитории, доски обмена и т.д.), использование отдельных фрагментов программ других студентов, обращение за получением помощи по выполнению заданий на любой основе к любому третьему лицу. Подобные факты являются нарушением академических норм, принятых в университете.

Все работы проходят систему электронного антиплагиата, а также независимо от результатов антиплагиата анализируются учебными ассистентами и преподавателями. В случае обнаружения признаков плагиата в зависимости от уровня достоверности указанного нарушения работа может признана *отбракованной* или *подозрительной*.

Работа признается *отбракованной* в случае наличия достаточных оснований полагать факт плагиата установленным.

В случае *подозрительной* работы студенту — автору работы до объявления оценки предлагается осуществить *защиту* перед комиссией в составе преподавателей и учебных ассистентов. В случае опровержения факта плагиата и демонстрацией студентом уверенных знаний по работе, исключающих возможности получения помощи при выполнении от третьего лица, работа оценивается на общем основании.

В противном случае, а также в случае *отбракованной* работы студент получает оценку «ноль» с пометкой «плагиат» с возможностью дальней-

Выявление признаков нарушения академических норм регулируется следующими нормативными документами ВШЭ: *Устав университета* http://www.hse.ru/docs/26598076.html, *Положение об организации контроля знаний* http://www.hse.ru/docs/35010753.html, *Правила внутреннего распорядка* (см. также прил. 7 к нему) http://www.hse.ru/docs/48094015.html. Вовлеченные лица подлежат дисциплинарному взысканию вплоть до отчисления с учебной программы и из университета.

шей передачи дела на рассмотрение административной комиссии по академической этике.

В случае установления факта наличия нескольких идентичных работ, установление доноров и реципиентов работ не производится, все работы признаются одинаково *отбракованными*.

2.3 Учебные ассистенты

Сопровождение семинарских занятий осуществляется при содействии учебных ассистентов (УА).

Учебные ассистенты оказывают преподавателям помощь при организации аудиторных занятий, проверке самостоятельных работ студентов, подготовки учебных материалов, тестов и контрольных мероприятий, осуществляют контроль во время проведения экзаменационных мероприятий[15].

Учебные ассистенты не наделены правом проставления любого вида промежуточных и/или окончательных оценок в результате контроля полученных студентами знаний. Все рекомендации, вырабатываемые учебными ассистентами, являются внутренней информацией и не могут ни в каком виде служить основанием для студентов апеллировать к результирующей оценке, выставленной с учетом указанных рекомендаций.

Прикрепление учебных ассистентов к группам не осуществляется. Распределение работа на проверку УА осуществляется для каждого проекта индивидуально.

По состоянию на 11.09.2017 г. сопровождение дисциплины осуществляется следующими УА:

- Артур Антонов (email).
- Александр Варгулёв (email).
- Данил Нечай (email).
- Александр Плесовских (email).

[15] Виды активности учебных ассистентов представлены в соответствующем положении: http://www.hse.ru/docs/167318558.html

3 *Основные требования к инструментам разработки*

Разработка индивидуальных проектов осуществляется на языке программирования C++ стандарта 11. Полнота определяемых стандартом возможностей, которые можно использовать в решении, должна соответствовать используемого референтного компилятора.

Референтной IDE[16] для практических занятий является Microsoft Visual Studio 2013 (версия компилятора C++ — vc12[17]). Референтные компиляторы: MS VC120, gcc 4.8.1 (режим совместимости `gnu++11`).

Несовместимость между используемыми студентом версией языка (с учетом реализованного подмножества), IDE и компилятора с указанными в настоящем документе референтными версиями не является удовлетворительным основанием для пересмотра оценки в большую сторону.

Студенты должны соблюдать предложенную для каждого индивидуального проекта структуру рабочего каталога.

Заготовка для проекта может опционально включать помимо заготовки модулей исходных кодов проекты решений для референтной IDE/компилятора, а также для других систем сборок, например CMake.

Оформление программных проектов и исходного кода должно осуществляться в соответствии с рекомендациями, изложенными в настоящем документе. Если заготовки кода включают элементы, оформленные не в сооветствии с настоящими рекомендациями, они не должны изменяться студентами, если иное не оговорено в задании к индивидуальному проекту[18]. Тем не менее, в части кода, которую студент дописывает самостоятельно оформление должно осуществляться в соответствии с настоящими требованиями.

[16] Использование других IDE допустимо, однако любые их специфические особенности, не совместимые с референтной IDE, могут повлиять на невозможности компиляции приложения для проверки и, как следствие, повлечь получение низкой оценки.

[17] Поддерживаемые Visual Studio 2013 возможости стандарта 11 перечислены на сайте `https://msdn.microsoft.com/ru-ru/library/hh567368.aspx`

[18] Причина появления таких несоответствий состоит в том, что адаптация существующих заданий к новым требованиям осуществляется поэтапно, и некоторые задание все-еще могут быть оформлены в соответствии с устаревшими требованиями.

4 Работа с кодом

4.1 Общие требования к оформлению кода

1. Размер отступа — 4 пробельных символа.
2. Использование символов табуляции запрещено (настроить в редакторе опцию «вставлять пробелы»).
3. «Правило структурной распечатки кода»: код должен быть оформлен с учётом отступов соответствующих элементов, отображающих уровень вложенности объектов, их область видимости и время существования (object lifetime).
4. Традиционным исключением из правила отступов является заключение кода некоторого модуля внутрь `namespace`-ов: как-правило, неймспейс открывается один раз в начале модуля и закрывается в конце, поэтому для экономии пространства его тело не отбивается.
5. Открывающая фигурная скобка всегда идет с новой строки (возможные исключение: inline однострочные функции вида `int getValue() return _value;`).
6. Примеры рекомендуемого использования оступов см. в листинге 1 и листинге 2.

Листинг 1. Примеры правильной структурной распечатки описания класса.

```
1  namespace MyNamespace { namespace MyNestedNamespace     //
2
3  class MyClass                                           //
4  {
5  public:                                                 //
6      const int MY_CONTANT_LONG_NAME = 42;
7      const int MY_SIMPLE_CONTANT    = 42;                //
8  public:
9      MyClass();
10     ~MyClass();                                         //
11 protected:
12     /** \brief Doxygen—java style comment for a method.
13      *
14      *  This is verbose description.
15      *  Discussing purpose of \param a,
16      *  continue discussing \returns function return.
17      */
18     double foo(int a);
19 private:
20     /** Doxygen—style comment for a field */
21     int _a;
22
23     // simple comment
24     double _fieldLongName;                              //
25 }; // class MyClass
26 }}; // namespace MyNamespace  namespace MyNestedNamespace
```

Исключение: в C++ традиционно оступы внутри namespace-ов не делаются.

Открывающая скобка на новой строке (допустимо оставлять на строке `class`).

Модификаторы области видимости не отбивать

Обычно константы выравнивают по значениям

Правилом хорошего тона является начинать строковые комментарии с одной (максимум двух) горизонтальных позиций (табуляций). Это позволяет визуально легче отделять код от второстепенной информации.

Так как поля/методы и т.д. обычно сопровождаются семантическими комментариями, появление их на соседних строках — редкость, и поэтому их обычно не выравнивают, как в случае с константами.

Листинг 2. Примеры правильной структурной распечатки определения метода.

```
1  int foo(double a, int b)
2  {                                                       //
3      double x = a * b;
4                                                          //
```

Откр. скобка тела метода всегда с новой строки и на ней только скобка.

Группировка лог. связанных частей кода отбивкой пустыми строками.

```
5       for(int i = 0; i < b; ++i)
6       {                                                   //
7           if(x < a * a)
8               bar(i, b);                                  //
9           else
10          {
11              int y = b + i;                              //
12              double z = bar(y, b);
13
14              if(z < x)
15              {
16                  if(z * z > a)                           //
17                      return 42;
18              }
19              else
20              {
21                  if(z * z * z > a)
22                      return 43;
23                  else
24                      return 44;
25              }  // if(z < x)
26          }  // if(x < a * a)
27      }  // for
28  }
```

{ для циклов, логических блоков — всегда с новой строки

Однострок. инструкции без { }. Возможны исключения для повышения читаемости кода (вложенные однотипные блоки и др.)

Операторы =, +, – и т.д. окружат одиночными пробелами

Вложенные однотипные блоки — для устранения неоднозначности и повышения читабельности кода.

Слишком много закр. скобок — лучше показать, к чему они относятся с помощью комментариев, особенно в случае «длинных» блоков, не помещающихся целиком на один стандартный экран.

4.2 Правила именования элементов кода

1. Объекты необходимо именовать осмысленным образом, повышая тем самым удобочитаемость кода за счет более быстрого погружения в контекст[19].
2. При именовании функций и методов ожидаемой частью идентификатора является глагол, описывающий кратко поведение метода[20].
3. Длинные идентификаторы затрудняют восприятие кода, поэтому их по возможности следует избегать, используя акронимы и сокращения.
4. Переменные и методы именуются с помощью `camelStyle`.
5. Название типов (новых, своих) именуется с помощью `PascalStyle`. Варианты с приписками букв `_t` в конце или `t_` в начале не применять.
6. Название констант задается `БОЛЬШИМИ_БУКВАМИ_С_ПОДЧЕРКИВАНИЕМ`.
7. Именование закрытых (private) и защищенных (protected) полей начинать с подчеркивания: `_field`. Префиксы вида `m_`, `f_`, а также постфиксы не использовать.
8. Статические методы (и поля) специально не выделять наименованием.
9. Венгерскую нотацию с типами на употреблять.
10. При именовании шаблонов `typename` выглядит предпочтительнее, чем `class`, в объявлении параметров типа; если тип один, его принято называть `T`, если больше — лучше называть понятным словом с префиксом `T` (для примера смотри листинг 4).
11. Примеры рекомендуемого именования см. в листинге 3.

[19] Переменные можно условно разделить на *технические* — например, для организации цикла — и «логические» — для обозначения сущностных объектов. Для организации цикла допустимыми являются обозначения `i`, `j`, `it` для итераторов. Для обозначения компонентов простых векторов — `a`, `b`, `c`, `x`, `y`, `z` и т.д. Однако для обзначения переменной «таблица сотрудников» идентификатор `a` представляется неудачным; лучшим вариантом является использование акронима `tblEmployees`, если в текущем контексте есть несколько таблиц, либо `emplTable`, если помимомо таблицы сотрудников есть переменные еще каких-то объектов, связанных с «сотрудниками»

[20] `getX()`, `setY()`, `calcSum()`

Листинг 3. Примеры написания названий объектов, методов, типов.

```
1  double myIntegerObject;                                  //
2  // ...
```

camelStyle

```
3  int foo()                                          //   camelStyle
4  {
5      return 42;
6  }
7  // ...
8  class MyClass {                                    //   PascalStyle
9  public:
10 public:
11     const int MY_CONSTANT = 42;                    //   ALL_CAPITAL
12 public:
13     MyClass();                                     //   Constructor, same as the class name
14     ~MyClass();                                    //   Destructor, same as the class name
15 public:
16     double calcSquare() const { return _a * _a; }
17     static int saySomeSpiritual() { return 42; }   //   Nothing special for static
18 public:
19     int getA() const { return _a; }                //   getter
20     void setA(int a) { _a = a; }                   //   setter
21 protected:
22     int _a;                                        //   not public member field
23 private
24     static long _b;                                //   not public static member field
25 }; // class MyClass
```

Листинг 4. Примеры именования элементов шаблона.

```
1  template<typename T>
2  class...
3
4  ...
5
6  template<typename TElement, typename TAllocator>
7  class...
```

4.3 Работа с модулями и рабочими файлами и каталогами

1. Именование файлов исходных кодов осуществляется только с использованием латинских букв, цифр и знаков подчеркивания (для разделения слов) в нижнем регистре[21]. Использование пробелов, русских букв и других специальных символов в именах каталогов и файлов недопустимо.
2. Не рекомендуется злоупотреблять практикой «один тип на один модуль cpp/h» (лучше вообще ее избегать)[22].
3. Каждый заголовочный файл должен содержать `#ifndef...` guard[23], включающий название проекта и названием файла (с путем, если он есть). Для примера см. листинг 5.
4. Каждый модуль представляется в виде пары одноименных файлов cpp/h (или cpp/hpp, cxx/hpp). Определение типов в cpp за исключением некоторых особо специальных случаев (тесты, как правило), запрещено. В заголовочном файле исполняемый (не декларационный) код допустим только в inline-функциях, либо в шаблонах.
5. Заголовочный файл без сопутствующего cpp (модуля трансляции) используется только для описания интерфейсов и шаблонов.

[21] Требование продиктовано тем, что разные ОС по-разному относятся к уникальности имен с большими и маленькими буквами.

[22] С учетом особенностей языка (частое предварительное описание типов, важность порядка представления типов), могут возникать конфликты при различном порядке появления определения заголовков с типами при компиляции.

[23] Использование директивы `#pragma once` не является достаточным.

6. Для файлов исходных кодов (src), временных файлов (intermediate, objects и libs) и результирующих двоичных файлов (exe, dll) должны предусматриваться отдельные каталоги — подкаталоги папки проекта. Неупорядочное расположение разнотипных файлов в одном каталоге недопустимо.

Листинг 5. Примеры использование guard для заголовочного файла.

```
#ifndef MYPROJECT_PATH_MYHEADER_H_   //
#define MYPROJECT_PATH_MYHEADER_H_

// полностью хидер

#endif  // #define MYPROJECT_PATH_MYHEADER_H_
```

Здесь `MYPROJECT` — название проекта, `PATH` — путь к хидеру относительна корня исходников, `MYHEADER` — название хидера

4.4 Пространства имен

1. Использование `using namespace` (`std`) где-либо, кроме тела функции, запрещено[24].
2. При разработке пользовательских проектов (особенно библиотек) рекомендуется использовать уникальное одноуровневое пространство имен.

[24] Это ведет к засорению пространства имен, а в случае помещения такой директивы внутри заголовочного файла — этот негативный эффект проявляется во всем графе включения его другими заголовочными файлами.

4.5 Работа с памятью и указателями

1. Появление неинициализированных (подвешенных) указателей в коде категорически запрещено![25]
2. Исключение: когда указатель невозможно сразу инициализировать адресом; в таком случае его необходимо инициализировать явно нулем (см. листинг 6).
3. Идеологически, символ * при имени типа делает из типа — тип, указатель на тип. Технически, эта звездочка, конечно же, относится к объекту[26]. **По договорённости**, звездочка «типоуказателя» всегда будет «прижиматься» к имени типа (см. листинг 7).
4. Определение нескольких объектов-указателей в одной инструкции не применять (см. листинг 7)[27].
5. Для управление динамической памятью необходимо использовать операторы `new` и `delete` **вместо** C-функций `malloc()`, `free()`, `memcpy()` и др[28]. Создание и копирование «сложных» объектов (с конструкторами) с помощью этих функций недопустимо, т.к. в этом случае конструкторы не будут вызваны и, следовательно, новый объект-реплика при создании повторяет побитовое состояния своего оригинала, что в общем случае неверно.

[25] Пример:
`int* a;`
страницей позже программист использует `a` в rvalue-позиции и получит заслуженный Access Violation.

[26] Исторически со времен языка C и требований к обратной совместимости.

[27] Такой стиль следует признать неудачным. Во-первых, мы хотим «указатель на инт», а не «интовый объект-указатель», во-вторых, вторая нотация часто провоцирует появление подвешенных указателей, что недопустимо.

[28] Мы используем язык C++, а не C-с-классами.

Листинг 6. Пример, демонстрирующий невозможности инициализации указателя сразу при определении.

```
int* p = nullptr;                                    //
if…()
```

Обязательная инициализация!

```
{
    // 5 страниц кода
    p = ...
}
else
{
    // 6 страниц кода
    p = ...
}
```

Листинг 7. Куда прижимать звездочку при определении указателя.

```
int*    pa = ...        // делать надо так! //
...
int    *pa = ...        // так делать НЕ надо!
...
int *a, *b;             // так делать НЕ надо!
```

К вопросу, куда «прижимать» звездочку

4.6 Некоторые рекомендации по использованию исключений

1. Использование исключений — достаточно «дорогая» операция в C++, которую надо использовать крайне аккуратно[29].
2. В каждом конкретном задании необходимо обращать внимание на политику применения исключений, если она специально там оговаривается.
3. Всю разрабатываемую программу рекомендуется заключать в блок обработки исключительных ситуаций вне зависимости от того, генерируются ли исключения автором программы или библиотекой (см. пример на листинге 8).

[29] Мнемоническое правило: исключительная ситуация должна генерироваться только в исключительной ситуации.

Листинг 8. Пример обработчика исключительных ситуаций уровня приложения.

```
int main(..)
{
    int res;
    try {
        res = main2();
    }
    catch(SpecificType) { ... }
    catch(SpecificOtherType) { ... }
    catch(...) { ...; throw; }                          //
    ...
    } // try

    return res;
}

int main2(..)
{
// .. решение задачи
}
```

Если кодом полностью не закрывается вопрос обработки исключения (например, вывод стека вызовов в отладочный файл), лучше выполнить rethrow на обработчик по умолчанию, по крайней мере в отладочной сборке (исп. для этого директивы препроцессора).

4.7 Некоторые замечания по определению классов

1. Порядок следования секций в классе — от public к private, от методов к полям, от нестатических к статическим[30].

[30] Мотивация: интерфейсная часть типа (а это паблик) должна быть доступна для чтения как можно раньше.

2. Открытые поля в классах, как правило — нарушение принципа инкапсуляции (исключение — примитивные структуры). Даже в простейшем случае необходимо создавать тройку: поле + аксессоры (см. описание класса в листинге 3)[31].
3. Если некоторому типу, описываемому классом, недостаточно побитного копирования, выполняемого компилятором по умолчанию, одно из следующих условий должно иметь место: либо есть эксплицитный конструктор копирования/операция присваивания, либо они в виде болванок помещены в protected так, что присваивание (и инициализация копированием) будет является недопустимой операцией[32].
4. Интерфейсы (классы с только чисто виртуальными функциями) и абстрактные классы должны либо иметь дефолтный публиковый виртуальный деструктор, если предполагается удалять полиморфные объекты через указатель на базовый тип, либо деструктор в protected-секции, чтобы через полиморфный базовый тип можно было только эксплуатировать интерфейсные методы, но не управлять временем жизни объекта. Все другие варианты завсегда ведут к ошибкам времени исполнения.
5. Если некоторая функция объявлена с ключевым словом `virtual` в базовом классе, необходимо также использовать это ключевое слово[33] и для перекрытых функций в производных классах для улучшения читабельности кода, несмотря на то, что это технически необязательно.
6. Если некоторый метод класс константный по сути (например, возвращает значение некоторого выражения, которое рассчитывается без изменения состояния), он обязан быть объявлен с модификатором `const`[34], т.к. противное исключает возможность использования его в других константных методах.
7. Метод, возвращающий ссылку на некоторый стационарный объект (например, поле класса), должен обязательно иметь константный перегруженный вариант, чтобы можно было использовать его в константных r-value позициях (см. листинг 9).

Листинг 9. Использование перегруженной константной версии метода.

```
1  ... //
2  public:
3      std::string& getName() {return _name;}
4      const std::string& getName() const  {return _name;}
5  private:
6      std::string _name;
7
8  ...
9  // в этом случае первый вариант будет использоваться только в
       l-value-позициях, например таких:
10 getName() = "Name";
```

[31] Так как код за программиста сегодня пишут по большей части IDE, проблем с автоматическим созданием этого быть не должно. Также предлагается использовать такое именование сеттеров/геттеров. Для булевых типов рекомендуется использвать такую модификацию геттера:
`bool isState() {return _stateFlag; }`

[32] Известно как «правило трех» и «правило пяти» при учете семантики перемещения. При проверке работ будет осуществляться попытка копирования объекта, описываемого пользовательским классом. Если при этом будет осуществляться копирование указателей без выделения памяти или передачи прав владения, это будет считаться серьезной ошибкой проектирования типа с соответствующим снижением оценки.

[33] Стандарт C++11 вводит для этих целей новое ключевое слово `override`, которое рекомендуется использовать вместе с `virtual` в определении перекрываемых методов.

[34] пример:
`int getA() const { return _a;}`

4.8 Другие требование к программе

1. Явное лучше неявного: в сложных выражениях с неочевидными на 100 % приоритетами операций всегда явно указывать их скобками[35].
2. Использование `#define` для определения констант запрещено.
3. Множественное определение объектов в одной инструкции следует по возможности избегать. Исключения — для простых POD-типов; недопустимо — для указателей.
4. Использование ключевого слова `auto` для типов вида `int`, `double`, `int*` недопустимо. Допустимая область применения — для сложновыводимых объектов с короткой жизнью (например, итераторы, трэиты и т.д.)[36].
5. Необходимо стараться избегать копирования «больших» объектов при использовании их в выражении. Так, в коде на листинге 10 список будет раза 3 пересоздаваться с копированием всех элементов. Альтернативные более удачные решения: а) возвращать указатель на создаваемые в куче объекты (с последующим уничтожением); б) передавать сложные объекты через параметр-ссылку; использовать семантику перемещения (`std::move()` и т.д.), если связанные объекты поддерживают ее.
6. Если нет явных показаний к использованию постфиксной формы записи оператора (++, – и т.д.), предпочтение следует отдавать префиксной форме[37].
7. Если некоторый метод принимает (сложный) объект по ссылке и не изменяет его, ссылка обязательно должна быть константной, в противном случае искусственно и бесполезно сужается область применения метода (см листинг 11).
8. Использование C-style приведение типов[38] объявлено нежелательным. Вместо этого следует использовать один из операторов `xxx_cast<>()`.

[35] Примером нечитаемых приоритетов является инструкция вида
`++i++;`

[36] C++ — строготипизированный язык, который во многих случаях может породить кучу проблем при недостаточном внимании к типам. Злоупотребление ключевым словом `auto` делает код плохопонимаемым. **Правило: «код должен быть удобно читаемым, а не удобно пишущимся»**.

[37] В первом случае создается ненужный временный объект, который никак не используется. В случае применения постфиксного инкремента к, например, итератору, на каждой итерации будет создаваться потенциально большой ненужный объект-итератор.

[38] Вида `(тип)объект`.

Листинг 10. Пример нерационального многократного копирования «большого» объекта, с использованием версии библиотеки stl, не поддерживающей семантику перемещения.

```
1  list<int> foo(...)
2  {
3      list<int> l;
4      l.push_back(...)
5
6      return l;
7  }
8
9  void bar()
10 {
11     list<int> k = foo(...);
12 }
```

Листинг 11. Пример невозможности передачи константы в случае некорректного определения типа параметра в виде неконстантной ссылки.

```
1  void foo(std::string& s) {..};                    //
```

Неконстантная ссылка допустима только к случае, если объект через нее будет изменен.

```
2  void bar(const std::string& s) {..};
3
4  ...
5
6  foo(”Abc”);         // ошибка
7  bar(”Abc”);         // OK
```

5 *Дополнительная информация по семинарам*

Данный раздел включает дополнительную информацию по конкретным семинарам.

6 *Список изменений и дополнений*[39]

- Редакция 1.3 от 11.01.2017 г.[40]
  1. Новые правила на 2017/2018 уч. г.
  2. Добавлен новый раздел 2.1.1.
- Редакция 1.2 от 13.01.2017 г.[41]
  1. Новые правила на 2016/2017 уч. г.
- Редакция 1.1.3 от 23.01.2016 г.[42]
  1. Исправлен месяц в шапке документа.
  2. Уточнение в разделе 4.1 по поводу неоходимости окружения пробелами операторов (листинг 2).
  3. В разделе 4.2 добавлены правила именования объектов.
  4. В разделе 4.5 добавилось требование к управлению памятью с помощью операторов `new` и `delete`.
- Редакция 1.1.2 от 19.01.2016 г.[43]
  1. Уточнение п. 5 раздела 4.7.
- Редакция 1.1.1 от 19.01.2016 г.[44]
  1. Добавления к п. 5 раздела 4.7.
- Редакция 1.1 от 18.01.2016 г.[45]
  1. Поясняющие уточнения в секции 4.4.
  2. Добавилась информация о распределении учебных ассистентов по группам.

[39] Раздел включает информацию по изменениям и дополнениям к документу по каждой его редакции.

[40] Вступает в действие с 14.01.2017 г.

[41] Вступает в действие с 16.01.2017 г.

[42] Вступает в действие немедленно после опубликования.

[43] Вступает в действие немедленно после опубликования.

[44] Вступает в действие немедленно после опубликования.

[45] Вступает в действие немедленно после опубликования.