# Приложение А
# Стиль кодирования

Для того чтобы считаться хорошим программистом или хотя бы просто человеком, умеющим программировать, недостаточно научиться писать программы, «хоть как-то» (кое-как) решающие поставленную задачу. Необходимо писать их *качественно*, а это понятие традиционно включает в себя выполнение некоторых требований, как к алгоритму, так и к тексту программы. Следование определённым рекомендациям — аккуратное написание кода, выбор проверенных архитектурных и алгоритмических решений — позволяет уменьшить количество ошибок в программах и упростить их поддержку. В конце концов, с коммерческой точки зрения это удешевляет производство.

Здесь приводится ряд подобных инструкций, которые следует соблюдать при написании лабораторных работ к курсу «Алгоритмические языки». В данном случае требования носят не рекомендательный, а обязательный характер. Стилей кодирования (т.н. «косметики») множество, каждый вариант имеет свои достоинства и недостатки. Здесь не приводится аргументация в пользу выбранных решений, а насильно вводится один из возможных вариантов, чтобы приучить к самому понятию стиля и упростить преподавателям работу по проверке программ.

Эти требования не являются требованиями на всю жизнь, они ограничены настоящим курсом. В дальнейшем, в том числе и в профессиональной деятельности, придется столкнуться с другими вариантами промышленных стандартов кодирования (по-английски — coding standard, coding style, programming style. . . ), возможно сильно отличающимися или даже противоречащими приведенному здесь. Необходимо будет научиться легко переключаться между различными стандартами.

## А.1. Выравнивание

**Пункт 1.1** В одной строке может располагаться только одна команда.

```
// так нельзя:
a = 5; b = sin(a);

// нужно так:
a = 5;
b = sin(a);
```

**Пункт 1.2** Запрещено объединять две команды в одну с помощью операции «запятая» за исключением случаев, когда иначе не поступить.

```
// так можно (иначе не поступить):
for (i = 0, j = 10; i < j; i++, j--)
  ...

// так нельзя:
if (c == 1)
  a = 5, b = sin(a);

// нужно так:
if (c == 1) {
  a = 5;
  b = sin(a);
}
```

**Пункт 1.3** Тела составных команд (`while`, `do-while`, `for`, `if`, `switch`) и функций не должны располагаться на той же строке, что и сама составная команда.

```
// так нельзя:
if (x < 0) x = -x;

// нужно так:
if (x < 0)
  x = -x;

// так нельзя:
int even(int x) { return x % 2 == 0; }

// нужно так:
int even(int x) {
```

```
  return x % 2 == 0;
}
```

**Пункт 1.4** Первые буквы всех команд одного уровня вложенности находятся в одной колонке.

```
// так нельзя:
 float g;
       g = sin(x);
return g;

// нужно так:
float g;
g = sin(x);
return g;
```

**Пункт 1.5** Тела составных команд и функций выделяются отступом ровно в 2 пробела по отношению к родительской команде или заголовку функции. Использование символов табуляции для отступов запрещено, необходимо корректно установить настройки редактора или среды разработки.

```
// так нельзя:
void f(int* x)
{
for (i = 1; i < 5; i++)
for (j = 1; j < 5; j++)
*(x++) = i - j;
}

// нужно так:
void f(int* x) {
  for (i = 1; i < 5; i++)
    for (j = 1; j < 5; j++)
      *(x++) = i - j;
}
```

**Пункт 1.6** Длинные строки (>100 символов) должны разбиваться в логически обоснованных местах на несколько строк, при этом продолжения выделяются отступом в 4 пробела вместо двух.

**Пункт 1.7** Метки `case` выравниваются с дополнительным отступом в команде `switch`.

```
// нужно так:
switch (a) {
  case 1:
    b = c * 2;
    break;
  case 2:
    b = c * c + 3;
    break;
}
```

## А.2. Фигурные скобки

**Пункт 2.1** Открывающаяся фигурная скобка, которая выделяет тело функции или составной команды, располагается в конце соответствующей строки, а закрывающая — в одной колонке с началом заголовка функции или команды.

```
// так нельзя:
int square(int x)
{
  return x * x;
}

// нужно так:
int square(int x) {
  return x * x;
}

// так нельзя:
while (n > 1)
{res *= n--;}

// нужно так:
while (n > 1) {
  res *= n--;
}
```

**Пункт 2.2** Фигурные скобки, выделяющие тело команды `do-while`, располагаются аналогично. Слово `while` помещается на одной строке с закрывающей фигурной скобкой, отделяясь от нее одним пробелом.

```
// так нельзя:
do
{
  ...
}
while (i > 0);

// нужно так:
do {
  ...
} while (i > 0);
```

**Пункт 2.3** Фигурные скобки внутри `case` не используются, если только нет необходимости в локальных для блока переменных, но и таких случаев лучше избегать.

```
// так можно в случае необходимости:
switch (a) {
  case 1:
    b = c * 2;
    break;
  case 2: {
    int a = c + 1;
    b = a * a + 3;
    break;
  }
}
```

## A.3. Пробелы

**Пункт 3.1** Знаки унарных операций прижимаются к своему единственному аргументу (пробел отсутствует): `-5`, `n++`, `&count`.

**Пункт 3.2** Знаки бинарных операций отделяются от обоих своих аргументов одним пробелом (слева и справа): `a + b`, `count = 3`, `x >= 0`.

**Пункт 3.3** Круглые и квадратные скобки в выражениях прижимаются и к внутреннему содержимому, и к внешнему аргументу: `sin(x)`, `a[5]`.

**Пункт 3.4** Круглые скобки в командах `if`, `while`, `for`, `switch` и т.д. отделяются от самих операторов одним пробелом, прижимаясь к содержимому. Круглые скобки в команде `return` не ставятся: `if (a < 0)`, `switch (bitCount)`, `return a + b`.

**Пункт 3.5** Знаки пунктуации `,` и `;` (запятая и точка с запятой) в списках параметров, в описаниях переменных, в заголовке цикла `for` прижимаются влево, а от правой части отделяются одним пробелом: `float a, b, c;` и `for (i = 0; i < 5; i++)`.

**Пункт 3.6** Операция `,` (запятая) как и знаки пунктуации прижимается к левому аргументу и отделяется от правого одним пробелом: `i = 5, j = 0`.

**Пункт 3.7** Операции `.` и `->` (доступ к членам структур и объединений) не выделяются пробелами, прижимаясь к обоим аргументам: `pos.x`, `block->address`.

**Пункт 3.8** Оба знака операции `?:` отделяются от своих трех аргументов пробелами с обеих сторон: `minimum = a < b ? a : b`.

## А.4. Имена

**Пункт 4.1** Идентификаторы, используемые в программах для именования переменных, функций, типов, макроопределений (`#define`) и других элементов языка должны составляться из слов английского языка, возможно с сокращениями.

**Пункт 4.2** Идентификаторы должны быть понятными, отражающими сущность именуемого объекта. Имена `kkk`, `z3`, `m` в общем случае таковыми не являются и только затрудняют чтение кода. Использование `i`, `j`, `k` в качестве переменной цикла `for` может быть оправданным, но в 99% случаев можно и нужно придумать более говорящее имя даже им.

**Пункт 4.3** Локальные переменные именуются по следующему правилу: имя составляется из маленьких букв без использования знаков подчеркивания, каждое слово, начиная со второго, пишется с большой буквы. Примеры: `message` («сообщение»), `wordCount` («счетчик слов»), `found` («найден»),

**Пункт 4.4** Функции именуются по следующему правилу: имя составляется из маленьких букв без использования знаков подчеркивания, каждое слово пишется с большой буквы: `AppendString()` («присоединить строку»), `VectorLength()` («длина вектора»).

**Пункт 4.5** В библиотечных модулях и больших программах из нескольких файлов локальные для файла функции определяются с ключевым словом `static`, начинаются с подчёркивания и маленькой буквы. Слова, начиная со второго, по-прежнему пишутся с большой буквы: `_doSomeJob()`.

**Пункт 4.6** Глобальные для всей программы переменные предваряются префиксом `g_`. Пример: `g_fatalMessage`.

**Пункт 4.7** Локальные для файла, но глобальные по отношению к функциям данного модуля (т.е., `static`) переменные предваряются префиксом `s_`. Пример: `s_wordCount`.

**Пункт 4.8** Типы именуются маленькими буквами, отдельные слова разделяются знаками подчеркивания, добавляется окончание `_t`. К типам относятся идентификаторы, определенные с использованием ключевого слова `typedef` через базовые простые или сложные типы — структуры (`struct`), объединения (`union`) и перечисления (`enum`). Теги структур, объединений и перечислений строятся по тому же принципу, благо, различие пространств имен позволяет. Примеры имен: `vector_t` («вектор»), `memory_block_t` («блок памяти»), `block_state_t` («состояние блока»).

```
// нужно так:
typedef struct list_element_t {
  int id;
```

```
  block_type_t type;
  struct list_element_t* next;
} list_element_t;
```

**Пункт 4.9** Имена значений перечисления записываются большими буквами, отдельные слова разделяются знаками подчеркивания. Имена должны начинаться со слова, описывающего принадлежность данной константы к конкретному перечислению (содержать имя перечисления или его сокращение).

```
// нужно так:
typedef enum {
  BLOCKSTATE_FREE = 0,
  BLOCKSTATE_OCCUPIED = 1,
  BLOCKSTATE_ILLEGAL = -1,
} block_state_t;
```

**Пункт 4.10** Макроопределения именуются большими буквами, отдельные слова разделяются знаками подчеркивания. Примеры имен: `MAX_WORD_LENGTH` («максимальная длина слова»), `CHECK_VALUE(x)` («проверить значение»),

## А.5. Комментарии

**Пункт 5.1** Для пояснения того, что делает код, необходимо писать Комментарии — для других людей, которые будут читать вашу программу, для себя самого, чтобы через пару лет (или недель), вернувшись к написанному, можно было понять как это работает и что делает.

**Пункт 5.2** Комментарии пишутся на понятном английском (в очень крайнем случае — русском) языке, с использованием обычных предложений (а не конструкций языка программирования или псевдокода).

**Пункт 5.3** В программах на Си для удобства разрешены однострочные Си++ комментарии, начинающиеся с `//`.

**Пункт 5.4** В хорошо документированном коде должна встречаться одна строчка Комментария на каждые 5–10 строк кода (обычно это — логически связанный блок кода, выполняющий одну задачу). Также коментируются все функции, типы и глобальные (`extern`, `static`) переменные.

**Пункт 5.5** Комментарий такого вида должен объяснять не столько **как** что-то делается, сколько **что** (очень краткая сводка блока кода), и, в первую очередь, **для чего** и **почему так** делается (если решение может показаться кому-либо неочевидным или даже неверным).

## A.6. Оформление файлов

**Пункт 6.1** Программы на Си пишутся в файлах с расширением **.c** (не .cpp), заголовочные файлы имеют расширение **.h**.

**Пункт 6.2** Заголовочные файлы содержат только описания макроопределений, типов, прототипы глобальных функций, жизненно необходимых для использования этого модуля, и **не содержат** исполняемого кода.

**Пункт 6.3** Каждый h-файл должен обрамляться директивами препроцессора, исключающими его повторное включение. Идентификатор, используемый в этих директивах, необходимо образовывать из имени файла, записывая его большими буквами, заменяя точку на знак подчеркивания и добавляя слово INCLUDED и два знака подчеркивания в конце: `MYVECTOR_H_INCLUDED__` для файла `myvector.h`. Для компиляторов, её поддерживающих, добавляется директива `#pragma once`:

```
#ifndef MYVECTOR_H_INCLUDED__
#define MYVECTOR_H_INCLUDED__
#pragma once

...

#endif // MYVECTOR_H_INCLUDED__
```

**Пункт 6.4** Каждая глобальная функция (вызывающаяся из другого модуля) должна иметь прототип, описанный в h-файле с тем же именем, что и c-файл, содержащий тело функции. Этот c-файл обязан содержать директиву `#include`, включающую h-файл с прототипами описанных функций.

**Пункт 6.5** Описания и определения обычно должны идти в следующем порядке: подключение стандартных и собственных h-файлов, макроопределения, описания типов, определения глобальных переменных, объявления (прототипы) статических функций при необходимости, а затем определения функций.