Стиль форматирования программ

8 ноября 2026 г.

Следование этому *стилю оформления* программ *обязательно* для всех сдаваемых программ.

1 Набор символов

1.1. В тексте программы запрещено использование любых управляющих символов, кроме символов «Перевода строки» LF (код 10) и «Возврата каретки» CR (код 13). К управляющим относятся символы с кодами от 0 до 31 и символ с кодом 127. 1.2. Этот запрет включает в себя запрет на использование символа табуляции (код 9). Все отступы в программе должны быть сделаны только с использованием символа пробела.

2 Отступы и ширина текста

2.1. Размер отступа - 4 символа. Прочие отступы не допускаются (кроме исключения в п. 2.3 и 4.5.3).

2.2. Ширина текста программы (то есть количество символов в одной строке кода) не должна превышать 100 символов.

2.3. В случаях, когда длинная конструкция программы размещается на нескольких строках, так как не помещается в одну, для продолжения длинной строки на следующих строках используется отступ 8 символов относительно отступа первой части длинной строки. Например:

```
if (a > b &&
        c < d) {
    call_function();
}
```

3 Имена

3.1. Имена глобальных переменных и функций должны быть осмысленными и отражать назначение. Слишком длинные имена, однако, не рекомендуются. В именах допускаются сокращения, которые не затрудняют понимание имени и назначения функций и переменных.

3.2. В именах должны использоваться только английские слова. Транслитерации русских слов не допускаются. Например, имя функции f является недопустимым. Имена функций `calculate_square_root` и `calc_sqrt` допустимы, но последнее предпочтительнее, так как короче и использует общепринятые сокращения. Имя функции `vychislenie_kornya` недопустимо, так как использует русские слова.

3.3. Имена констант записываются полностью заглавными буквами. Если имя константы состоит из нескольких слов, для их разделения используются подчеркивания. Например, `RED`, `POLL_INTERVAL`.

3.4. Имена переменных, функций и меток записываются полностью строчными буквами. Если имя состоит из нескольких слов, для их разделения используются символы подчеркивания. Например, `buf`(допустимое сокращение от слова buffer), `create_tree`. В случаях, оговоренных преподавателями, допускаются отступления от правила именования функций.

3.5. В именах не допускается использование каких-либо префиксов или суффиксов, отражающих тип объекта (т. н. венгерская нотация), например, `a_crszkvc30LastNameCol`.

3.6. Имена типов и тегов начинаются с заглавной буквы. Каждое новое слово в имени начинается с заглавной буквы. Например, `Node`, `BinaryOperation`.

3.7. Имена не должны вызывать визуальные затруднения при прочтении. Например, переменная с именем `l` или константа с именем `I` недопустимы, так как могут быть спутаны с числом 1. То же правило применимо к константе `O`.

4 Стиль отступов

4.1. Структурные типы
Определения структурных типов оформляются следующим образом:

```
struct StructureType {
    int field1;
    int field2;
    char *field3;
};
```

или следующим образом:

```
typedef struct Node {
    struct Node *next;
    double data;
} Node;
```

Каждое определение поля структуры записывается на отдельной строке. Не допускается объединять определение переменной и структуры, например, следующая запись недопустима:

```
struct Foo {
    int fudge;
} foos[10];
```

(C++) Определения классов записываются следующим образом:

```
class Complex {
    double re, im;
public:
    Complex(double re_, double im_) : re(re_), im(im_) {}
};
```

Не допускается именование полей классов ни с префиксом `m`, ни с суффиксом `_`,таким образом неправильно:

```
class Complex1 {
    double re_, im_;
};
class Complex2 {
    double m_re, m_im;
};
```

4.2. Перечислимые типы
Определения перечислимых типов оформляются следующим образом:

```
enum Colours {
    RED,
    GREEN
};
```

или следующим образом:

```
typedef enum Colours {
    RED,
    GREEN
} Colours;
```

Допускается использование анонимных перечислений:

```
enum {
    READY_BIT = 0x1,
    ERROR_BIT = 0x2
};
```

Если определяется единственная константа, допускается запись анонимного перечисление в одной строке:

```
enum { BUFFER_SIZE = 1024 };
```

4.3. Указатели

При объявлении переменных указательного типа знак * относится к имени переменной, а не к имени типа. Пробел между * и именем переменной не ставится.
Правильно: `int *p; char *str, **pstr;`
Неправильно: `int* p; int* q, r; char * * * s;`

4.4. Инициализация

4.4.1. Инициализация составных типов оформляется так, чтобы была понятна структура инициализационных данных.
Например: `int matrix[3][2] = { { 4, 5 }, { 10, 2 }, { 1, 1 } };`
Не допускается пропускать фигурные скобки вложенного инициализатора, даже если они и не обязательны. Например, `int matrix[3][2] = { 4, 5, 10, 2, 1, 1 };`

4.4.2. Инициализация простых и указательных типов записывается на той же строке, что и определение, например: `double sum = 0.0;`

4.4.3. Инициализация символьных строк записывается, как правило, на той же строке, что и определение переменной.

```
char errmsg[] = "No error";
```

Если строка достаточно длинная, допускается разбивать инициализацию на несколько строк текста.

```
char *helpmsg = "hello: my first program\n"
        " usage: hello world\n";
```

4.4.4. Для инициализации локальных структур нулями используйте функцию memset. Даже если некоторые поля структуры инициализируются ненулевыми значениями, предварительно вся структура должна быть обнулена.

4.4.5. Если локальная переменная инициализируется сложным выражением (то есть не константой), каждое такое определение переменной должно записываться на отдельной строке. Неправильно:

```
int var1 = some_func(params), var2 = some_other_func(params);
```

Правильно:

```
int var1 = some_func(params);
int var2 = some_other_func(params);
```

4.5. Функции

4.5.1. Определение функции записывается следующим образом:

```
char *
copy_str(const char *instr) {
    return NULL;
}
```

То есть тип возвращаемого значения, квалификаторы типа и классы памяти записываются на отдельной строке. Имя функции начинается с первой позиции, между именем функции и открывающей скобкой пробелы не ставятся. Закрывающая фигурная скобка блока функции располагается на отдельной строке на первой позиции. Тело функции имеет отступ 4 пробела, оператор `return` обязателен к использованию вне зависимости от типа возвращаемого значения.

4.5.2. Если функция не принимает параметров, в списке параметров обязательно указание ключевого слова `void`:

```
int
random(void) {
}
```

4.5.3. Если у функции становится много параметров, определение функции записывается так, чтобы каждый параметр располагался на отдельной строке с отступом 8 символов. Открывающая фигурная скобка тела функции должна размещаться на отдельной строке. (Много параметров это когда длина строки, содержащей имя функции, начинает превышать 70 символов.)

4.6. Операторы

4.6.1. Оператор `if` без `else` записывается следующим образом.

```
if (test) {
    statements...
}
```

`if` и открывающая скобка, а также закрывающая скобка и открывающая фигурная скобка отделяются одним пробелом. Закрывающая фигурная скобка находится на отдельной строке.

4.6.2. Оператор `if` с `else` записывается следующим образом.

```
if (test) {
    statements...
} else {
    statements...
}
```

4.6.3. Цепочка операторов `if`, `else if`, `else` записывается следующим образом.

```
if (test1) {
    statements...
} else if (test2) {
    statements...
} else {
    statements...
}
```

4.6.4. Оператор `while` записывается следующим образом:

```
while (test) {
    statements...
}
```

4.6.5. Оператор `do-while` записывается следующим образом:

```
do {
    statements...
} while (test);
```

4.6.6. Оператор `for` записывается следующим образом:

```
for (init; test; update) {
    statements...
}
```

4.6.7. Оператор `switch` записывается следующим образом:

```
switch (expr) {
    case LABEL1:
        statements...
        break;
    case LABEL2:
        statements...
        break;
    default:
        statements...
        break;
}
```

4.6.8. В операторах `if`, `while`, `do-while`, `for` тело цикла заключается в фигурные скобки (составной оператор), даже если тело единственный оператор.

4.6.9. Операторы вычисления выражения размещаются каждый на отдельной строке. Пример:

```
a = b;
b = c;
```

4.6.10. Метки записываются с первой позиции вне зависимости от текущего отступа блока. Каждая метка записывается на отдельной строке.

```
void func(void) {
    for (...) {
        for (...) {}
exit1:  ;
    }
}
```

4.6.11. Использование оператора goto настоятельно не рекомендуется. Допускается использование оператора goto при обработке ошибок и при выходе из вложенных циклов. Допускаются только переходы вперед по тексту программы.

4.6.12. (C++) Оператор try-catch записывается следующим образом:

```
try {
    statements...
} catch (ex1 e1) {
    statements...
} catch (ex2 e2) {
    statements
}
```

Типы обрабатываемых исключений должны быть упорядочены от более специализированных к более общим. Объекты исключений пользовательских типов должны передаваться по константной ссылке.

4.7. Операции

4.7.1. Бинарные операции +, * и пр. отделяются пробелами от аргументов, например, `a + b`, `a != 0`, `d >= a`

4.7.2. Операции доступа к полю . и -> бинарной операцией не считаются, поэтому пробелами не окружаются. Скобки [,] пробелами не окружаются. Примеры: `var.field`, `var->field`, `var->do_method()`, `arr[i][j]`

4.7.3. После запятой во всех случаях и после точки с запятой в заголовке цикла for ставится один символ пробела.

```
int a, b, c = 0;
printf("%d\n", n);
for (i = 0; i < 10; ++i)
```

4.7.4. Между унарной операцией и операндом пробел не ставится. Примеры: `b = *a; i++; *p++ = 0;`

4.7.5. Между именем функции и открывающей скобкой пробелы не ставятся.

```
printf("%d\n", value);
while ((c = getchar()) != EOF);
```

4.7.6. После открывающей скобки и перед закрывающей скобкой пробел не ставится.

4.7.7. После закрывающей скобки операции приведения типов ставится один пробел.

```
(unsigned char *) str
```

4.7.8. Конструкции `*(a + i)`, `(*p).f` запрещены. Используйте `a[i]` и `p->f` соответственно.

4.7.9. В выражении не должны присутствовать заведомо лишние скобки. Неправильно: `if ((a > 0) && (!(b))) { }`
Правильно: `if (a > 0 && !b) { }`

5 Использование элементов языка и библиотек

5.1. В тексте программы не допускается использование явных литералов целого и вещественного типов, не объявленных как константы, кроме следующих случаев: Константы 0 и 1 без ограничений. Константа 2, когда она обозначает файловый дескриптор потока ошибок. Константа -1 как специальное значение, сигнализирующее об ошибке. Следующий фрагмент недопустим, так используется константа 10.

```
for (i = 0; i < 10; ++i) { }
```

5.2. Для определения констант целых типов не разрешается использовать ни директиву `#define`, ни квалификатор `const`. Все целые константы должны определяться с помощью ключевого слова `enum` и иметь осмысленные имена. Неправильно:

```
#define BUFSIZE 1024 // использование #define
const int BOARD_WIDTH = 8; // использование const
enum { TEN = 10 }; // бессмысленное имя
```

5.3. Стандартные заголовочные файлы должны подключаться с помощью директивы `#include <FILE>` Пользовательские заголовочные файлы должны подключаться с помощью директивы `#include "FILE"` Неправильно:

```
#include "stdio.h" // стандартный заголовочный файл
#include <tree.h> // пользовательский заголовочный файл
```

5.4. Все сообщения об ошибках, предупреждения, отладочные сообщения и прочая диагностическая информация, не являющаяся результатом работы программы в предположении корректных входных данных, должна выводиться на стандартный поток ошибок `stderr`.

5.5. Каждое диагностическое сообщение должно заканчиваться символом перехода на новую строку.

5.6. При выводе результата работы на стандартный поток вывода или в текстовый файл вывод должен заканчиваться символом перехода на новую строку. Исключением являются программы, выполняющие посимвольную обработку потока данных. Например, неправильно:

```
int
main(void) {
   int a, b;
    scanf("%d%d", &a, &b);
    printf("%d", a + b);
    return 0;
}
```

Но правильно:

```
int
main(void) {
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d\n", a + b);
    return 0;
}
```

5.7. Работа функции `main` должна заканчиваться выполнением оператора `return` с соответствующим кодом завершения. Код завершения — это число в диапазоне $[0, 128)$.

5.8. Использование функций, не проверяющих переполнение буфера, запрещено. К таким функциям относятся: `gets`, `sprintf`, `strcpy`, `strcat`. Использование спецификатора `%s` функций `scanf` и `fscanf` без указания максимального количества вводимых символов запрещено. Использование функций `strncpy`, `strncat` запрещено.

5.9. Использование функции `malloc` не рекомендуется. Допускается использовать `malloc` при реализации расширяемых массивов, если предполагается выполнять `realloc`. Во всех остальных случаях используйте функцию `calloc`. Запрещается использовать `malloc` при выделении памяти под элементы списков, деревьев и прочих подобных структур.

5.10. Запрещается использовать конструкцию `sizeof(TYPE)`, если ее можно заменить конструкцией `sizeof(VALUE)`. Например, неправильно:

```
struct List *p = calloc(1, sizeof(struct List));
```

Правильно:

```
struct List *p = calloc(1, sizeof(*p));
```

5.11. Даже при операциях с массивами элементов типа `char` необходимо вычислять размер с помощью умножения количества элементов на размер одного элемента. Неправильно:

```
s = realloc(s, sz);
```

Правильно:

```
s = realloc(s, sz * sizeof(s[0]));
```

5.12. В программе не должны присутствовать закомментированные строки кода, которые использовались для отладки.

5.13. Вспомогательные функции общего назначения (такие, как `getline`, функции работы со списками и т. д.) не должны ничего выводить на `stdout` или `stderr`. В случае возникновения ошибки такие функции должны, как правило, возвращать специальное значение.

5.14. Глобальные переменные должны использоваться только при необходимости. Передача параметров предпочтительнее использования глобальных переменных.

5.15. Локальные массивы константного и небольшого (до 64кб) размера должны создаваться на стеке, а не в динамической памяти. Неправильно:

```
{
    char *buf = calloc(PATH_MAX, sizeof(*buf));
    // ...
    free(buf);
}
```

Правильно:

```
{
    char buf[PATH_MAX];
}
```

5.16. Запрещается использовать локальные массивы переменного размера. Неправильно:

```
scanf("%d", &size);
double arr[size];
```

5.17. Переменные, которые хранят дескрипторы ресурсов (файловые дескрипторы, дескрипторы потоков `FILE *`, указатели в динамическую память и пр.) должны определяться в начале функции и инициализироваться либо корректным значением дескриптора ресурса, либо специальным значением, обозначающим, что ресурс не выделен. Все ресурсы должны освобождаться когда становятся ненужными. Неправильно:

```
void
func(void) {
    some_code();
    char *s;
    some_other_code();
    s = calloc(size, sizeof(*s));
    still_some_code();
}
```

Правильно:

```
void
func(void) {
    char *s = NULL;
    some_code();
    some_other_code();
    s = calloc(size, sizeof(*s));
    still_some_code();
    free(s);
}
```

5.18. (только для C) Запрещается использовать явное приведение типа результата, возвращаемого функциями `malloc`, `calloc`, `realloc`. Неправильно:

```
double *ptr = (double*) calloc(init_size, sizeof(*ptr));
```

Правильно:

```
double *ptr = calloc(init_size, sizeof(*ptr));
```

6 Многомодульные программы

6.1. Каждая независимо компилируемая единица программы (модуль) должна состоять из двух файлов: файла с суффиксом `.h` (заголовочного файла), содержащего определения типов данных, констант и прототипов функций, и файла с суффиксом `.c` (файла реализации), содержащего тела функций. Головной модуль (исходный файл, содержащий функцию `main`) может не иметь соответствующего `.h` файла.

6.2. Заголовочный файл должен быть защищен от повторного включения. Например, если файл называется `module.h`, то защита от повторного включения может реализовываться следующим образом:

```
#ifndef MODULE_H_INCLUDED
#define MODULE_H_INCLUDED
// текст файла
#endif
```

6.3. Заголовочный файл модуля должен обязательно включаться в файл реализации этого модуля. Другими словами, в файле `module.c` должна быть обязательно директива `#include "module.h"`

6.4. Заголовочный файл модуля должен обязательно включаться во все файлы, которые используют сущности, определенные в этом заголовочном файле.

6.5. Заголовочный файл должен содержать минимальное количество директив `#include` — только те, в которых определяются сущности, используемые в этом заголовочном файле. Если сущности используются в соответствующем файле реализации, но не в данном заголовочном файле, то и включаться требуемый заголовочный файл должен в файле реализации.

6.6. Для сборки проекта в каталоге проекта должен присутствовать `Makefile`, содержащий правила `all` для сборки всего проекта и `clean` для очистки всех файлов, возникающих при компиляции (объектных и исполняемых файлов).