Правила стиля написания программ на C

Контекст

Правила, который перечислены здесь, не являются абсолютными. Они появились из типичного практического контекста, в котором мы пишем код. Хороший код должно быть:

- Легко переиспользовать. Это очень экономит время на отладку программ, не нужно тестировать новые функции и меньше шанс сделать ошибку, добавляя новую логику.

  Примеры нарушений:

  - глобальные переменные;
  - функции, которые делают слишком много разнообразной работы;
  - функции, которые одновременно и кодируют логику, и используют ввод-вывод.
- Легко модифицировать. Ошибки частые гости в программах, сложно модифицировать значит сложно исправлять ошибки.

  Примеры нарушений:

  - глобальные переменные;
  - дублирование кода (модифицировать надо все дубликаты, как их найти?)
- Легко читать. Код постоянно читают ваши коллеги (и вы сами).

  Примеры нарушений:

  - смесь `camelCase` и `snake_case`.
  - большие функции
  - магические константы. Читаете код и видите число 428, что оно означает в контексте? Лучше использовать именную константу.
- Легко тестировать.

  Примеры нарушений:

  - большие функции
  - глобальные переменные
  - функции, которые возвращают свой результат через указатель, который получают в параметрах.

```
void f(int* return_value ) { *return_value = 42; }
```

Иногда эти правила *не применимы*. Например, код для микроконтроллеров часто компилируется проприетарными компиляторами низкого качества, которые не умеют оптимизировать код должным образом. Поэтому приходится делать код "менее красивым" чтобы, например, удовлетворять требованиям производительности, или ужать размер программы, чтобы она поместилась в крохотную память.

Но обычно эти правила дают разумный критерий оценки того, хорошее ли вы приняли архитектурное решение или нет. Если вы их нарушаете, вы должны это осознавать и понимать, чего вы этим хотите добиться.

Правила

Мы подготовили для вас список эмпирических правил, которые помогут вам повысить качество кода. При выполнении заданий по C нужно их обязательно соблюдать.

1. Структура программ

1. Функции должны получать все необходимые им данные через аргументы.
2. Не используйте изменяемые глобальные переменные (константные можно).
3. Нельзя смешивать логику вычислений и ввод-вывод.
4. Нельзя использовать `typedef` для определения структур (объяснение), кроме структур из одного поля, которые являются аналогом `typedef`, но без неявных преобразований (объяснение).
5. В заданиях указан только минимально необходимый набор функций. Вы можете добавлять любое число вспомогательных функций для удобства, это поощряется.
6. Проверьте архитектуру. **Решение внутри одного файла приниматься не будет**.
7. Пишите маленькие функции, каждая из которых делает *что-то одно*.
8. Про каждую функцию задайте себе вопрос: что она делает? Если ответ длиннее нескольких слов, возможно, функцию надо разбить на функции поменьше.
9. Именование.

- Выбирайте хорошие имена для функций и переменных, максимально краткие но информативные (это непросто, хорошие имена приходят в голову не сразу).
- Всегда именуйте функции и переменные единообразно.

11. Делайте маленькие функции даже для тех кусочков кода, которые вы не планируете переиспользовать. Маленькая функция имеет *имя*, которое быстро даёт понять, что она делает.
12. К каждой функции и переменной доступ должен быть в наименьшей возможной части программы. Как следствие:

- Функции и глобальные переменные, которые предназначены только для использования в одном модуле, должны быть помечены `static`
- Не нужно создавать переменные-индексы вне циклов (это было необходимо в стандарте C89).
- Полезно использовать opaque types.

13. Структура файлов:

- В заголовочных файлах нужен Include guard.
- Любой заголовочный файл `header.h` должен быть независим от остальных. Это значит, что файл, состоящий из одной строчки:

```
#include "header.h"
```

должен компилироваться.

- Примерная структура заголовочного файла:
    1. Includes
    2. Макросы.
    3. Определения типов.
    4. Глобальные переменные.
    5. Функции.
- Примерная структура файла `.c`:
    1. Includes
    2. Макросы (которые не должны использоваться в других файлах).
    3. Определения локальных для файла типов.
    4. Глобальные переменные.
    5. Статические глобальные переменные.
    6. Функции.
    7. Статические функции.
- В каком порядке включать заголовочные файлы для модуля `file.c`:
    1. `file.h` (соответствующий ему заголовочный файл).
    2. Файлы из стандартной библиотеки
    3. Другие заголовочные файлы.

2. Типы

1. Всё, что может быть помечено `const`, должно быть помечено. Исключение можно делать для аргументов функций, которые не являются указателями.
2. Для индексов используйте тип `size_t`.

3. Используйте только платформо-независимые типы, такие, как `int64_t` или `int_fast64_t`. Численные типы с пометкой `fast` предпочтительны, т.к. их существование гарантируется на всех платформах.
4. Используйте правильные спецификаторы ввода и вывода.
5. Не используйте спецификаторы `PRI...` для ввода вместо `SCN...`, и наоборот.

3. Использование ввода-вывода

1. Сообщения об ошибках должны выводиться в `stderr`. Общее правило: результаты вычислений — в `stdout`, информация о том, *как* происходят вычисления — в `stderr`.
2. Нельзя смешивать ввод-вывод и логику. Одна функция считает, другая — выводит.

4. Компиляция

1. Мы пишем код для стандарта C17 (или C18, что почти то же самое).
2. Код должен компилироваться с флагами `-std=c18 -pedantic -Wall -Werror` (gcc) или `-std=c17 -pedantic -Wall -Werror` (clang).
3. Пользователям MS Visual Studio придётся тяжко, поддержка C11/C17 пока есть только в Visual Studio 2019 version 16.8 Preview 3. Установите флаги `/W4` (warning level 4) и `/WX` (warnings as errors).

Можете попробовать использовать `cl-clang`.

4. Не забудьте написать `Makefile`. Он должен позволять при изменении одного `.c` файла пересобрать часть проекта не пересобирая всё остальное.

Проверять Ваш код мы будем с помощью `gcc` и `Makefile`. Разрешается `cmake`.

5. Отправка решения

Пожалуйста, присылайте решение в виде pull-request. Инструкция.