

# Основы программирования

Харитонцев-Беглов Сергей

7 сентября 2021 г.

## Содержание

<b>1. Тестирование. Методы мышления</b>	<b>1</b>
1.1 Как думать над задачей. . . . .	1
1.2 Пример задачи. cd и pwd . . . . .	1
1.3 Как меньше лажать внутри кода . . . . .	2
1.4 Исправление багов . . . . .	2

# 1. Тестирование. Методы мышления

## 1.1. Как думать над задачей.

1. Неформальная задача. Например: написать ВКонтакте. Это это уже общение с заказчиком / маркетинг.
2. Формальное условия задачи (на контестах/ДЗ обычно это первое). Также у вас есть примеры, но вы должны их уметь придумывать сами.
3. Идея решения. Как понять, что она есть: умеете рисовать как код работает на примерах.
4. Декомпозиция кода (контракт — соглашение между кусочками). То есть разбиение кода на более мелкие участки. Зачем это нужно? Это помогает удобно думать про задачу, думать отдельно про переменные/тестирование. Удобно, когда мало связей. Как понять, что декомпозиция верна: для каждого «квадратика» понимать как он работает на примере. *Здесь бы схему сделать*
5. Инвариант (микроархитектура). Условия внутри какого-то кусочка кода, например в каком-то форе **a** должна быть четной.
6. Код.

## 1.2. Пример задачи. `cd` и `pwd`

Рассмотрим пример задачи: хотим написать эмулятор консоли, которая умеет выполнять две команды: `pwd` — вывести текущую папку и `cd` — перейти по пути. Два типа пути: абсолютные — `/...` и `a/b/c....`

Стандартная олимпиадная задача, поэтому есть две стандартные части: ввод и ~~вывод~~. Далее из ввода есть `cur_d`, `init()`, `do_cd(string a)`, `do_pwd()`, а `do_cd(string a)` вызывает `do_cd_a(...)` и `do_cd_r(...)`.

```
1 | init()
2 |     cur_d = ""
3 |
4 | do_cd_a(path):
5 |     cur_d = path
6 | do_cd_r(path):
7 |     while path.startswith("../"):
8 |         path.erase(0, 3)
9 |         last_slash = 0
10 |         for i in 0..cur_d:
11 |             if cur_d[i] == "/"
12 |                 last_slash = i
13 |             cur_d.erase(last_slash, cur_d.len);
14 |         cur_d += path
```

Проблема: код не работает. Проблема с тем, где могут стоять слешы, туда-сюда, в рот наоборот. Ну, бахнем инвариант: путь начинается на `/` и кончается на него же. Еще и код перехода назад работает за  $O(n^2)$ . Двоеточия могут быть в абсолютном пути, может быть просто `cd ..` из `/`. Имя папки заканчивается на точку.

Пусть теперь папка — `vector<string>`. Инвариант: внутри не хранятся `."`, `.."`, `."`, `"/`, `""`. По-хорошему, надо сделать функцию, которая постоянно проверяет корректность вектора. Это удобно при отладке.

Теперь сотрем весь старый код. Потребуется новый кусок кода — `split(string) -> vector<string>`.

```
1 | init()
2 |     cur_d = {}
3 | do_cd_a(path)
4 |     cur_d =
5 | do_cd_r(path)
6 | ...
```

Проблемы лучше искать заранее: исправлять код, который еще не написан проще. [Лучше думать про общее, чем про частное.](#)

### 1.3. Как меньше лажать внутри кода

- Название переменной. Название переменной должно показывать, что в нем лежит. Пример: `tmp` — плохое название переменной. Потому что, да, она может быть временной, но непонятно что в ней лежит и как она будет использоваться.
- Плохое название в контексте. Например, `m` — может быть норм, а вот еще и `mm` — это кринж. Вообще использовать две переменные, названия которых являются префиксом/подстрокой другой — кринжайший кринж. [Хорошая статья на эту тему.](#)

### 1.4. Исправление багов

Баг — отклонение от ожиданий.

Если вы знаете баг, то найдите тест, на котором этот тест не работает, запомните его и исправьте баг, спустившись вверх по лестнице.

Виды тестов:

1. Примеры.
2. Простые/разные/общие. Программа умеет делать все виды ходов/все виды операций. То есть проверить все ветки.
3. Минимальные/максимальные/граничные. Минимальные/максимальные по вводу/ответу/количеству действий. Короче минимизируем/максимизируем/ставим в граничные точки все возможные переменные во всех комбинациях. Есть исследование, которое говорит, что так находится сильно больше ошибок.
4. Красивые тесты: строка Туи-Морса, симметричный граф, полный граф, строка из буквы `"a"`.
5. Случайные тесты.
6. Регрессионные тесты. Тесты на баги, которые вы уже нашли (см. выше).
7. Все возможные тест (если возможно).
8. Как бы регрессионные тесты. Тест, если бы в строке `x` был баг.

Но если не запускать решение на тестах, то это бесполезно. Есть несколько методики тестирования:

- Мульти-тест. Это когда решение на вход принимает несколько тестов и выполняет их тест за тестом.
- Unit-тестирование. Это когда тестируются кусочки отдельно, что удобно, когда программа не работает целиком, то вы хотя бы уверены в том, что этот кусок работает.
- Stress-тестирование. Это когда решение запускается на множестве случайных тестов и сравнивается с медленным решением/чекером.