C++ code style guide

Содержание

1 Основные правила

Прежде чем писать какой-либо код, обязательно прочитайте какой-нибудь c++ style guide (наберите эту строку в вашем любимом поисковике и читайте первую попавшуюся ссылку). Например, этот

`http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`

Общее правило: ваша задача на этом курсе — написать наиболее простой, понятный, читаемый и гибкий код среди тех, которые проходят ограничения по времени и по памяти. То есть в первую очередь асимптотическая сложность должна быть правильной, а потом сразу же думайте, как все сделать максимально просто.

Боритесь с дублированием кода: это самое большое возможное зло. Если в процессе написания вам понадобилось копировать и вставить кусок своего кода в этот же код, то это первый признак того, что происходит дублирование. Постарайтесь детектировать идентичные и похожие места, вынесите общую часть в отдельную функцию или класс и воспользуйтесь ей дважды с разными аргументами.

Старайтесь писать аккуратно. Удаляйте лишний, неиспользуемый, закомментированный код, удаляйте переменные и функции, которые вам на самом деле не нужны, остальные называйте понятно.

Придерживайтесь одного определенного стиля. Если работаете с какой-то библиотекой (например, STL), делаете что-то аналогично, старайтесь и свой стиль приводить под ее каноны.

1.1 Синтаксис

На код должно быть приятно смотреть, его должно быть легко читать. Вы его пишете один раз, сохраняете, после чего его читают много раз, поэтому выгодно потратить при написании немного времени на приведение кода в порядок, чтобы

вполследствии сократить своё и чужое время на чтение.
Простые правила ниже служат для улучшения визуального восприятия.

1. Вокруг всех бинарных операторов (`=`, `==`, `+`, `-`, `*`, `/`, `>`, `<<` и др.) должны быть пробелы с обеих сторон.
   Исключением являются операторы `.`, `->`, `::`.

2. После запятой должен быть пробел.

3. Между закрывающейся круглой скобкой и открывающейся фигурной должен быть пробел.

4. Не жадничайте с пустыми строками. Вставляйте всегда пустые строки между определениями глобальных функций, классов, констант, `typedef`'ов, `#include`'ов, между объявлениями методов и функций, между реализациями функций, между объявлениями классов и реализациями функций и т.д.

5. Вставляйте пустые строки в код реализации функций, чтобы подчеркнуть разделение логических частей кода.

6. Не размещайте `if`, `else`, `for`, `while` и др. на одной строке со своим `statement` вот так:

   ```
   if (condition) statement;
   else statement;
   ...
   for (...) statement;
   ```

   Это, во-первых, ухудшает читаемость кода. Вы можете вообще один из statement'ов не заметить или ошибочно решить, что он относится к if'у:

   ```
   if (number % 2 == 0) std::cout << "Even\n"; even = true;
   ```

   А во-вторых, при отладке debugger'ом невозможно понять, выполнив команду "Step Over", выполнилось или не выполнилось условие (или сколько итераций цикла прошло).

   Также рекомендуется всегда обрамлять `if`, `else`, `for`, `while` фигурными скобками вот так:

   ```
   for (int index = 0; index < array.size(); ++index) {
     statement1;
     statement2;
     ...
   }
   ```

даже если внутри только один `statement`.

```
if (number % 2 == 0) {
  std::cout << "Even\n";
}
```

Это более читаемо и безопасно. В варианте без скобок легко ошибиться, например, вот так:

```
if (number % 2 == 0)
  std::cout << "Even\n";
  even = true;
```

Легко подумать, что код `even = true;` — тоже находится под `if`ом.

1.2 Язык C++

Существуют разные языки программирования: C, C++, Java, Python и великое множество других. Между ними есть очевидные внешние сходства и различия: как написать цикл, как определить оператор, как создать класс. Однако основные их отличия кроются в принятых в них методах решения типовых задач и инструментах для этого: если писать цикл по индексу, то какие должны быть его границы? если определить оператор, каков должен быть тип принимаемых аргументов и возвращаемого значения? если создавать класс, какие переменные-члены стоит в нем определять, какие методы, что следует вынести во внешние функции? На все эти вопросы можно дать разные ответы, и все они будут отчасти верными. Есть и общие рекомендации и конструкции, которые зарекомендовали себя за долгое время использования как надежные и удобные, а также примеры, как делать не надо. Некоторые из них описаны в этом разделе.

1. `using namespace std;` — так делать не рекомендуется, включать целый namespace, т.к. из-за этого может возникнуть конфликт имен. Вследствие чего могут возникнуть нетривиальные ошибки компиляции/линковки, а если не повезет, то переменная из namespace может совпасть по названию с какой-то вашей переменной, про которую вы не помните ее область видимости, что приведет к еще более сложнонаходимым багам, хоть все и скомпилируется, но иногда вы будете использовать переменную, думая, что это ваша переменная, и в ней такое-то значение, а значение будет совсем другим. Если нужно использовать много раз `std::vector`, напишите `using std::vector;` если `cout`, то `using std::cout;` и т.д. Кроме того, включая `namespace`, вы сам принцип `namespace`'ов, разделяющих имена, нарушаете.

2. Не используйте массивы фиксированной длины `int[]`, `int*` — используйте вместо них `std::vector<int>`.

3. Не используйте `C-type` строки `char[]` и `char*` — используйте вместо них `std::string`.

4. Не используйте ввод-вывод в стиле C через функции scanf, printf — используйте вместо них операторы `>>` и `<<` у `std::cin` и `std::cout` соответственно. Пример на пункты 1–4:

```
#include <algorithm>
#include <vector>
#include <string>

using std::vector;
using std::string;

void Input(vector<string>* table) {
  int rows, columns;
  std::cin >> rows >> columns;
  for (int row = 0; row < rows; ++row) {
    std::string line;
    std::cin >> line;
    table->push_back(line);
  }
}

void Reverse(vector<string>* table) {
  std::reverse(table->begin(), table->end());
}

void Output(const vector<string>& table) {
  for (int row = 0; row < table.size(); ++row) {
    std::cout << table[i] << std::endl;
  }
}

int main() {
  vector<string> table;
  Input(&table);
  Reverse(&table);
  Output(table);
  return 0;
}
```

Обратите внимание, что по умолчанию для `iostream` включен режим совместимости с `stdio`, который позволяет одновременно использовать оба интерфейса для ввода/вывода. В этом режиме производительность `std::cin` и `std::cout` понижается в несколько раз. Поэтому если размер ввода/вывода имеет порядок от 100 000 чисел, вам надо будет отключить этот режим. Делать это надо до совершения каких-либо операций ввода-вывода, желательно первой же строкой в программе:

```
#include <iostream>

int main() {
    std::ios_base::sync_with_stdio(false);
    ...
    return 0;
}
```

5. Если используется значение типа истина/ложь, то используйте тип `bool`, а не `int`.

6. Не используйте тип `long`. Более стандартный тип — `int`, к нему у всех уже привыкли глаза, и `long` с теми же намерениями — просто смотрится странно. На 32-битных машинах оба типа являются 32-битными и ничем не отличаются, поэтому используйте `int` вместо `long`. Если вам нужен 64-битный тип, придется воспользоваться типом `long long` — отличайте его от просто `long`.

7. При прочих равных, используйте преинкремент `++i`, а не постинкремент `i++`. Это полезная привычка. В случае int'ов это все равно, но если у вас будет в коде сложный итератор, то в процессе постинкремента создается его копия в памяти, что может создать вам неожиданные тормоза и повышенное использование памяти, а догадаться о том, что вся проблема — в коротком выражении `it++` — будет сложно.

8. `main` должен заканчиваться `return 0;`, в противном случае на некоторых компиляторах программа может завершиться с ненулевым кодом возврата, что в свою очередь приводит к Run-time error в тестирующей системе.

9. Вставляйте слово `const` везде, где только это возможно по смыслу. Если какая-то переменная по сути меняться в функции не должна, она должна быть `const`. Если метод класса не меняет при вызове содержимое класса, он должен быть `const`-методом. Таким образом вы обезопасите себя от многих глупых ошибок: они отловятся еще на этапе компиляции.

   Если у вас из-за того, что вы где-то поставили в правильном месте `const`, не компилируется код, то `const` выполнил свою главную задачу. Тогда надо не его убирать, а найти и исправить проблему в другом месте: вы где-то еще

забыли поставить `const` или изменяете переменную, которую не собирались изменять. Надо в этом разобраться, доставить `const` туда, где он еще нужен, а не удалять там, где он вам мешает

10. Используйте везде в программе индексацию с нуля. Если какие-то входные или выходные данные в задаче используют индексацию с единицы, лучше в функции ввода, соответственно вывода, переведите индексацию из одной системы в другую, а везде внутри программы, помимо функций ввода и вывода пользуйтесь индексацией с нуля. Весь язык C++ так спроектирован, что индексация с нуля гораздо удобнее, а как только вы начинаете использовать индексацию с единицы, становится неудобно, появляются вычитания единицы из переменных по всему коду и т.д.

11. Задумывайтесь о переполнениях типов. Если у вас есть две переменные типа `int`, значение каждой равно миллиону, и вы их перемножаете, то тип переполнится (максимальное значение — $2^{31} - 1$), и вы получите неправильный результат. Необходимо перед перемножением привести обе переменные к 64-битному типу `long long`. Если у вас есть две `int` переменные со значением два миллиарда и вы их складываете,— тоже произойдет переполнение, тоже нужно предварительно приводить к `long long`.

12. Не вычитайте никогда просто так ничего из `container.size()`, где `container` — какой-нибудь контейнер из STL, например `vector`. `size()` возвращает `unsigned int`, и если вы будете из него вычитать, то можете легко получить переполнение. Например, если вектор пустой, а вы вычитаете единицу, чтобы узнать последний элемент, или вектор состоит только из одного элемента, а вы вычитаете 2, чтобы узнать предпоследний элемент, и т.д. Всегда приводите результат вызова `size()` к `int`'у, если вам совершенно необходимо вычесть из `size()`. При этом в самом распространенном случае, когда вам нужно написать цикл `for`, проходящий по всем элементам, кроме, скажем, последних десяти, надо просто писать не так

```
// Wrong! If container.size() < 10, you'll get an infinite cycle.
for (int index = 0; index < container.size() - 10; ++index) {
  ...
}
```

В этом цикле, если, к примеру, `container.size() == 5`, то вы получаете реально цикл

```
// Note that 4294967291 > MAX_INT, so the cycle is infinite
for (int index = 0; index < 4294967291; ++index) {
  ...
}
```

А пишите лучше всегда так

```
// Correct: adding to int
for (int index = 0; index + 10 < container.size(); ++index) {
  ...
}
```

ну или хотя бы так

```
// Correct: casted to int
for (int index = 0; index < static_cast<int>(container.size()) - 10; ++index) {
  ...
}
```

Соответственно, если вам нужно вызвать функцию, в которую вы должны передать индекс первого и последнего элемента вектора, то делайте это так:

```
SomeFunction(0, static_cast<int>(container.size()) - 1)
```

По-хорошему, здесь надо бы еще проверять, что в контейнере что-то есть, но к `int`'у приводить надо в любом случае, иначе появляются неочевидные баги.

13. Не пользуйтесь макросами для определения констант. Макросы — это очень опасная и неудобная вещь. Их раскрывает специальный препроцессор, который начинает работать еще до компилятора C++, и он ничего не знает о самом языке. Все конструкции раскрываются буквально. В связи с этим есть множество возможных неочевидных побочных эффектов, а у компилятора нет возможности выполнить проверку типов, константность и т.д. Читайте более подробно об этом в книге Майерса "Effective C++".

    Итак, неправильный вариант:

    ```
    #define MAX_LENGTH 100000  // Wrong! Don't use macros!
    ```

    Правильный вариант:

    ```
    const int MAX_LENGTH = 100000;  // Correct
    ```

14. Входные параметры передавайте в функцию по константной ссылке; по ссылке — чтобы они лишний раз не копировались, по константной — чтобы вы не могли их случайно изменить. **Не забывайте про const!** Выходные параметры передавайте в функции по указателю — чтобы вы могли их изменить; по указателю, а не по ссылке,— чтобы вы могли в месте вызова отличить входные параметры от выходных по амперсанду перед именем переменной. Размещайте входные параметры перед выходными в списке параметров функции или метода.

    Аргументы примитивных типов следует передавать в функции по-другому. Входные параметры типов `int`, `char`, `bool`, `double` передавайте по значению. Они будут копироваться, но это так же почти бесплатно, как и в случае ссылок или указателей. При этом вы не сможете их изменить изнутри функции, что и нужно, т.к. это входные параметры. Если вам нужны эти типы как выходные параметры функции, лучше передавайте их по ссылке, т.к. иначе легко внутри функции перепутать указатель на переменную с самой переменной, и сделать совсем не то, что вы собирались.

    Примеры:

```
void Input(vector<point>* sequence, int& points_to_cover);
void FindMaximumsInSlidingWindow(
    const vector<int>& sequence,
    const string& shifts,
    vector<int>* maximums);
double FindMinimumCoveringCircleRadius(
    const vector<point>& points,
    int points_to_cover);
```

    Примеры вызовов:

```
vector<int> sequence;
int points_to_cover;
Input(&sequence, points_to_cover);
...
...
vector<int> sequence;
string shifts;
Input(&sequence, &shifts);
vector<int> maximums;
FindMaximumsInSlidingWindow(sequence, shifts, &maximums);
...
...
double min_radius = FindMinimumCoveringRadius(points, points_to_cover);
```

Обратите внимание на амперсанды & перед переменными, в которые записывается результат вызова функции.

Если функция возвращает одну величину, пусть она делает это по значению. Это столь же быстро, зато удобнее в месте вызова.
Пример:

```
std::vector<int> readNumbers(std::istream& inputStream = std::cin)
{
    size_t sequenceLength;
    inputStream >> sequenceLength;
    std::vector<int> numbers(sequenceLength);
    for (size_t i = 0; i < numbers.size(); ++i) {
        inputStream >> numbers[i];
    }
    return numbers;
}

int main() {
    std::vector<int> firstSequence = readNumbers();
    std::vector<int> secondSequence = readNumbers();
    ...
}
```

Лишнего копирования в этом месте не возникнет. Дело в том, что эта операция настолько часто встречается, что компиляторы научились ее распознавать и генерировать эффективный код для нее. Технология называется return value optimization, известна также под своей аббревиатурой RVO. Можно и следует по умолчанию считать, что она есть и исправно работает, и писать код так, чтобы им было удобнее пользоваться. Чтобы узнать об этом более подробно, поищите в вашем любимом поисковике ее описание по названию.

Если переданный на вход параметр для выполнения алгоритма необходимо изменять,— это не означает, что параметр автоматически становится выходным параметром. Если целью алгоритма не является менять входной параметр, то изменять этот параметр функция не должна: пользователь алгоритма этого не ожидает, и будет очень не рад такому побочному эффекту. Кроме того, если просто передать параметр по ссылке и поменять его внутри, то пользователь даже не будет догадываться о том, что переданные им данные будут изменены. Появляющиеся вследствие таких побочных эффектов баги очень тяжело искать. Соответственно, в таких ситуациях есть два решения: передавать параметр по значению или передавать как обычно ко константной ссылке, а внутри функции копировать и изменять уже копию. Первый

вариант (передавать по значению) обычно предпочтителен. Т.к. объект передается по значению, его можно менять внутри функции в процессе работы алгоритма (например, сортировать, если это вектор), при этом объект не изменится в месте вызова функции. При копировании аргумента, переданного по константной ссылке, в функции появляется два одинаковых по смыслу объекта, что может привести к путанице и использованию одного из них вместо другого, кроме того, копировать приходится вручную, тогда как при передаче объекта по значению копия делается автоматически, без написания дополнительного кода.
Пример:

```
std::vector<int> unique(std::vector<int> numbers) {
    // here we sort a copy of given numbers,
    // so that the user does not lose his data
    std::sort(numbers.begin(), numbers.end());
    numbers.erase(
        std::unique(numbers.begin(), numbers.end()),
        numbers.end());
    return numbers;
}
```

15. Разделяйте использование `class` и `struct`: классом должна быть любая сущность, которая содержит в себе логику, тогда как структура — это набор данных, объединенных в один объект. В классе все переменные-члены должны быть приватными, для доступа к ним делайте аксессоры, в структуре все переменные должны быть публичными, нетривиальных методов быть не должно.

```
struct Point {
    double x, y;
};

// Compares first by x-coordinate, then by y-coordinate
bool operator < (const Point& first, const Point& second) {
    if (first.x != second.x) {
        return first.x < second.x;
    }
    return first.y < first.y;
}

class Path {
public:
    Path(double time, double averageSpeed)
```

```
        : time_(time), averageSpeed_(averageSpeed)
    {}

    double time() const {
        return time_;
    }

    double averageSpeed() const {
        return averageSpeed_;
    }

    double distance() const {
        return time_ * averageSpeed_;
    }

private:
    double time_;
    double averageSpeed_;
};
```

От структуры точки нам ничего не требуется, поэтому она состоит только из двух публичных полей. Метод `compare` добавлять нельзя, задача сравнения решается определением внешнего оператора $<$. Если нужно, например, запретить изменять координаты (устанавливать их только при создании точки), то ее нужно делать классом с двумя get-аксессорами.
В классе `Path` хранится две величины, а получать требуется три. Если бы `time` и `averageSpeed` были публичными переменными, то доступ к значениям скорости и времени происходил бы как `path.time` и `path.averageSpeed`, а доступ к пройденному расстоянию — как `path.distance()`. Для нахождения расстояния приходится добавлять скобки, то есть всегда приходится помнить о том, что расстояние — это метод, а время и скорость — переменные. Если по какой-то причине (например, недостаточная точность) в будущем хранимые переменные нужно будет поменять и перейти к системе (время, расстояние), то в нашем случае с приватными переменными лишь изменится реализация методов, сохранив интерфейс класса. В случае же с публичными переменными придется изменять интерфейс класса, что немедленно влечет изменение всего кода, который его использует. Хранить все три величины переменными категорически нельзя: если время было равно 1, то действие `path.time = 0.0` нарушит инвариант `time * speed == distance`, что приведет к совершенно непредсказуемым последствиям.

Итак, если вам нужно хранить данные под общим именем, вам подойдет структура; во всех остальных случаях создавайте полноценный класс только с приватными переменные-членами.

16. Старайтесь не использовать по возможности динамическое выделение памяти (с помощью `new` и `malloc`): если вы будете его использовать, вам необходимо будет заботиться и об "уборке мусора", т.е. освобождении памяти. Правильный, безопасный способ это делать — не очень простой, и не входит в материалы курса. Кроме того, вызов `new` довольно медленный, поэтому если очень много раз это сделать, то можете не влезть в Time limit. Если вам интересно, как правильно управлять динамической памятью, читайте книгу Майерса "Effective C++" или наберите в поисковике "RAII".

17. При использовании `vector` имейте в виду, что у него есть удобные методы: различные конструкторы, позволяющие задать размер и значение элемента вектора по умолчанию, операторы присваивания, сравнения (лексикографического) и оператор `swap`. Примеры:

    Создание двумерного вектора размером `rows` × `columns`, заполненного значением 100.

    ```
    vector< vector<int> > cache(
        rows,
        vector<int>(columns, 100));
    ```

    Перестановка двух векторов местами без копирования всего содержимого:

    ```
    vector<int> first(1000000, 1);
    vector<int> second(2000000, 2);
    first.swap(second);
    ```

    Здесь меняются местами реально два внутренних указателя `int*`, что значительно эффективнее, чем копирование векторов целиком, особенно если они большого размера.

18. Имейте в виду, что функция `abs` по стандарту принимает на вход `int` и возвращает `int`, а для взятия модуля вещественного числа (`float`, `double`) необходимо пользоваться функцией `fabs`. При этом в Microsoft Visual Studio сделана перегрузка `abs`, которая работает и для вещественных чисел. Однако на сервере будет `abs(-2.75) != 2.75`.

19. Имейте в виду, что значение `RAND_MAX` — ограничения сверху на значения, выдаваемые функцией `rand()`,— отличаются в разных компиляторах. Тщательно изучайте, каково значение компилятора в вашем компиляторе, а каково — на компиляторе в автоматической системе (компилятор вы выбираете при сдаче задания). Подходит ли вам такое ограничение сверху, или нужно построить на базе функции `rand()` алгоритм, позволяющий возвращать случайные числа, равномерно распределенные в более широком диапазоне, чем `[0, RAND_MAX - 1]`?

20. В большинстве случаев нельзя сравнивать числа типа `float` и `double` просто операторами $<, >, <=, >=, ==$: при вычислениях в вещественных типах накапливается погрешность, вследствие чего равные по сути числа, вычисленные с помощью разной последовательности действий, могут получить различные значения в типах `float` и `double`, и даже $a < b$ может измениться на $b < a$. Погрешность вычислений можно оценить, используя точные знания о том, как именно выполняются арифметические операции, а также как происходят вычисления в используемых вами функциях. Обычно делать этого точно не нужно, т.к. точность типа `double` позволяет хранить 15-16 знаков, а требуемая в задаче точность обычно порядка 10^{-6} или 10^{-9}, но не меньше. Однако для того, чтобы корректно сравнивать числа, следует использовать порог сравнения. Примеры:

```
const double COMPARISON_THRESHOLD = 1e-8;

bool less(double first, double second) {
  return first < second - COMPARISON_THRESHOLD;
}

bool lessOrEqual(double first, double second) {
  return first < second + COMPARISON_THRESHOLD;
}

bool equal(double first, double second) {
  return fabs(first - second) < COMPARISON_THRESHOLD;
}
```

21. Для своих типов (классов, структур), если объекты типа необходимо сравнивать между собой, реализуйте всегда `operator<` и не реализуйте остальные операторы сравнения (`operator<=`, `operator>`, `operator>=`): через `operator<` выражаются все остальные, и общепринятая конвенция — реализовывать только сравнение на “меньше”. В противном случае, дублируется код, а работа различных операторов может оказаться несогласованной. Точно так же, общая конвенция,— что сортировка объектов по умолчанию делается по возрастанию, и в качестве компаратора передается функция сравнения на “меньше”. Это правило необходимо соблюдать, чтобы вашу программу было легко понимать другим программистам.
    Также имейте в виду, что в библиотеке `<utility>` есть готовое решение для автоматического создания остальных операторов, если `operator<` уже реализован. Подробнее см. документацию на
    `http://www.cplusplus.com/reference/std/utility/rel_ops/`

22. Не используйте `std::pair` (за исключением случая, описанного ниже). Причина в том, что в месте использования объекта `pair` невозможно понять, что

кроется за полем `first`, а что — за полем `second`. Это абстрактные названия, которые могут означать что угодно, а в месте использования никаких указаний на это нет. Даже если в месте определения переменной указать, что в ней хранится в `first` и `second`, при чтении придется постоянно возвращаться к месту определения переменной, чтобы разобраться в коде и убедиться, в частности, что `first` и `second` нигде не перепутаны местами — часто встречающаяся ошибка!

Исключением являются небольшие участки кода (помещающиеся на один экран), в рамках которых создается из имеющихся объектов `pair`, далее удобно используется для какой-нибудь операции (например, сортировка), и затем все `pair` обратно "расшифровываются" в новые объекты и более не используются. Это может быть удобно для сортировки по вторичному параметру, т.к. для `pair` есть оператор сравнения по умолчанию, который сравнивает сначала по `first`, затем по `second`. При этом код легко понять, т.к. `pair` определен и используется в одном очень локальном куске кода, который можно охватить взглядом целиком.

2 Организация кода

Как и любая система, код при разрастании становится все более путаным и сложным. Однако есть способы перевести эту сложность преимущественно в его размер, сохраняя логику ясной и прозрачной. Помогает в этом грамотное структурирование: что может быть классом, что должна делать функция, где что должно объявляться. Оно же позволяет удобно осуществить повторное использование нужных участков кода.

1. У каждой переменной должна быть одна-единственная явная цель. Никогда не создавайте переменных `tmp`, выполняющих несколько разных вспомогательных функций во всем коде. Используйте переменную только с одной целью. Переменные, в названии которых используется `tmp` или `temp`, почти всегда либо бессмысленные и ненужные, либо неправильно названы.

2. Объявляйте переменные как можно ближе к месту их первого использования. Старайтесь сразу же инициализировать переменные. Если переменная используется только внутри функции, она должна быть локальной для функции. Если только внутри цикла, она должна быть локальной для цикла. Никогда не делайте глобальных переменных. Локальные переменные блока предпочтительнее по сравнению с локальными переменными функции, локальные переменные функции — по сравнению с переменными-членами класса, а последние — по сравнению с глобальными переменными. Стремитесь сократить "время жизни" каждой переменной: чем меньше время жизни переменных, тем меньше переменных приходится одновременно держать в

голове при чтении и написании кода. Исследования показывают, что человек может эффективно держать в памяти не более 5-7 переменных одновременно. Большее количество неизбежно приводит к ошибкам.

3. Разделяйте программу на ввод, решение и вывод, это делает ваш код более модульным. Способы ввода и вывода часто меняются. У нас используются стандартные потоки и определенный описанный формат, в следующий раз те же данные могут быть записаны в файле или в базе данных в другом формате, затем они же могут поставляться уже в виде переменных в более сложной программе, которая использует ваш алгоритм в качестве подпрограммы. Записывайте вход в отдельные переменные и результат работы — в отдельные. Для их заполнения и вывода напишите отдельные функции. В частности, ваш код становится легче тестируемым, что является важным свойством. Вы можете написать альтернативное решение и сравнить его с вашим, можете запустить стресс-тест.

   Вообще это две принципиально разные области ответственности: ввод-вывод и преобразование данных. Не смешивайте в одном классе или функции несколько разных областей ответственности: один класс отвечает ровно за одну область. Иначе он разрастается, становится слишком сложным, а две разные области ответственности начинают быть слишком сильно связанными. Это плохо, потому что чем более независимы разные части программы, тем меньше поводов для ошибок и тем проще тестировать части программы по отдельности.

4. Никогда не используйте “магические константы” в коде. Если у вас где-то в коде встречаются, например, `'a'` и `'z'`, означающие минимальный и максимальный символ алфавита, то их надо заменить на именованные константы. Например так:

```
const char MIN_LETTER = 'a';
const char MAX_LETTER = 'z';
...

for (char letter = MIN_LETTER; letter <= MAX_LETTER; ++letter) {
    ...
}
...
```

5. Пишите комментарии только по делу. В идеальном случае лучше обходиться вообще без них — ваш код прокомментирует сам себя. Конечно, так редко удается, поэтому комментарии к классам и функциям бывают полезными.

   Не нужно оправдывать плохое имя (см. следующий раздел) подробным комментарием. Если у вас встречается объявление вида

```
int n;  // number of balls in the bucket
```

то нужно заменить его на `int number_of_balls;` или `int numBallsInBucket;` в зависимости от принятого стиля, от того, бывают ли шары не в корзине, и от контекста.

Писать комментарий следует **над** тем, к чему он относится. Комментарии в конце строки значительно удлиняют ее, поэтому ухудшают читаемость. При этом желательно, чтобы строка влезала в 80 символов, а зачастую бывает жесткое ограничение по длине строки (в нашей системе — 100 символов). Если вы все же пользуетесь комментарием в конце строки, то отделяйте его двумя пробелами от кода.

Комментарии к функции должны быть написаны рядом с интерфейсом, а не с реализацией, если они разделены: пользователь будет в первую очередь смотреть на интерфейс, к тому же реализация сторонних библиотек может быть вовсе недоступной. То же самое относится и к классам: комментарии к классу и к его методам должны быть в интерфейсе класса, а не в реализации.

Если вы решили снабдить свой код подробными комментариями, указывайте в них то, что будет интересно читающему. Для класса это описание того, для чего класс нужен, как им пользоваться. Для функции и метода — что они делают, что возвращают, что принимают на вход, какие исключения могут бросать.

Вот пример хорошего комментария к функции.

```
/* Applies per symbol transformation to string.
 * input[i] is transformed into transform[input[i]].
 * If transform map doesn't contain input[i] and defaultSymbol isn't null,
 *   input[i] is transformed to defaultSymbol.
 * If transform map doesn't contain input[i] and defaultSymbol == 0,
 *   function throws TransformError.
 */
void TransformString( const string& input,
    const map<char, char>& transform, const char defaultSymbol,
    string* result );
```

2.1 Имена

У каждой создаваемой сущности в коде есть имя. Сперва автор, а впоследствии и все читающие код ассоциируют имена с сущностями, которые они обозначают. Чтобы в каждый момент точно понимать, что в переменной хранится, чтобы быть уверенным в том, что вызов функция вернет ожидаемое значение, имена нужно давать осмысленные и грамотно определенные.

1. Имена переменных должны быть длинными и понятными. Каждый раз, когда вы пишете одно-двух-буквенное название переменной или используете что-то вроде cur, должно возникать неприятное чувство. Единственное место, где можно позволить себе однобуквенные переменные, — в качестве счетчика в очень коротком for'е без вложенных циклов. И то, у вас должны быть серьезные опасения, когда вы это делаете, вы должны делать это осознанно. Иначе можно легко допустить ошибку с индексами, например перепутать i с j, что происходит постоянно, если называть так переменные. Искать такую ошибку вы будете несколько часов или дней. Даже если в описании задачи у меня есть названия R и L, это не значит, что в программе нужно их так называть. Стиль математического текста очень сильно отличается от стиля кода программы. В математическом тексте есть очень много слов, описывающих формулы и то, что в них происходит. В самих формулах ценится краткость. В коде же наоборот, слов, описывающих происходящее, практически нет. Код должен описывать сам себя, названиями переменных, методов и классов. Поэтому названия должны быть очень прозрачными. Не должно быть нужно возвращаться и смотреть вверх в объявление переменной или смотреть на ее инициализацию, чтобы понять, что она в себе содержит. Никогда не называйте переменные `something1` и `something2`, так как очень легко ошибиться и попасть по соседней клавише, тем самым очень легко сделать баг, а искать его будет тяжело. Используйте `something_first` и `something_second` или что-нибудь еще.

2. Все, что относится к именам переменных, относится и к именам функций, классов и методов. Кроме того, в названиях методов (функций) обязательно должен быть глагол, описывающий действие, которое выполняет метод. Это действие должно быть одно. У каждой функции должна быть одна ясная цель. Если вы понимаете, что не можете придумать название функции без слова And (например `ReadFromFileAndSort`), значит функция выполняет две разные цели, и, скорее всего, ее нужно разбить на несколько меньших функций (`ReadFromFile` и `Sort`), и из внешней вызывать подряд внутренние.

3. Не сокращайте слова в названиях. Это ухудшает читаемость кода, а также делает невозможным поиск по нему. Не нужно сокращать `index` до `ind` или `idx`, `current` — до `cur` и т.д. Единственное исключение — общепринятые сокращения типа `Http` и т.д.

4. Выделяйте названия приватных членов классов, это позволяет отличить их от аргументов методов. Наиболее распространенными способами являются подчеркивание в конце: `name_`, — или префикс `m_`: `m_name`. Начинать имя переменной с подчеркивания не принято; следует помнить о том, что имена, начинающиеся на два подчеркивания или подчеркивание и заглавную букву, зарезервированы стандартом, и использовать их нельзя.

3 Продвинутые замечания

1. Не оптимизируйте преждевременно. Ваш алгоритм должен иметь правильную асимптотическую сложность, чтобы иметь шансы пройти в time limit. Он должен правильно работать, чтобы не получить wrong answer. Это два основных тезиса. Не нужно оптимизировать с целью ускорить программу в константу раз, если это хоть сколько-нибудь усложняет код. Старайтесь сделать свое изначальное решение максимально простым. Оптимизировать нужно только после того, как вы четко замерили время работы программы, убедились, что оно слишком большое, определили, какая именно функция создает узкое место. Даже суперпрофессионалы не берутся заранее предсказывать узкие места системы: в наше время, когда компиляторы умеют делать сумасшедшие оптимизации, это практически невозможно предугадать. Поэтому профессионалы и не пытаются делать это заранее и оптимизировать что-либо заранее. Сначала измерьте, найдите узкое место, а потом уже пытайтесь его оптимизировать. Все вышесказанное относится к выносу переменных из цикла для ускорения, к перебору не до n, а до $\frac{n}{2}$ и т.д. — не нужно ничего из этого сделать. Напишите максимально простое решение, добейтесь правильной его работы, и если вдруг после этого оно окажется слишком медленным — только тогда оптимизируйте. Ваша задача в программах, которые вы пишете на этом курсе,— написать наиболее простой, понятный, читаемый и гибкий код, среди тех, которые проходят в ограничения по времени и памяти. Помните об этом и не оптимизируйте, жертвуя простотой и удобством.

2. Не используйте в программах рандомизацию с помощью `srand(time(0))` и `srand(time(NULL))`. Когда вы параметризуете свою программу текущим временем, вы получаете недетерминированную программу, результат работы которой зависит от времени ее запуска. Это плохо вот чем: может сложиться такая ситуация, что ваша программа не работает только в части случаев, только при определенной генерируемой с помощью `rand()` последовательности чисел, и тогда вы можете 100 раз посылать свою программу в систему проверки и получать там `Wrong Answer`, а при этом тестировать на своей машине и вообще всегда получать правильные ответы. Если у вас упала программа на каком-то тесте, и вы даже знаете, на каком, то вы не можете ее отладить, т.к. баг может никогда не проявиться в тех запусках, что вы делаете у себя на машине, а при этом проявляться всегда на сервере. Может всегда проявляться на обычных запусках программы, а на отладочных запусках — не проявляться, т.к. там другое значение времени в момент запуска `srand`.

   Комментарий. Большая часть вышесказанного верна и в принципе для любой программы, использующей `rand()`, т.к. его реализация в разных компиляторах может быть разной и является разной. Но по крайней мере в разные

запуски на одной и той же машине с одним и тем же `random seed` функция `rand()` будет работать одинаково, а ваша программа с `srand(time(NULL))` — нет, что уж совсем плохо. Поэтому пользоваться `rand()` я разрешаю, чтобы не заставлять вас писать свой генератор псевдослучайных чисел, а вот делать `srand(time(0))` — нет.

Итак, правильная инициализация генератора — это не `srand(time(0))` и не отсутствие инициализации (в этом случае генератор имеет право выдавать не псевдослучайную последовательность, а вообще последовательность нулей, например), а инициализация фиксированной константой. Выберите число, которое вам нравится, например 239, и вызывайте всегда `srand(239)`.

Кроме того, почти всегда нужно вызывать `srand` в программе ровно один раз. Каждый раз, когда вы вызываете `srand`, заново инициализируется генератор, и если вы всегда будете инициализировать его одним и тем же числом, а потом уже вызывать, то вы получите последовательность, состоящую из одного и того же числа — вряд ли это то, что вы хотите. Поэтому вызывайте `srand` один раз прямо в начале функции `main`, если только у вас нет каких-то специальных целей.

3. Не выполняйте никакую сложную работу в конструкторе, также не обращайтесь в конструкторе к каким-либо внешним для программы объектам, таким как файловая система, стандартные потоки ввода и вывода, базы данных и т.д. В конструкторе должна быть только простейшая инициализация полей класса, автономная или в зависимости от параметров конструктора. Это связано с тестированием класса и гибкостью дизайна.

   На самом деле, почти всегда класс, который легко тестировать, имеет гибкий дизайн, и его удобно использовать, и наоборот. Если у класса в конструкторе происходят какие-то сложные действия (обращение к файлу, базе данных или запуск сложного алгоритма), то его сложно протестировать. Чтобы протестировать класс в юнит-тесте, нужно для начала хотя бы создать экземпляр этого класса.

   Если для этого требуется какой-то файл или база данных, то это уже получается сразу не юнит-тест, т.к. ему для работы нужны внешние данные, внешние объекты, что неудобно, в идеале тест должен быть изолирован от остальной системы для чистоты эксперимента. Подменить данные, лежащие в базе данных или в файле, существенно сложнее, чем передать другие значения параметров в какую-то функцию. Для того, чтобы обойти использование файлов и баз данных, придется переделывать класс, в частности убирать у него из конструктора непосредственные обращения к файлам и базе данных. А если это делается не непосредственно в конструкторе этого класса, а в методах других классов, вызываемых в конструкторе, то придется выносить объекты этих классов наружу и подменять их.

Если внутри конструктора сложный алгоритм, то его тоже было бы неплохо протестировать, однако это уже становится невозможно, потому что как только мы захотим создать экземпляр класса, так сразу же вызовем конструктор, и там уже весь алгоритм выполнится, отдельные функции, которые он использует протестировать не получится. Если алгоритм работает неправильно, то к моменту создания объекта класса он будет находиться в некорректном состоянии, и тестировать его будет уже бессмысленно.

Иногда даже иметь в классе указатели на объекты других конкретных классов и создавать их в конструкторе неправильно. Например, если класс, выполняющий какой-то конкретный алгоритм, имеет у себя указатель на объект для работы с базой данных, который в конструкторе инициализируется для обращения к конкретной базе, то такой класс тоже невозможно протестировать по вышеописанным причинам.

На самом деле, нашему алгоритмическому классу нужны от класса, работающего с базой данных, лишь конкретные данные, которые тот берет из базы данных, и скорее всего далеко не все данные, которые есть в базе. Поэтому имеет смысл написать “обертку” вокруг класса, работающего с базой, которая будет обращаться к базе и доставать произвольные данные с помощью внутреннего класса, работающего непосредственно с базой, а наружу отдавать только те куски данных, которые имеют смысл для алгоритмического класса. А для того, чтобы впоследствии можно было работать не только с базой данных, но те же данные брать из файла или откудато из памяти другого объекта, нужно сделать общий интерфейс для классов, поставляющих данные алгоритмическому, и конкретный класс, берущий данные именно из базы, породить от этого интерфейса. Под интерфейсом в данном случае имеется в виду класс с чисто виртуальными методами, который определяет интерфейс всех своих потомков, но инстанцировать который невозможно. Далее, в конструктор алгоритмического класса передавать уже указатель на такой интерфейс, а не указатель на конкретный класс для работы с базой данных, и в конструкторе просто копировать этот указатель во внутреннюю переменную для будущего использования. В таком случае при тестировании можно будет создать mock класса, достающего данные, реализовав этот интерфейс. Наш mock будет “подсовывать” алгоритмическому классу те данные, которые мы хотим, то есть абсолютно любые, что и нужно для полного тестирования. Соответственно, мы сможем проверить реакцию на разные крайние случаи, запустить стресс-тест, понять, какие ограничения на данные должен проверять на входе алгоритмический класс. Нам не придется создавать специальные базы данных для тестирования с подмененными данными, мы сможем генерировать эти данные прямо в памяти, в огромных количествах, сможем выполнить хоть 100 000 тестов, если каждый из них выполняется быстро. С базами данных это не получится, потому что, во-первых, один тест, обращающийся в процессе к базе данных, уже в

любом случае будет занимать существенное время, а во-вторых потому что не получится создать 100 000 различных таблиц.

Более подробное описание, примеры и другие советы для написания хорошо тестируемых классов см. по ссылке
`http://misko.hevery.com/attachments/Guide-Writing%20Testable%20Code.pdf`