Структуры и алгоритмы обработки данных

Оглавление

[Хороший стиль записи программ на C++ (на основе Google Code Style Guide) 2](#_Toc533504262)

[Шаблоны (templates) C++, шаблонные функции и шаблонные классы 5](#_Toc533504263)

[Перегрузка (overloading) функций и операторов в C++ 6](#_Toc533504264)

[Базовые типы данных и строки в С++ 7](#_Toc533504265)

[Класс std::vector<T>, основные свойства и методы. Циклы по элементам вектора 8](#_Toc533504266)

[Реализация матриц на основе std::vector<T>. Создание и обработка 9](#_Toc533504267)

[Стек и основные операции над ним. Реализация на основе std::vector<T> 10](#_Toc533504268)

[Ввод и вывод в C++, классы, операторы, форматирование. Консоль и файлы 11](#_Toc533504269)

[Нотация асимптотического роста для алгоритмов (О-нотация). Классификация алгоритмов по сложности 12](#_Toc533504270)

[Стековый алгоритм проверки сбалансированности скобочных выражений 13](#_Toc533504271)

[Динамическое программирование 14](#_Toc533504272)

[Поиск наибольшей возрастающей последовательности 15](#_Toc533504273)

[Быстрая сортировка 16](#_Toc533504274)

[Замена рекурсии на итерацию на примере быстрой сортировки 17](#_Toc533504275)

[Сортировка слиянием 18](#_Toc533504276)

[Поразрядная сортировка 19](#_Toc533504277)

[Алгоритм Кнута-Морриса-Пратта 20](#_Toc533504278)

[Алгоритм Бойера-Мура 21](#_Toc533504279)

[Поиск с возвратом (backtracking) 22](#_Toc533504280)

[Генерация перестановок 23](#_Toc533504281)

# Хороший стиль записи программ на C++ (на основе Google Code Style Guide)

Главное, что нужно понимать: современный код пишется не для компилятора (он разберётся и скомпилирует что угодно; или откажется компилировать), а для человека. 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 в последствии 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.

Кроме того, в процессе разработки и использования языка C++ родились некоторые полезные правила (которые иногда прямо противоположны тому, что было принято в языке C):

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 повезёт, 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 сложно находимым 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 С через функции scanf, printf используйте вместо них операторы >> и << у std::cin и std::cout соответственно.

5. Если используется значение типа истина/ложь, то используйте тип bool, а не int.6. main должен заканчиваться return 0;

7. 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”.

8. 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 С++ 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.

9. 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. Итак, неправильный вариант:

*#define MAX\_LENGTH 100000 // Wrong! Don’t use macros!*

Правильный вариант:

*const int MAX\_LENGTH = 100000; // Correct*9. 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.

Кроме того, не зависимо от используемого языка полезно придерживаться таких правил:

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. 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.

3. 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.

4. Разделяйте программу на ввод, решение и вывод, это делает ваш код более модульным. Способы ввода и вывода часто меняются. 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.

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.

5. Пишите комментарии только по делу. В идеальном случае лучше обходиться вообще без них - ваш код прокомментирует сам себя. Конечно, так редко удаётся, поэтому комментарии к классам и функциям бывают полезными. Не нужно оправдывать плохое имя подробным комментарием.

# Шаблоны (templates) C++, шаблонные функции и шаблонные классы

Шаблоны (template) – средство языка C++предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).

Шаблоны позволяют создавать параметризованные классы и функции. Параметром может быть любой тип или значение одного из допустимых типов.

Шаблон начинается с ключевого слова template (или по старинке слово class), за которым в угловых скобках следует список параметров. Затем следует объявление функции или класса.

*template <typename T>*

*T min(T a, T b) {*

*return a < b ? a : b;*

*}*

Такая функция может применяться для разных типов входных данных

*min(1, 2);*

*min('a', 'b');*

*min(string("abc"), string("cde"));*

Причём как правило в полной записи *min<double>(3.14159, 2.71828)* имя типа можно опускать, компилятор сам догадается и подставит его. В сложных случаях имя типа указывать можно и нужно.

Подобно функциям, классы тоже могут быть шаблонными, самым ярким примером является класс std::vector<T>, который может хранить вектор любого типа – от самых простых, до сколь угодно сложных (например, вектор векторов).

При использовании класса, как правило, тип-параметр опустить невозможно, потому что компилятор не имеет возможности догадаться.

*std::vector<double> Speed(3);*

Хотя шаблоны предоставляют краткую форму записи участка кода, на самом деле их использование не сокращает исполняемый код, так как для каждого набора параметров компилятор создаёт отдельный экземпляр функции или класса. Как следствие, исчезает возможность совместного использования скомпилированного кода в рамках разделяемых библиотек. Однако, этим недостатком в наше время как правило пренебрегают в пользу того увеличения производительности труда программиста.

Шаблонное метапрограммирование в С++ также страдает от множества ограничений, включая проблемы переносимости, отсутствие поддержки отладки или ввода/вывода в процессе инстанцирования шаблонов, длительное время компиляции, низкую читабельность кода, скудную диагностику ошибок и малопонятные сообщения об ошибках. В некоторых других языках программирования шаблоны реализованы, пожалуй, удачнее.

Однако, именно шаблоны C++ сделали этот вид программирования известным и популярным.

# Перегрузка (overloading) функций и операторов в C++

# Базовые типы данных и строки в С++

# Класс std::vector<T>, основные свойства и методы. Циклы по элементам вектора

# Реализация матриц на основе std::vector<T>. Создание и обработка

# Стек и основные операции над ним. Реализация на основе std::vector<T>

# Ввод и вывод в C++, классы, операторы, форматирование. Консоль и файлы

# Нотация асимптотического роста для алгоритмов (О-нотация). Классификация алгоритмов по сложности

# Стековый алгоритм проверки сбалансированности скобочных выражений

# Динамическое программирование

# Поиск наибольшей возрастающей последовательности

# Быстрая сортировка

# Замена рекурсии на итерацию на примере быстрой сортировки

# Сортировка слиянием

# Поразрядная сортировка

# Алгоритм Кнута-Морриса-Пратта

# Алгоритм Бойера-Мура

# Поиск с возвратом (backtracking)

# Генерация перестановок