2023

Руководство по стилю C++

ГИКБРЕЙНС КОМАНДНАЯ РАЗРАБОТКА КОМАНДА 2

Текстовый редактор

Оглавление

1 Введение

Цель этого документа - обеспечить максимально понятное руководство при разумных ограничениях. Как всегда, здравый смысл никто не отменял. Этой спецификацией мы хотим установить соглашения для всего сообщества разработчиков C++ , не только для отдельных команд или людей. Относитесь со скепсисом к хитрым или необычным конструкциям: отсутствие ограничения не всегда есть разрешение.

Правила построены с учетом руководства по стилю C++ от Google. Дополнительно будут разработаны правила для:

- кроссплатформенной разработки;
- разработки на Qt;
- комментирования для автоматического формирования программной документации с помощью Doxygen.

Изложенные ниже рекомендации рассчитаны на применение стандарта C++17.

2 Заголовочные файлы

2.1 Базовые требования

2.1.1 Заголовочные файлы должны быть самодостаточными и иметь расширение «.h».

2.1.2 Файлы C++ имеют расширения «.cpp».

2.1.3 Шаблоны (template) описываются в заголовочных файлах «.hpp».

2.1.4 Файлы без заголовков, предназначенные для включения, должны заканчиваться на конце «.inc» и использоваться с осторожностью.

2.1.5 Желательно, чтобы каждый «.cpp» файл исходного кода имел парный «.h» заголовочный файл. Исключения: модульные тесты (unit test) или небольшие «.cpp» файлы, содержащие только функцию main().

2.1.6 Каждый класс, как правило, описан в своём файле («*.cpp» и «*.h»), имя файла в общем случае совпадает с именем класса.

2.1.7 Желательно в одном заголовочном файле иметь декларацию только одного класса.

2.2 Кодировка

2.2.1 Сохранять файлы в кодировке Unicode с сигнатурой (UTF-8 with signature).

2.3 Include

2.3.1 При написании путей к включаемым заголовочных файлам следуетиспользовать косую черту `/` (не обратную косую черту `\`!), дабы избежать проблем с компиляцией UNIX-компиляторами.

2.3.2 Для контроля за тем, чтобы конкретный исходный файл при компиляции подключался строго один раз, используется препроцессорная директива «#pragma once». Механизм «#include guard» не используется.

3 Пространство имен (namespace)

3.1 Базовые требования

3.1.1 Размещайте код в пространстве имен.

3.1.2 Не используйте встроенные (inline) пространства имен.

3.1.3 Не объявляйте ничего в пространстве имен std, включая прямые объявления классов стандартной библиотеки. Объявление сущностей в пространстве имен std является неопределенным поведением, т. е. не переносимым. Чтобы объявить сущности из стандартной библиотеки, включите соответствующий файл заголовка.

3.1.4 Не используйте псевдонимы пространств имен (using namespace) в области пространства имен в файлах заголовков, за исключением явно отмеченных пространств имен, предназначенных только для внутреннего использования, поскольку все, что импортировано в пространство имен в файле заголовка, становится частью общедоступного API, экспортируемого этим файлом.

3.2 Локальные переменные

3.2.1 Помещайте переменные функции в максимально узкую область видимости и инициализируйте переменные в объявлении.
Вместо *int i;* , следует писать *int i = 0;*.

```
int i;
i = f (); // Плохо - инициализация отдельно от объявления.
```

```
int j = g (); // Хорошо - объявление инициализировано.
```

```
std :: vector <int> v;
v.push_back (1); // Предпочитать инициализацию с использованием фигурных
скобок.
v.push_back (2);
```

```
std :: vector <int> v = {1, 2}; // Хорошо - v начинает инициализироваться.
```

3.2.2 Рекомендуется при инициализации нового объекта всегда использовать auto (AAA Style, Herb Sutter). В реалиях C++17 единственным исключением является direct member initializers (источник - [habr.com/ru/company/jugru/blog/469465])

3.2.3 C++ позволяет объявлять переменные в любом месте функции. Рекомендуется объявлять их как можно более локально и как можно ближе к первому использованию. Это облегчает читателю поиск объявления.

3.2.4 По возможности используйте переменные с областью видимости условия

```
if (Status status = Foo(); !status.ok())
{
   return status;
}
```

3.3 Статические и глобальные переменные

3.3.1 Использование внешних и статических переменных крайне нежелательно. Практически их использование запрещено. Это может препятствовать реализации многопоточных алгоритмов.

3.3.2 Если вам требуется статическая фиксированная коллекция, например, набор для поиска или таблица поиска, вы не можете использовать динамические контейнеры (maps, sets и др.) из стандартной библиотеки в качестве статической переменной, поскольку они не имеют тривиальные деструкторы. Вместо этого рассмотрите простой массив тривиальных типов, например, массив массивов целых чисел (для «сопоставления от int к int») или массив пар (например, пары int и const char*). Для небольших коллекций вполне достаточно линейного поиска (и эффективного из-за локальности памяти). При необходимости храните коллекцию в отсортированном порядке и используйте алгоритм двоичного поиска. Если вы действительно предпочитаете динамический контейнер из стандартной библиотеки, рассмотрите возможность использования статического указателя, локального для функции.

3.3.3 Если вам требуются статические постоянные данные типа, который вам нужно определить самостоятельно, дайте типу тривиальный деструктор и constexpr конструктор.

3.3.4 Если все остальное не помогает, вы можете создать объект динамически и никогда не удалять его, используя статический указатель или ссылку на локальную функцию (например, static const auto& impl = *new T(args...);).

4 Классы

4.1 Базовые требования

4.1.1 Новые классы должны размещаться в пространстве имен (namespace) модуля, в котором класс реализуется.

4.1.2 Рекомендуется отделять классы API от реализации. То есть в классе API, по возможности, не должно быть приватных методов.

4.1.3 В реализации классов API желательно придерживаться идиом PIMPL, RAII и Priv для написания более надёжного кода (защита от утечек и обращений к пустым объектам).

4.1.4 В классах API рекомендуется вместо прямого использования типов шаблонов STL использовать псевдонимы имен типов с помощью using (создание псевдонимов типов с помощью typedef считается устаревшим и не рекомендуется). В перспективе это позволит изменить реализацию типа.

4.1.5 Если нужно запретить копирующий конструктор и оператор присвоения, то можно сделать так:

```
class ClassName
{
public:
        ClassName (const ClassName &) = delete;
        ClassName & operator=(const ClassName &) = delete;
};
```

4.1.6 Если конструктор или деструктор не имеют явной реализации, то можно делать так:

```
class ClassName
{
public:
        ClassName() = default;
        ~ClassName() = default;
};
```

4.1.7 Каждый класс должен быть предназначен для решения при помощи его методов и членов единственной задачи.

4.1.8 Поля класса можно инициализировать параметрами по умолчанию прямо в месте их декларации. Всегда используйте инициализаторы элементов по умолчанию (direct member initializers, с C++11).

```
class ClassName
{
        int field = 1; // direct member initializers
};
```

4.2 Структуры против классов

4.2.1 Struct и class ключевые слова ведут себя почти одинаково в C++. Мы добавляем собственные семантические значения к каждому ключевому слову, поэтому вы должны использовать соответствующее ключевое слово для определяемого вами типа данных.

4.2.2 Struct должен использоваться для пассивных объектов, которые несут данные, и могут иметь связанные константы. Все поля должны быть общедоступными. Структура не должна иметь инвариантов, которые подразумевают отношения между различными полями, поскольку прямой доступ пользователя к этим полям может нарушить эти инварианты. Могут присутствовать конструкторы, деструкторы и вспомогательные методы; однако эти методы не должны требовать или обеспечивать соблюдение каких-либо инвариантов.

4.2.3 Если требуются дополнительные функции или инварианты, class будет более подходящим вариантом. Если сомневаетесь, делайте class.

4.3 Структуры против пар и кортежей

4.3.1 Предпочитайте использовать struct вместо пары или кортежа, когда элементы могут иметь значимые имена.

4.3.2 Пары и кортежи могут быть подходящими в общем коде, где нет конкретных значений для элементов пары или кортежа. Их использование также может потребоваться для взаимодействия с существующим кодом или API.

4.4 Наследование

4.4.1 Композиция часто более уместна, чем наследование.

4.4.2 Используйте открытое наследование (спецификатор доступа public). Если хотите сделать частное наследование, вы должны вместо этого включить экземпляр базового класса в качестве члена.

4.4.3 Ограничьте использование protected теми функциями-членами, к которым может потребоваться доступ из подклассов. Обратите внимание, что данные-члены должны быть закрытыми.

4.4.4 Явно аннотируйте переопределения виртуальных функций или виртуальных деструкторов точно с одним из override или (реже) final спецификатором. Не используйте virtual при объявлении переопределения. Обоснование: функция или деструктор, отмеченные override или final не являющиеся переопределением виртуальной функции базового класса, не будут компилироваться, и это помогает выявлять распространенные ошибки. Спецификаторы служат документацией; если спецификатор отсутствует, читатель должен проверить всех предков рассматриваемого класса, чтобы определить, являются ли функция или деструктор виртуальными, или нет.

4.4.5 Множественное наследование разрешено при наследовании от интерфейсов. Интерфейс - это класс, который не имеет переменных-членов и все методы которого являются чистыми виртуальными функциями (не забудьте о виртуальных деструкторах!). Множественное наследование реализации категорически не рекомендуется.

4.5 Контроль доступа

4.5.1 Создавайте данные-члены классов private. Это упрощает рассуждения об инвариантах за счет некоторого простого шаблона в виде средств доступа, если это необходимо.

4.6 Порядок декларирования

4.6.1 Определение класса обычно должно начинаться с public раздела, за которым следует protected, затем private. Пропустите разделы, которые будут пустыми.

4.6.2 В каждом разделе лучше группировать похожие типы объявлений вместе и предпочитать следующий порядок: типы (включая using и вложенные структуры и классы), константы, фабричные функции, конструкторы и операторы присваивания, деструктор, все другие методы, члены данных.

4.6.3 Не помещайте в определение класса определения методов, за исключением реализации шаблонов и встроенных методов.

4.7 Методы

4.7.1 Методы чтения (getter) возвращают результат по значению, за исключением случаев предоставления доступа к коллекциям. В таком случае определяется метод, возвращающий константную ссылку на коллекцию, а в также методы добавления и удаления элементов коллекции.

```
const std::list<std::shared_ptr<IAbonentAccountingRecord>>&
GetAbonentAccountingRecords();
void AddAbonentAccountingRecord(const
std::shared_ptr<IAbonentAccountingRecord>);
void RemoveAbonentAccountingRecord(const
std::shared_ptr<IAbonentAccountingRecord>);
```

4.7.2 Методы чтения (getter) объявляются константными. Рекомендуется использовать универсальный атрибут [[nodiscard]] (возвращаемое функцией значение нельзя игнорировать и нужно сохранить в какую-либо переменную).

4.7.3 Модифицирующие методы (setter) в качестве параметров используют константные ссылки (если владение не передаётся) или умные указатели, если владение передаётся.

5 Функции

5.1 Базовые требования

5.1.1 Избегать рекурсивных функций. Как альтернатива – стековая реализация.

5.1.2 Аргументы по умолчанию не используются в виртуальных функциях.

5.1.3 Форма определения функции «Trailing return types» используются только по необходимости.

```
int foo(int x); // используется повсеместно
auto foo(int x) -> int; // обычно не используется
```

5.2 Входные параметры

5.2.1 В списках параметров функции все ссылки должны быть const.

```
void Foo (const std :: string& in, std :: string* out);
```

Однако в некоторых случаях использование const T* предпочтительнее const T& для входных параметров. Например, вы хотите передать нулевой указатель или функция сохраняет указатель или ссылку на ввод.

Помните, что большую часть времени входные параметры будут указаны как const T&. Использование const T* вместо этого сообщает читателю, что ввод каким-то образом обрабатывается по-другому. Поэтому, если вы выбираете, const T*а не const T&, делайте это по конкретной причине (уточните её в комментарии к функции); иначе это, скорее всего, запутает читателей, заставив их искать объяснение, которого не существует.

5.2.2 При необходимости передачи в функцию большого количества параметров (аргументов) задумайтесь об использовании структуры или класса, содержащих требуемые параметры в качестве данных-членов.

5.3 Выходные параметры

5.3.1 Вывод функции C ++, естественно, обеспечивается через возвращаемое значение, а иногда и через выходные параметры.

5.3.2 Предпочтительнее использовать возвращаемые значения, а не выходные параметры: они улучшают читаемость и часто обеспечивают такую же или лучшую производительность. Если используются параметры только для вывода, они должны записываться после параметров ввода.

5.3.3 Все параметры только для ввода помещайте перед любыми параметрами вывода. В частности, не добавляйте новые параметры в конец функции только потому, что они новые; поместите новые параметры только для ввода перед параметрами вывода.

5.4 Написание коротких функций

5.4.1 Предпочитайте небольшие и целенаправленные функции.

5.4.2 Старайтесь избегать функции с большим количеством аргументов.

5.4.3 Иногда уместны длинные функции, поэтому жестких ограничений на длину функций нет. Если функция превышает примерно 40 строк, подумайте, можно ли ее разбить, не повредив структуру программы.

Даже если ваша длинная функция сейчас работает отлично, кто-то, изменив ее через несколько месяцев, может добавить новое поведение. Это может привести к ошибкам, которые трудно найти. Если ваши функции будут короткими и простыми, то другим людям будет легче читать и изменять ваш код. Небольшие функции также легче тестировать.

При работе с некоторым кодом можно встретить длинные и сложные функции. Не пугайтесь изменения существующего кода: если работа с такой функцией оказывается сложной, вы обнаруживаете, что ошибки трудно отлаживать, или вы хотите использовать ее часть в нескольких разных контекстах, рассмотрите возможность разбиения функции на более мелкие и больше управляемых частей (для этого используйте рефакторинг типа «Извлечение метода»).

5.5 Перегрузка функций

5.5.1 Используйте перегруженные функции (включая конструкторы) только в том случае, если читатель, может получить хорошее представление о том, что происходит, без необходимости сначала точно выяснить, какая перегрузка вызывается.
Вы можете написать функцию, которая принимает a, const std::string& и перегрузить ее другой, которая принимает const char*. Однако, вместо этого рассмотрите использование std::string_view.

```
class MyClass
{
       public:
              void Analyze(const std::string& text);
              void Analyze(const char* text, size_t textlen);
};
```

Перегрузка может сделать код более интуитивным, позволяя функции с одинаковым именем принимать разные аргументы. Это может быть необходимо для шаблонного кода, а может быть удобно при использовании шаблона проектирования «Посетитель» (Visitor).

Вы можете перегрузить функцию, если между вариантами нет семантических различий. Эти перегрузки могут различаться по типам, квалификаторам или количеству параметров (аргументов). Однако читатель такого вызова не должен знать, какой член набора перегрузки выбран, только то, что вызывается что-то из набора. Если вы можете задокументировать все записи в наборе перегрузки с помощью одного комментария в заголовке, это хороший признак того, что это хорошо спроектированный набор перегрузки.

5.6 Аргументы по умолчанию

5.6.1 Аргументы по умолчанию разрешены для не виртуальных функций, если по умолчанию всегда гарантировано одно и то же значение. Следуйте тем же ограничениям, что и для перегрузки функций , и отдавайте предпочтение перегруженным функциям, если удобочитаемость, полученная с использованием аргументов по умолчанию, не перевешивает недостатки, указанные ниже.
Часто у вас есть функция, которая использует значения по умолчанию, но иногда вы хотите переопределить значения по умолчанию. Параметры по умолчанию

позволяют легко сделать это, не определяя множество функций для редких исключений. По сравнению с перегрузкой функции аргументы по умолчанию имеют более чистый синтаксис, с меньшим количеством шаблонов и более четким различием между «обязательными» и «необязательными» аргументами.

Аргументы по умолчанию - это еще один способ достичь семантики перегруженных функций, поэтому применимы все причины, по которым не следует перегружать функции .

5.6.2 Значения по умолчанию для аргументов в вызове виртуальной функции определяются статическим типом целевого объекта, и нет гарантии, что все переопределения данной функции декларируют одни и те же значения по умолчанию.
Параметры по умолчанию повторно оцениваются при каждом вызове, что может привести к раздуванию сгенерированного кода. Читатели также могут ожидать, что значение по умолчанию будет фиксироваться при объявлении, а не изменяться при каждом вызове.

Указатели функций сбивают с толку при наличии аргументов по умолчанию, поскольку сигнатура функции часто не соответствует сигнатуре вызова. Добавление перегрузок функций позволяет избежать этих проблем.

5.6.3 Аргументы по умолчанию запрещены для виртуальных функций, где они не работают должным образом, и в тех случаях, когда указанное значение по умолчанию может отличаться от того же значения в зависимости от того, когда оно было вычислено. (Например, не пишите void f (int n = counter++);.)
В некоторых других случаях аргументы по умолчанию могут улучшить читаемость их объявлений функций, чтобы преодолеть указанные выше недостатки, поэтому они разрешены. Если сомневаетесь, используйте перегрузки.

5.7 Встраиваемые (inline) функции

5.7.1 Для повышения эффективности использовать inline реализации мелких функций вы можете объявлять функции встраиваемыми и указать компилятору на возможность включать её напрямую в вызывающий код, помимо стандартного способа с вызовом функции.

5.7.2 Определяйте функции как встраиваемые только когда они маленькие, например, не более 10 строк. Учтите, что на современных процессорах более компактный код выполняется быстрее благодаря лучшему использованию кэша инструкций.

5.7.3 Для повышения читаемости класса реализацию inline функций больше одной строки выносить в конец заголовочного файла.

5.7.4 Нет смысла делать встраиваемыми функции, в которых есть циклы или операции switch (кроме вырожденных случаев, когда цикл или другие операторы никогда не выполняются).

5.7.5 Использование встраиваемых функций может генерировать более эффективный код, особенно когда функции маленькие. Используйте эту возможность для get/set функций, других коротких и критичных для производительности функций.

5.7.6 Избегайте делать встраиваемыми деструкторы, т.к. они неявно могут содержать много дополнительного кода: вызовы деструкторов переменных и базовых классов.

5.7.7 Рекурсивные функции не должны объявляться встраиваемыми.

6 Разное

6.1 Enum

6.1.1 Используйте перечисления с областью видимости (scoped enum, enum class). Более предпочтительным для перечислений является использование объявления enum class, так как это более строгая типизация, которая позволяет защититься от присвоения целых значений, которых нет в перечислении. Также это защита от возможного конфликта имён.

6.1.2 Для ссылки на элементы нумераторов надлежит использовать префикс типа.

6.1.3 Используйте enum class или std::variant для представления внутреннего состояния объектов.

6.1.4 Предпочитайте std::variant, если в разных состояниях класс способен хранить разные поля данных.

6.1.5 Используйте старый enum если вам крайне важна неявная конвертация enum в целое число.

6.1.6 Используйте enum class или enum вместо магических чисел.

6.1.7 Используйте enum class, enum или constexpr вместо макросов-констант.

6.2 Друзья (friend)

6.2.1 Использование ключевого слова friend должно иметь основания. Например, для сокрытия методов, используемых в одном модуле.

6.2.2 Друзья обычно должны быть определены в одном файле, чтобы читателю не приходилось искать в другом файле, как использовать закрытые члены класса.

6.2.3 Друзья расширяют, но не нарушают границу инкапсуляции класса. В некоторых случаях это лучше, чем делать члена общедоступным, когда вы хотите предоставить к нему доступ только одному другому классу. Однако большинство классов должны взаимодействовать с другими классами исключительно через своих публичных членов.

6.3 Приведение типов

6.3.1 Не используйте приведение в стиле C. Вместо этого используйте приведения в стиле C++, когда необходимо явное преобразование типа. Используйте static_cast в качестве эквивалента приведения в стиле C, который выполняет преобразование значений, когда вам нужно явно преобразовать указатель от класса к его суперклассу, в случае, когда вам нужно явно привести указатель от суперкласса к подклассу, используйте dynamic_cast. В последнем случае вы должны всегда делать проверку результата динамического приведения на нулевой указатель.

6.3.2 Используйте прямую инициализацию списком (direct-list-initialization) для преобразования арифметических типов (например, `int64{x}`). Это самый безопасный подход, поскольку код не будет компилироваться, если преобразование может привести к потере информации. Синтаксис также лаконичен.

6.3.3 Используйте const_cast для удаления const квалификатора, если уверены, что объект не константный или объект не будет изменен. Const_cast рекомендуется использовать только в исключительных случаях.

6.3.4 В исключительных случаях используется `reinterpret_cast` для небезопасного преобразования типов указателей в целочисленные и другие типы указателей и обратно. Используйте это, только если вы знаете, что делаете, и понимаете проблемы псевдонимов (aliasing).

6.4 Const

6.4.1 Мы настоятельно рекомендуем использовать const в API (например, в параметрах функций, методах и нелокальных переменных) везде, где это имеет смысл и точность. Это обеспечивает согласованную, в основном проверенную компилятором документацию о том, какие объекты операция может изменять. Наличие последовательного и надежного способа отличать чтение от записи имеет решающее значение для написания потокобезопасного кода, а также полезно во многих других контекстах.

6.4.2 Если функция гарантирует, что она не будет изменять аргумент, переданный по ссылке или по указателю, соответствующий параметр функции должен быть ссылкой на const (`const T&`) или указателем на const (`const T*`) соответственно.

6.4.3 Объявляйте методы `const` если они не изменяют логическое состояние объекта (или не позволяют пользователю изменять это состояние, например, возвращая неконстантную ссылку, но это бывает редко), или они не могут быть безопасно вызваны одновременно.

6.4.4 Использование const для локальных переменных не рекомендуется.

6.4.5 Все const операции класса должны быть безопасными для одновременного вызова друг с другом. Если это невозможно, класс должен быть четко задокументирован как «не потокобезопасный».

6.4.6 Const используется везде, где это имеет смысл (в частности, для всех входных параметров, передаваемых по указателю/ссылке).

6.5 Контейнеры и массивы

6.5.1 Настоятельно рекомендуется придерживаться парадигмы STL итераторов и алгоритмов для работы с ними, как при использовании существующих контейнеров, так и при реализации новых контейнеров

6.5.2 Для массивов фиксированной длины рекомендуется использовать std::array вместо встроенных массивов([]).

6.6 Указатели

6.6.1 Указатели рекомендуется хранить в умных указателях. Например std::shared_ptr, std::unique_ptr. Нужно иметь ввиду, что циклические связи не рекомендуются, так как в этом случае удаление не происходит, что приводит к утечкам памяти. Например, в случае обратных ссылок можно использовать std::weak_ptr. Обратной стороной использования умных указателей являются накладные расходы на их хранение и копирование.

6.7 0 и nullptr / NULL

6.7.1 Для обозначения нулевых указателей используем nullptr.

6.7.2 Никогда не используйте NULL для числовых значений.

6.7.3 Используйте '\0' для нулевого символа. Использование правильного типа делает код более читабельным.

6.8 Лямбда

6.8.1 При необходимости используйте лямбда-выражения с форматированием, как описано ниже.

6.8.2 Предпочитайте явные захваты, если лямбда может выйти за пределы текущей области. Например, вместо:

```
{
  Foo foo;
  ...
  executor->Schedule([&] { Frobnicate(foo); })
  ...
}
// BAD! The fact that the lambda makes use of a reference to `foo` and
// possibly `this` (if `Frobnicate` is a member function) may not be
// apparent on a cursory inspection. If the lambda is invoked after
// the function returns, that would be bad, because both `foo`
// and the enclosing object could have been destroyed.
```

надо писать:

```
{
  Foo foo;
  ...
  executor->Schedule([&foo] { Frobnicate(foo); })
  ...
}
// BETTER - The compile will fail if `Frobnicate` is a member
// function, and it's clearer that `foo` is dangerously captured by
// reference.
```

6.8.3 Используйте захват по умолчанию по ссылке ([&]) только тогда, когда время жизни лямбда-выражения явно короче, чем любые возможные захваты.

6.8.4 Используйте захват по умолчанию по значению ([=]) только как средство привязки нескольких переменных для короткой лямбды, когда набор захваченных переменных очевиден с первого взгляда. Не рекомендуется писать длинные или сложные лямбды с захватом по умолчанию по значению.

6.8.5 Используйте захват только для фактического захвата переменных из охватывающей области. Не используйте захваты с инициализаторами для введения новых имен или существенного изменения значения существующего имени. Вместо этого объявите новую переменную обычным способом и затем зафиксируйте ее, или избегайте сокращения лямбда и явно определите объект функции.

6.9 Sizeof

6.9.1 Предпочитайте sizeof(varname) вместо sizeof(type)

Используется, когда вы берете размер конкретной переменной. Обновится соответствующим образом, если кто-то изменит тип переменной сейчас или позже.

```
struct data;
memset(&data, 0, sizeof(data));
```

```
memset(&data, 0, sizeof(Struct));
```

```
if (raw_size < sizeof(int)) {
  LOG(ERROR) << "compressed record not big enough for count: " << raw_size;
  return false;
}
```

6.10 Goto

6.10.1 Оператор goto не используется.

7 Именования

7.1 Базовые требования

7.1.1 Используйте именования CamelCase. UpperCamelCase для именования файлов и классов.

7.1.2 Стиль CamelCase на аббревиатуры не распространяется.

7.1.3 В названиях классов, методов и переменных используются английские термины по возможности без сокращений.

7.2 Имена типов

7.2.1 Имена всех типов - классов, структур, псевдонимов, перечислений, параметров шаблонов - именуются в UpperCamelCase. Подчёркивания не используются. Например:

```
// classes and structs
class UrlTableTester
{
        ...
}
struct UrlTableProperties
{
        ...
}

// typedefs
typedef hash_map<UrlTableProperties *, std::string> PropertiesMap;

// using aliases
using PropertiesMap = hash_map<UrlTableProperties *, std::string>;

// enums
enum UrlTableErrors
{
        ...
}
```

7.3 Имена переменных

7.3.1 LowerCamelCase для именования переменных.

7.3.2 Члены данных структуры, статические и нестатические, именуются как обычные переменные.

```
struct UrlTableProperties
{
        std::string name;
        int numEntries;
        static Pool<UrlTableProperties>* pool;
};
```

7.4 Имена функций

7.4.1 UpperCamelCase для именования функций.

7.5 Имена членов классов

7.5.1 Не рекомендуется использовать публичные члены класса. В исключительных случаях именование публичных членов соответствует правилам именования переменных.

7.5.2 Private и protected члены класса именуются с добавлением подчёркивания в начале.

```
class TableInfo
{
        ...
        private:
                std::string _tableName;  // OK - подчёркивание в конце
                static Pool<TableInfo>* _pool;  // OK.
};
```

7.6 Имена именованных констант

7.6.1 Именованные константы определяемые директивой define пишутся целиком заглавными буквами. В качестве разделителя используется нижнее подчеркивание.

7.7 Именование пространства имён (namespace)

7.7.1 Пространства имен должны иметь уникальные имена на основе имени проекта и, возможно, пути к нему.

7.7.2 UpperCamelCase для именования пространств имен

7.8 Имена перечислений

7.8.1 Имя самого перечисления - это тип. Следовательно, используется UpperCamelCase.

7.9 Имена макросов

7.9.1 Обычно, макросы *не* должны использоваться. Однако, если они вам абсолютно необходимы, именуйте их прописными буквами с символами подчёркивания.

```
#define ROUND(x) ...
#define PI_ROUNDED 3.0
```

7.10 Имена указателей

7.10.1 При именовании указателей, звёздочка и наименование типа указателя пишутся слитно, затем указывается имя указателя.

8 Комментирование

8.1 Базовые требования

8.1.1 Комментарии к методам, классам и структурам обязательны.

8.1.2 Комментарии пишутся на русском языке.

8.1.3 Типы и их члены комментируются перед их объявлением.

8.1.4 Для комментариев используется могут использоваться команды doxygen XML (https://www.doxygen.nl/manual/xmlcmds.html). Пример:

```
///<summary>Итерировать выбранные объект модели</summary>
///<param name="it">Итератор</param>
///<returns>Выбранный объект модели</returns>
```

8.2 Комментарии в шапке файла

8.2.1 В начало каждого файла вставляйте шапку с лицензией.

8.2.2 Комментарии в файле должны описывать его содержимое. Если файл объявляет, описывает или тестирует одну абстракцию (на которую уже есть комментарий), дополнительное описание в шапке файла не нужно. В ином случае, в начало файла вставляйте описание содержимого.

8.3 Комментарии класса

8.3.1 Каждое объявление класса (кроме совсем очевидных) должно сопровождаться комментарием, для чего класс и как им пользоваться.

8.3.2 Комментарий к классу должен быть достаточным для понимания: как и когда использовать класс, дополнительные требования для правильного использования класса. Описывайте, если требуется, ограничения (предположения) на синхронизацию в классе. Если экземпляр класса может использоваться из разных потоков, обязательно распишите правила многопоточного использования.

8.3.3 В комментарии к классу также можно привести короткие примеры кода, показывающие как проще использовать класс.

8.3.4 Класс объявляется/определяется в разных файлах (.h и .cpp). Комментарии, описывающие использование класса должны быть рядом с определением интерфейса. Комментарии о тонкостях реализации должны быть рядом с кодом самих методов.

8.4 Комментарии функции

8.4.1 Комментарии к объявлению функции должны описывать использование функции (кроме самых очевидных случаев). Комментарии к определению функции описывают реализацию.

8.4.2 Объявление каждой функции должно иметь комментарий (прямо перед объявлением), что функция делает и как ей пользоваться. Комментарий можно опустить, только если функция простая и использование очевидно. Старайтесь начинать комментарии в изъявительном наклонении ("Открывает файл"). Использование повелительного наклонение ("Открыть файл") - не рекомендуется. Комментарий описывает суть функции, а не то, как она это делает.

8.4.3 В комментарии к объявлению функции обратите внимание на следующее:

- Что подаётся на вход функции, что возвращается в результате.

- Для функции-члена класса: сохраняет ли экземпляр ссылки на аргументы, нужно ли освобождать память.
- Выделяет ли функция память, которую должен удалить вызывающий код.
- Могут ли быть аргументы nullptr.
- Сложность (алгоритмическая) функции.
- Допустим ли одновременный вызов из разных потоков. Что с синхронизацией?

Пример:

```
// Возвращает итератор по таблице. Клиент должен удалить
// итератор после использования. Нельзя использовать итератор
// если соответствующий объект GargantuanTable был удалён.
//
// Итератор изначально указывает на начало таблицы.
//
// Этот метод эквивалентен следующему:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
//    return iter;
// Если вы собираетесь сразу же делать новую операцию поиска,
// быстрее будет вызвать NewIterator() и избежать лишней операции поиска.
Iterator* GetIterator() const;
```

Однако не стоит разжёвывать очевидные вещи.

8.4.4 Когда документируйте перегружаемые функции, делайте основной упор на изменениях по сравнению с исходной функцией. А если изменений нет (что бывает часто), то дополнительные комментарии вообще не нужны.

8.4.5 Тривиальные конструкторы и деструкторы не комментируются.

8.4.6 Если есть какие-то хитрости в реализации функции, то можно к определению добавить объяснительный комментарий. В нём можно описать трюки с кодом, дать обзор всех этапов вычислений, объяснить выбор той или иной реализации (особенно если есть лучшие альтернативы). Можете описать принципы синхронизации кусков кода (здесь блокируем, а здесь рыбу заворачиваем).

8.4.7 В определении функции вы *не должны* повторять комментарий из объявления функции (из .h файла или т.п.). Можно кратко описать, что функция делает, однако основной упор должен быть *как* она это делает.

8.5 Комментарии к переменным

8.5.1 Имя переменной должно сразу говорить, что это и зачем. Однако, в некоторых случаях требуются дополнительные комментарии.

8.5.2 Назначение каждого члена класса должно быть очевидно. Если есть неочевидные тонкости (специальные значения, завязки с другими членами, ограничения по времени жизни) — всё это нужно комментировать. Однако, если типа и имени достаточно - комментарии добавлять не нужно.

8.5.3 Должны быть описания особых (и неочевидных) значений (nullptr или -1).

Например:

```
private:
 // Используется для проверки выхода за границы
 // -1 - показывает, что мы не знаем сколько записей в таблице
 int numTotalEntries_;
```

8.5.4 Ко всем глобальным переменным следует писать комментарий о их назначении и (если не очевидно) почему они должны быть глобальными.

```
// Общее количество тестов, прогоняемых в регрессионом тесте
const int kNumTestCases = 6;
```

8.6 Комментарии к реализации

8.6.1 Комментируйте реализацию функции или алгоритма в случае наличия неочевидных, интересных, важных кусков кода.

8.6.2 Блоки кода, отличающиеся сложностью или нестандартностью, должны предваряться комментарием.

```
// Делим результат на 2. Переменная x содержит флаг переноса
for (int i = 0; i < result->size(); ++i)
{
        x = (x << 8) + (*result)[i];
        (*result)[i] = x >> 1;
        x &= 1;
}
```

8.6.3 Строки кода с неочевидным смыслом желательно дополнять комментарием (обычно располагаемым в конце строки). Этот комментарий должен отделяться от кода 2-мя пробелами.

```
// Мапируем блок данных, если объём позволяет
mmapBudget = max<int64>(0, mmapBudget - index_->length());
if (mmapBudget >= dataSize_ && !MmapData(mmap_chunk_bytes, mlock))
{
        return;  // Ошибку уже логировали
}
```

Отметим, что здесь 2 комментария на блок кода: один описывает что код делает, другой напоминает, что ошибка уже в логе, если идёт возврат из функции.

8.7 Комментарии TODO

8.7.1 Используйте комментарии TODO для временного кода или достаточно хорошего (промежуточного, не идеального) решения.

8.7.2 Комментарий должен включать строку TODO (все буквы прописные), за ней имя, адрес e-mail, ID дефекта или другая информация для идентификации разработчика и сущности проблемы, для которой написан TODO. Цель такого описания - возможность потом найти больше деталей. Наличие TODO с описанием не означает, что указанный программист исправит проблему. Поэтому, когда вы создаёте TODO, обычно там указано Ваше имя.

```
// TODO(kl@gmail.com): Используйте "*" для объединения.
// TODO(Zeke) Изменить для связывания.
// TODO(bug 12345): удалить функционал "Последний посетитель".
```

8.7.3 Если ваш TODO вида "В будущем сделаем по-другому", то указывайте либо конкретную дату ("Исправить в ноябре 2005"), либо событие ("Удалить тот код, когда все клиенты будут обрабатывать XML запросы").

8.8 Комментарии к namespace

8.8.1 Завершайте многострочные пространства имен комментариями, как показано в примере.

```
// В .h файле
namespace mynamespace
{
     // Все объявления находятся в области видимости пространства имен.
     class MyClass
     {
             public:
             ...
             void Foo();
     };
}  // конец namespace mynamespace
```

9 Форматирование

9.1 Базовые требования

9.1.1 При определении типов, функций, условных операторов и т.п. составной оператор определяется на новой строке. Открывающая и закрывающая фигурные скобки на новой строке без отступа.

9.1.2 Содержимое составного оператора смещается на один символ табуляции относительно уровня выравнивания предыдущего оператора.

9.1.3 Члены-функции, составляющие конкретный интерфейс (как объявление, так и реализацию), заключаются в директивы препроцессора «#pragma region <название_региона>» и «#pragma endregion <название_региона>».

9.1.4 В одной строке кода может быть только один оператор.

9.1.5 Проверки с указателями и bool пишутся в виде:

```
if (pntr), if (!pntr)
```

9.1.6 Проверки на целочисленные типы всегда пишутся в виде явного сравнения.

```
if (n == 0), if (n != 0), if (n > 0) etc.
```

9.1.7 Операции обрамляются пробелами.

9.1.8 При объявлении ссылок и указателей перед знаками * и & пробел не ставится, а после — ставится:

```
Curve& projectionCurve = pCurve->GetProjection(plane)
```

9.1.9 Два указателя не объявляются в одной строке.

9.1.10 Желательно ограничивать длину строк кода 80-ю символами.

9.1.11 Используйте табуляцию для отступов. 1 Tab на один отступ.

9.2 Include

9.2.1 Директивы «#include» отделяются от директивы «#pragma once» одной строкой.

9.2.2 Директивы «#include» группируются по назначению и источнику.

9.2.3 Директивы «#include» отделяются от исходного кода двумя пустыми строками.

9.3 Объявления и определения функций

9.3.1 Старайтесь размещать тип возвращаемого значения, имя функции и её параметры на одной строке (если всё умещается). Разбейте слишком длинный список параметров на строки также как аргументы в вызове функции.

```
ReturnType ClassName::FunctionName(Type parName1, Type parName2)
{
        DoSomething();
        ...
}
```

В случае если одной строки мало:

```
ReturnType ClassName::ReallyLongFunctionName(Type parName1, Type parName2,
                                             Type parName3)
{
        DoSomething();
        ...
```

```
}
```

9.3.2 Можно опустить имя неиспользуемых параметров, если это очевидно из контекста:

```
class Foo
{
public:
        Foo(const Foo&) = delete;
        Foo& operator=(const Foo&) = delete;
};
```

9.3.3 Неиспользуемый параметры с неочевидным контекстом следует закомментировать в определении функции:

```
class Shape
{
public:
        virtual void Rotate(double radians) = 0;
};

class Circle : public Shape
{
public:
        void Rotate(double radians) override;
};

void Circle::Rotate(double /*radians*/) {}
```

```
// Плохой стиль - если кто-то потом захочет изменить реализацию функции,
// назначение параметра не ясно.
void Circle::Rotate(double) {}
```

9.3.4 Атрибуты и макросы старайтесь использовать в начале объявления или определения функции, до типа возвращаемого значения:

```
ABSL_MUST_USE_RESULT bool IsOk();
```

9.4 Лямбды

9.4.1 Форматируйте параметры и тело выражения аналогично обычной функции, список захватываемых переменных - как обычный список.

9.4.2 Для захвата переменных по ссылке не ставьте пробел между амперсандом (&) и именем переменной.

```
int x = 0;
auto x_plus_n = [&x](int n) -> int { return x + n; }
```

9.4.3 Короткие лямбды можно использовать напрямую как аргумент функции.

```
std::set<int> blacklist = {7, 8, 9};
std::vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
digits.erase(std::remove_if(digits.begin(), digits.end(), [&blacklist](int i)
{
        return blacklist.find(i) != blacklist.end();
}),
digits.end());
```

9.5 Числа с плавающей запятой

9.5.1 Числа с плавающей запятой всегда должны быть с десятичной точкой и числами по обе стороны от неё (даже в случае экспоненциальной нотации). Такой подход улучшить читабельность: все числа с плавающей запятой будут в одинаковом формате, не спутаешь с целым числом, и символы E/e экспоненциальной нотации не примешь за шестнадцатеричные цифры. Помните, что число в экспоненциальной нотации не является целым числом.

```
float f = 1.f;
long double ld = -.5L;
double d = 1248e6;
```

```
float f = 1.0f;
float f2 = 1;   // Также правильно
long double ld = -0.5L;
double d = 1248.0e6;
```

9.6 Вызов функции

9.6.1 Следует либо писать весь вызов функции одной строкой, либо размещать аргументы на новой строке. И отступ может быть либо по первому аргументу, либо 1 Tab. Старайтесь минимизировать количество строк, размещайте по несколько аргументов на каждой строке.

```
bool result = DoSomething(argument1, argument2, argument3);
```

9.6.2 Если аргументы не помещаются в одной строке, то разделяем их на несколько строк и каждая следующая строка выравнивается на первый аргумент. Не добавляйте пробелы между круглыми скобками и аргументами:

```
bool result = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

9.6.3 Допускается размещать аргументы на нескольких строках с отступом в 1 Tab:

```
bool result = DoSomething(
        argument1, argument2,  // Отступ 1 Tab
        argument3, argument4);
```

9.7 Условия

9.7.1 Старайтесь не вставлять пробелы с внутренней стороны скобок. Размещайте if и else на разных строках.

```
if (condition)
{
        ...  // отступ 1 tab
}
else if (...)
{  // 'else' находится на новой строке
        ...
}
else
{
        ...
}
```

9.7.2 Должен быть пробел между if и открывающей скобкой.

```
if(condition) {   // Плохо - нет пробела после 'if'
```

```
if (condition)
{  // Хороший код - правильное количество пробелов перед 'if'
```

9.7.3 Короткие условия можно записать в одну строку, если это улучшит читабельность. Используйте этот вариант только если строка короткая и условие не содержит секцию else.

```
if (x == kFoo) return new Foo();
if (x == kBar) return new Bar();
```

9.7.4 Старайтесь, чтобы количество уровней вложенности условий не превышало 3.

9.8 Циклы и switch-и

9.8.1 Конструкция switch использует скобки для блоков. Описывайте нетривиальные переходы между вариантами. Скобки необязательны для циклов с одним выражением. Пустой цикл должен использовать либо пустое тело в скобках или continue.

9.8.2 Рекомендуется в switch делать секцию default. Это необязательно в случае использования перечисления, да и компилятор может выдать предупреждение если обработаны не все значения. Если секция default не должна выполняться, тогда формируйте это как ошибку. Например:

```
switch (var)
{
        case 0:
        {
                ...      // Отступ 4 пробела
                break;
        }
        case 1:
        {
                ...
                break;
        }
        default:
        {
                ...
        }
}
```

9.9 Указатели и ссылки

9.9.1 Вокруг '.' и '->' не ставьте пробелы. Оператор разыменования или взятия адреса должен быть без пробелов.

9.9.2 Ниже приведены примеры правильного форматирования выражений с указателями и ссылками:

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

9.9.3 При объявлении переменной или аргумента нужно размещать '*' к типу:

```
char* c;
const std::string& str;
```

9.9.4 Допускается объявлять несколько переменных одним выражением. Однако не используйте множественное объявление с указателями или ссылками - это может быть неправильно понято.

```
// Хорошо - читабельно
int x, y;
```

```
int x, *y;  // Плохо - не используйте множественное объявление с & или *
char * c;  // Плохо - пробелы с обеих сторон *
const std::string & str;  // Плохо - пробелы с обеих сторон &
```

9.10 Логические выражения

9.10.1 Если логическое выражение очень длинное (превышает типовое значение), используйте единый подход к разбивке выражения на строки. Для упрощения сложных логических выражений используйте рефакторинг «Введение поясняющей переменной»

Например, здесь при переносе оператор AND располагается в конце строки:

```
if (thisOneThing > thisOtherThing &&
    aThirdThing == aFourthThing &&
    yetAnother && lastOne)
{
        ...
}
```

9.10.2 Отметим, что разбиение кода (согласно примеру) производится так, чтобы && и оператор AND завершали строку. Также, можете добавлять дополнительные скобки для улучшения читабельности. Учтите, что использование операторов в виде пунктуации (такие как && и ~) более предпочтительно, что использование операторов в виде слов and и compl.

9.11 Возвращаемые значения

9.11.1 Не заключайте простые выражения return в скобки.

9.11.2 Используйте скобки в return expr только если бы вы использовали их в выражении вида x = expr;.

```
return result;                  // Простое выражение - нет скобок
// Скобки - Ок. Они улучшают читабельность выражения
return (some_long_condition &&
        another_condition);
```

```
return (value);               // Плохо. Например, вы бы не стали писать var
= (value);
return(result);               // Плохо. return - это не функция!
```

9.12 Директивы препроцессора

9.12.1 Знак '#' (признак директивы препроцессора) должен быть в начале строки.

9.12.2 Даже если директива препроцессора относится к вложенному коду, директивы пишутся с начала строки.

```
// Хорошо - директивы с начала строки
  if (lopsided_score) {
#if DISASTER_PENDING      // Корректно - начинается с начала строки
    DropEverything();
# if NOTIFY               // Пробелы после # - ок, но не обязательно
    NotifyClient();
# endif
#endif
    BackToNormal();
  }
```

```
// Плохо - директивы с отступами
if (lopsidedScore)
{
    #if DISASTER_PENDING  // Неправильно! "#if" должна быть в начале строки
    DropEverything();
    #endif                // Неправильно! Не делайте отступ для "#endif"
    BackToNormal();
  }
```

9.13 Форматирование классов

9.13.1 Имя базового класса пишется в той же строке, что и имя наследуемого класса (конечно, с учётом ограничения в 80 символов).

9.13.2 Ключевые слова public, protected, и private должны быть без отступа. Перед каждым из этих ключевых слов должна быть пустая строка (за исключением первого упоминания). Также в маленьких классах пустые строки можно опустить.

9.13.3 Не добавляйте пустую строку после этих ключевых слов.

Ниже описан базовый формат для класса (за исключением комментариев):

```
class MyClass : public OtherClass
{
public:
        MyClass();
        explicit MyClass(int var);
        ~MyClass() {}

        void SomeFunction();
        void SomeFunctionThatDoesNothing()
        {
        }
        void SetSomeVar(int var) { someVar_ = var; }
        int SomeVar() const { return someVar_; }

private:
        bool SomeInternalFunction();
        int someVar_;
        int someOtherVar_;
};
```

9.14 Форматирование пространств имён

9.14.1 Содержимое в пространстве имён пишется с отступом.

```
namespace
{
```

```
  void foo()
  {
     ...
  }

}  // namespace
```

9.14.2 При объявлении вложенных пространств имён используйте '::'.

```
namespace foo::bar
{
}
```

9.15 Вертикальная разбивка

9.15.1 Сведите к минимуму вертикальное разбиение.

Это больше принцип, нежели правило: не добавляйте пустых строк без особой надобности. В частности, ставьте не больше 1-2 пустых строк между функциями, не начинайте функцию с пустой строки, не заканчивайте функцию пустой строкой, и старайтесь поменьше использовать пустые строки. Пустая строка в блоке кода должна работать как параграф в романе: визуально разделять две идеи.

Базовый принцип: чем больше кода поместится на одном экране, тем легче его понять и отследить последовательность выполнения. Используйте пустую строку исключительно с целью визуально разделить эту последовательность.

9.15.2 Пустая строка в начале или в конце функции не улучшит читабельность.

9.15.3 Пустые строки в цепочке блоков if-else могут улучшить читабельность.

9.15.4 Пустая строка перед строкой с комментарием обычно помогает читабельности кода - новый комментарий обычно предполагает завершение старой мысли и начало новой идеи. И пустая строка явно на это намекает.