Руководство Google по стилю в C++

Содержание

	Лямбды Метапрограммирование на шаблонах Концепты и Ограничения Boost Ещё возможности C++ Нестандартные расширения Псевдонимы/Alias Оператор Switch
Инклюзивный Язык	
Именование	Общие принципы именования Имена файлов Имена типов Имена Концептов Имена переменных Имена констант Имена функций Именование пространства имён (namespace) Имена перечислений Имена макросов Исключения из правил именования
Комментарии	Стиль комментариев Комментарии в шапке файла Комментарии классов или структур Комментарии функции Комментарии к переменным Комментарии к реализации Пунктуация, орфография и грамматика Комментарии TODO
Форматирование	Длина строк Не-ASCII символы Пробелы против Табуляции Объявления и определения функций Лямбды Числа с плавающей запятой Вызов функции Форматирование списка инициализации Операторы цикла и ветвления Указатели и ссылки Логические выражения Возвращаемые значения Инициализация переменных и массивов Директивы препроцессора Форматирование классов Списки инициализации конструктора Форматирование пространств имён Горизонтальная разбивка Вертикальная разбивка
Исключения из правил	Существующий код, не соответствующий стилю Программирование под Windows
Перевод	

Вступление

C++ один из основных языков программирования, используемый в open-source проектах Google. Известно, что C++ очень мощный язык. Вместе с тем это сложный язык и, при неправильном использовании, может быть рассадником багов, затруднить чтение и поддержку кода.

Цель руководства - управлять сложностью кода, описывая в деталях как стоит, и как не стоит, писать код на C++. Правила этого руководства упростят управление кодом и увеличат продуктивность кодеров.

Style / Стиль - соглашения, которым следует C++ код. Стиль - это больше, чем форматирование файла с кодом.

Большинство open-source проектов, разрабатываемых Google, соответствуют этому руководству.

Примечание: это руководство не является учебником по C++: предполагается, что вы знакомы с языком.

Цели Руководства по стилю

Зачем нужен этот документ?

Есть несколько основных целей этого документа, внутренних **Зачем**, лежащих в основе отдельных правил. Используя эти цели можно избежать длинных дискуссий: почему правила такие и зачем им следовать. Если вы понимаете цели каждого правила, то вам легче с ними согласиться или отвергнуть, оценить альтернативы при изменении правил под себя.

Цели руководства следующие::

Правила должны стоить изменений
: Преимущества от использования единого стиля должны перевешивать недовольство инженеров по запоминанию и использованию правил. Преимущество оценивается по сравнению с кодовой базой без применения правил, поэтому если ваши люди всё равно не будут применять правила, то выгода будет очень небольшой. Этот принцип объясняет почему некоторые правила отсутствуют: например, `goto` нарушает многие принципы, однако он практически не используется, поэтому Руководство это не описывает.

Оптимизировано для чтения, не для написания
: Наша кодовая база (и большинство отдельных компонентов из неё) будет использоваться продолжительное время. Поэтому, на чтение этого кода будет тратиться существенно больше времени, чем на написание. Мы явно заботимся чтобы нашим инженерам было легко читать, поддерживать, отлаживать код. "Оставляй отладочный/логирующий код" - одно из следствий: когда кусок кода работает "странно" (например, при передаче владения указателем), наличие текстовых подсказок может быть очень полезным (`std::unique_ptr` явно показывает передачу владения).

Пиши код, похожий на существующий
: Использование единого стиля на кодовой базе позволяет переключиться на другие, более важные, вопросы. Также, единый стиль способствует автоматизации. И, конечно, автоформат кода (или выравнивание `#include`-ов) работает правильно, если он соответствует требованиям утилиты. В нестандартных случаях можно порекомендовать выбрать одно (наиболее подходящее) правило и ему следовать (и меньше спорить по этому поводу). С другой стороны, следование единому стилю тоже должно быть разумным. Использование текущего (единого) стиля хорошо работает для изолированных интерфейсов или отдельных файлов, когда нет проблем технического характера или смены парадигмы. Однако, следование единому стилю не должно препятствовать внедрению нового стиля, если в нём есть явные преимущества или код переводится в новый стиль.

Пиши код, похожий на используемый в C++ сообщества (по возможности)
: Согласованность нашего кода с C++ кодом других организаций и сообществ весьма полезна. Если возможности стандартного C++ или принятые идиомы языка облегчают написание программ, это повод использовать их. Однако, иногда стандарт и идиомы плохо подходят для задачи. В этих случаях (как описано ниже) имеет смысл ограничить или запретить использование некоторых стандартных возможностей. В некоторых случаях создаётся свой решение, но иногда используются внешние библиотеки (вместо стандартной библиотеки C++) и переписывание её под свой стандарт слишком затратно.

Избегайте неожиданных или опасных конструкций
: В языке C++ есть неочевидные и даже опасные подходы. Некоторые стили кодирования ограничивают их использование, т.к. их использование несёт большие риски для правильности кода.

Избегайте конструкций, которые средний C++ программист считает заумными и сложно поддерживаемыми

В C++ есть возможности, которые в целом не приветствуются по причине усложнения кода. Однако, в часто используемом коде применение хитрых конструкций более оправданно благодаря многократному использованию, также новые порции кода станут более понятны. В случае сомнений - проконсультируйтесь с руководством проекта. Это очень важно для нашей кодовой базы, т.к. владельцы кода и команда поддержки меняются со временем: даже если сейчас все понимают код, через несколько лет всё может измениться.

Учитывайте масштаб кода

С кодовой базой более 100 миллионов строк и тысячами инженеров, ошибки и упрощения могут дорого обойтись. Например, важно избегать замусоривания глобального пространства имён: коллизии имён очень сложно избежать в большой базе кода если всё объявляется в глобальном пространстве имён.

Оптимизируйте по необходимости

Оптимизация производительности иногда важнее, чем следование правилам в кодировании.

Намерение 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 Google в C++, не только для отдельных команд или людей. Относитесь со скепсисом к хитрым или необычным конструкциям: отсутствие ограничения не всегда есть разрешение. И, если не можешь решить сам, спроси начальника.

Версия C++

Сейчас код должен соответствовать C++20. Возможности языка, относящиеся к C++23, использовать не следует. В дальнейшем руководство будет корректироваться на более новые версии C++.

Не используйте нестандартные расширения.

Прежде чем использовать возможности C++17 и C++20 в проектах, оцените возможность портирования кода для другого окружения.

Заголовочные файлы

Желательно, чтобы каждый `.cc` файл исходного кода имел парный `.h` заголовочный файл. Также есть известные исключения из этого правила, такие как юниттесты или небольшие `.cc` файлы, содержащие только функцию `main()`.

Правильное использование заголовочных файлов может оказать огромное влияние на читабельность, размер и производительность вашего кода.

Следующие правила позволят избежать частых проблем с заголовочными файлами.

Независимые заголовочные файлы

Заголовочные файлы должны быть самодостаточными (в плане компиляции) и иметь расширение `.h`. Другие файлы (не заголовочные), предназначенные для включения в код, должны быть с расширением `.inc` и использоваться в паре с включающим кодом.

Все заголовочные файлы должны быть самодостаточными. Пользователи и инструменты разработки не должны зависеть от специальных зависимостей при использовании заголовочного файла. Заголовочный файл должен иметь блокировку от повторного включения и включать все необходимые файлы.

Когда в заголовочном файле объявляются встраиваемые функции или шаблоны (которые будут инстанцироваться внешним кодом), они должны целиком определяться в заголовочных файлах: либо в том же самом, либо во включаемых файлах. Не выделяйте определения в отдельные заголовочные файлы (`-inl.h`). Раньше такая практика была очень популярна, сейчас это нежелательно. В случае, если все инстанцирования шаблона производятся в одном .cc файле (либо они явные, либо шаблон используется только в этом файле), то определение шаблона может храниться в этом .cc файле.

Возможны редкие ситуации, когда заголовочный файл не самодостаточный. Это может происходить, когда файл подключается в нестандартном месте, например в середине другого файла. В этом случае может отсутствовать блокировка от повторного включения, и дополнительные заголовочные файлы также могут не подключаться. Именуйте такие файлы расширением `.inc`. Используйте их парой и старайтесь чтобы они максимально соответствовали общим требованиям.

Блокировка от повторного включения

Все заголовочные файлы должны быть с защитой от повторного включения посредством `#define`. Формат макроопределения должен быть: ***<PROJECT>_<PATH>_<FILE>***`_H_`.

Для гарантии уникальности, используйте компоненты полного пути к файлу в дереве проекта. Например, файл `foo/src/bar/baz.h` в проекте `foo` может иметь следующую блокировку:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

Подключайте используемые заголовочные файлы

Если файл с кодом или заголовочный используют внешние зависимости/объявления, тогда в этот файл необходимо напрямую подключать файл с этими объявлениями. В иных случаях не используйте подключение внешнего файла.

Не полагайтесь на вложенные подключения файлов. Это позволит удалить уже не используемые `#include` сохранив корректность другого кода. Правило используется даже в случае парных файлов: даже если `foo.h` подключает `bar.h`, то `foo.cc` также подключает `bar.h` если использует определения из последнего.

Предварительное объявление

По возможности, не используйте предварительное объявление. Вместо этого подключайте используемые заголовочные файлы.

Определение:

"Предварительное объявление" - декларация сущности без соответствующего определения.

```
// В C++ файле:
class B;
void FuncInB();
extern int variable_in_b;
ABSL_DECLARE_FLAG(flag_in_b);
```

За:

- Предварительное объявление может уменьшить время компиляции. Использование `#include` потребует от компилятора сразу открывать (и обрабатывать) больше файлов.
- Предварительное объявление позволит избежать ненужной перекомпиляции. Применение `#include` может привести к частой перекомпиляции из-за различных изменений в заголовочных файлах.

Против:

- Предварительное объявление может скрывать от перекомпиляции зависимости, которые изменились.
- Предварительное объявление (вместо `#include`) затрудняет поиск модуля с необходимым определением для утилит диагностики кода.
- При изменении API, предварительное объявление может стать некорректным. Как результат, предварительное объявление функций или шаблонов может блокировать изменение API: замена типов параметров на похожий, добавление параметров по

умолчанию в шаблон, перенос в новое пространство имён.
- Предварительное объявление символов из `std::` может вызвать неопределённое поведение.
- Иногда тяжело понять, что лучше подходит: предварительное объявление или обычный `#include`. Однако, замена `#include` на предварительное объявление может (без предупреждений) поменять смысл кода:

```
// b.h:
struct B {};
struct D : B {};

// good_user.cc:
#include "b.h"
void f(B*);
void f(void*);
void test(D* x) { f(x); }  // calls f(B*)
```

Если в коде заменить `#include` на предварительное объявление для структур `B` и `D`, то `test()` будет вызывать `f(void*)`.
- Предварительное объявление множества сущностей может быть чересчур объёмным, и может быть проще подключить заголовочный файл.
- Структурирование кода, допускающего предварительное объявление (и использующее указатели как члены класса вместо самих объектов) может сделать код запутанным и медленным.

Вердикт:

Старайтесь избегать предварительного объявления сущностей, объявленных в другом проекте.

Встраиваемые (inline) функции

Определяйте функции как встраиваемые только когда они маленькие, например не более 10 строк.

Определение:

Вы можете объявлять функции встраиваемыми и указать компилятору на возможность включать её напрямую в вызывающий код, помимо стандартного способа с вызовом функции.

За:

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.

Против:

Чрезмерное использование встраиваемых функций может сделать программу медленнее. Также встраиваемые функции, в зависимости от размера её, могут как увеличить, так и уменьшить размер кода. Если это маленькие функции, то код может быть уменьшен. Если же функция большая, то размер кода может очень сильно вырасти. Учтите, что на современных процессорах более компактный код выполняется быстрее благодаря лучшему использованию кэша инструкций.

Вердикт:

Хорошим правилом будет не делать функции встраиваемыми, если они превышают 10 строк кода. Избегайте делать встраиваемыми деструкторы, т.к. они неявно могут содержать много дополнительного кода: вызовы деструкторов переменных и базовых классов!

Ещё одно хорошее правило: обычно нет смысла делать встраиваемыми функции, в которых есть циклы или операции switch (кроме вырожденных случаев, когда цикл или другие операторы никогда не выполняются).

Важно понимать, что встраиваемая функция необязательно будет скомпилирована в код именно так. Например, обычно виртуальные и рекурсивные функции компилируются со стандартным вызовом. Вообще, рекурсивные функции не должны объявляться встраиваемыми. Основная же причина делать встраиваемые виртуальные функции - разместить определение (код) в самом определении класса (для документирования поведения или удобства чтения) - часто используется для get/set функций.

Имена и Порядок включения (include)

Вставляйте заголовочные файлы в следующем порядке: парный файл (например, foo.h - foo.cc), системные файлы C, стандартная библиотека C++, другие библиотеки, файлы вашего проекта.

Все заголовочные файлы проекта должны указываться относительно директории исходных файлов проекта без использования таких UNIX псевдонимов как `.` (текущая директория) или `..` (родительская директория). Например, `google-awesome-project/src/base/logging.h` должен включаться так:

```
#include "base/logging.h"
```

При подключении заголовочных файлов используйте угловые скобки только если это требуется библиотекой. В частности, следующие заголовочные файлы потребуют использования угловых скобок:

- Заголовочные файлы стандартной библиотеки C и C++ (например, `<stdlib.h>` и `<string>`).
- Системные заголовочные файлы POSIX, Linux и Windows (например, `<unistd.h>` и `<windows.h>`).
- В редких случаях это требуется и для других библиотек (например, `<Python.h>`).

Пример: если основная функция файлов ***`dir/foo`***`.cc` и ***`dir/foo_test`***`.cc` это реализация и тестирование кода, объявленного в ***`dir2/foo2`***`.h`, то записывайте заголовочные файлы в следующем порядке:

1. ***`dir2/foo2`***`.h`.
2. ------ Пустая строка

3. Системные заголовочные файлы C и любые другие в угловых скобках с расширением `.h`, например `<unistd.h>`, `<stdlib.h>`, `<Python.h>`.
4. ------ Пустая строка
5. Заголовочные файлы стандартной библиотеки C++ (без расширения в файлах), например `<algorithm>`, `<cstddef>`.
6. ------ Пустая строка
7. Заголовочные `.h` файлы других библиотек.
8. ------ Пустая строка
9. Файлы `.h` вашего проекта.

Отделяйте каждую (непустую) группу файлов пустой строкой.

Такой порядок файлов позволяет выявить ошибки, когда в парном заголовочном файле (***`dir2/foo2.h`***) пропущены необходимые заголовочные файлы (системные и др.) и сборка соответствующих файлов ***`dir/foo.cc`*** или ***`dir/foo`*`_test.cc`*** завершится ошибкой. Как результат, ошибка сразу же появится у разработчика, работающего с этими файлами (а не у другой команды, которая только использует внешнюю библиотеку).

Обычно парные файлы ***`dir/foo.cc`*** и ***`dir2/foo2.h`*** находятся в одной директории (например, `base/basictypes_test.cc` и `base/basictypes.h`), хотя это не обязательно.

Учтите, что заголовочные файлы C, такие, как `stddef.h`, обычно взаимозаменяемы соответствующими файлами C++ (`cstddef`). Можно использовать любой вариант, но лучше следовать стилю существующего кода.

Внутри каждой секции заголовочные файлы лучше всего перечислять в алфавитном порядке. Учтите, что ранее написанный код может не следовать этому правилу. По возможности (например, при исправлениях в файле), исправляйте порядок файлов на правильный.

Например, список заголовочных файлов в `google-awesome-project/src/foo/internal/fooserver.cc` может выглядеть так:

```
#include "foo/server/fooserver.h"

#include <sys/types.h>
#include <unistd.h>

#include <string>
#include <vector>

#include "base/basictypes.h"
#include "foo/server/bar.h"
#include "third_party/absl/flags/flag.h"
```

Исключения:

Бывают случаи, когда требуется включение заголовочных файлов в зависимости от условий препроцессора (например, в зависимости от используемой ОС). Такое включение старайтесь делать как можно короче (локализовано) и размещать после других заголовочных

файлов. Например:

```
#include "foo/public/fooserver.h"

#include "base/port.h"  // For LANG_CXX11.

#ifdef LANG_CXX11
#include <initializer_list>
#endif  // LANG_CXX11
```

Область видимости

Пространство имён

Размещайте свой код в пространстве имён (за некоторыми исключениями). Пространство имён должно иметь уникальное имя, формируемое на основе названия проекта, и, возможно, пути. Не используйте директиву *using* (например, `using namespace foo`). Не используйте встроенные (inline) пространства имён. Для безымянных пространств имён смотрите Внутреннее связывание.

Определение:

Пространства имён делят глобальную область видимости на отдельные именованные области, позволяя избежать совпадения (коллизий) имён.

За:

Пространства имён позволяют избежать конфликта имён в больших программах, при этом сами имена остаются достаточно короткими.

Например, если два разных проекта содержат класс `Foo` в глобальной области видимости, имена могут конфликтовать. Если каждый проект размещает код в своё пространство имён, то `project1::Foo` и `project2::Foo` будут разными именами, конфликтов не будет, в то же время код каждого проекта будет использовать `Foo` без префикса.

Пространства имён inline автоматически делают видимыми свои имена для включающего пространства имён. Рассмотрим пример кода:

```
namespace outer {
  inline namespace inner {
    void foo();
  }  // namespace inner
}  // namespace outer
```

Здесь выражения `outer::inner::foo()` и `outer::foo()` взаимозаменяемы. Inline пространства имён в основном используются для ABI-совместимости разных версий.

Против:

Пространства имён могут запутать программиста, усложнить понимание, что к чему относится.

Пространства имён inline, в частности, могут сбивать с толку т.к. область видимости не ограничена местом определения. Поэтому такой вид пространств имён может быть полезен только при обновлении интерфейсов с сохранением совместимости.

В ряде случаев требуется использование полных имён и это может сделать код сильно перегруженным.

Вердикт:

Используйте пространства имён следующим образом:

- Следуйте правилам Именования Пространств Имён.
- В конце объявления многострочного пространства имён добавляйте комментарий, аналогично показанным в примерах.
- Заключайте в пространство имён целиком файл с исходным кодом после #include-ов, объявлений/определений gflag-ов и предварительных объявлений классов из других пространств имён.

```
// В .h файле
namespace mynamespace {

// Все объявления внутри блока пространства имён.
// Обратите внимание на отсутствие отступа.
class MyClass {
 public:
  ...
  void Foo();
};

}  // namespace mynamespace
```

```
// В .cc файле
namespace mynamespace {

// Определение функций внутри блока пространства имён.
void MyClass::Foo() {
  ...
}

}  // namespace mynamespace
```

В `.cc` файлах могут быть дополнительные объявления, такие как флаги или using-декларации.

```
#include "a.h"

ABSL_FLAG(bool, someflag, false, "a flag");

namespace mynamespace {

using ::foo::Bar;

...code for mynamespace...    // Код начинается с самой левой границы.

}  // namespace mynamespace
```

- Чтобы генерируемый из protobuf-а код был размещён в требуемом пространстве имён, применяйте спецификатор `package` в `.proto` файле. Подробнее здесь: Protocol Buffer Packages.
- Ничего не объявляйте в пространстве `std`, в том числе и предварительные объявления классов стандартной библиотеки. Объявление в пространстве имён `std` приведёт к неопределённому поведению (UB) и это будет непереносимый код. Для объявления сущностей используйте соответствующий заголовочный файл.
- Не используйте *using-директиву*, чтобы сделать доступными все имена из пространства имён.

```
// Недопустимо -- Это загрязняет пространство имён.
using namespace foo;
```

- Не используйте *псевдонимы пространств имён* в блоке namespace в заголовочном файле, за исключением явно обозначенных "внутренних" пространств имён. Связано это с тем, что любая декларация в заголовочном файле становится частью публичного API, экспортируемого этим файлом.

```
// Укороченная запись для доступа к часто используемым именам в .cc файлах.
namespace baz = ::foo::bar::baz;
```

```
// Укороченная запись для доступа к часто используемым именам (в .h файле).
namespace librarian {
namespace internal {  // Внутреннее содержимое, не являющееся частью API.
namespace sidetable = ::pipeline_diagnostics::sidetable;
}  // namespace internal

inline void my_inline_function() {
  // Пространство имён, локальное для функции (или метода).
  namespace baz = ::foo::bar::baz;
  ...
}
}  // namespace librarian
```

- Не используйте inline-пространства имён.
- Если пространство имён не предназначено для пользователей API, то используйте слово "internal" в его имени.

```
// Не следует использовать такие внутренние имена в не-absl коде.
using ::absl::container_internal::ImplementationDetail;
```

- В новом коде предпочтительно декларировать пространство имён в одну строчку. Это не является обязательным требованием.

Внутреннее связывание

Когда определения внутри `.cc` файла не используются в других исходных файлах, используйте для них внутреннее связывание: размещайте такие определения в безымянном пространстве имён или объявляйте их как `static`. Не используйте такие конструкции в `.h` файлах.

Определение:

Размещённые в безымянном пространстве имён объявления сразу получают внутреннее связывание. Функции и переменные также могут быть с внутренним связыванием, если они заявлены как `static`. Такие типы объявления подразумевают, что они будут недоступны из другого файла. Если другой файл объявляет сущность с таким же именем, то оба объявления будут полностью независимы.

Вердикт:

Использование внутреннего связывания в `.cc` файлах предпочтительно для любого кода, к которому не обращаются снаружи (из других файлов). Не используйте внутреннее связывание в `.h` файлах.

Формат описания безымянного пространства имён полностью аналогичен именованному варианту. Не забывайте к закрывающей скобке написать комментарий, в котором имя оставьте пустым:

```
namespace {
...
}  // namespace
```

Функции: глобальные, статические, вне класса

Предпочтительно заключать отдельные функции (вне класса) в пространство имён. Использование полностью глобальных функций должно быть минимальным. Также не используйте класс только лишь для группировки функций, объявляя их статическими. Статические методы класса должны использоваться при совместной работе с экземплярами класса или его статическими данными.

За:

Статические функции-члены или функции вне класса могут быть полезными в отдельных ситуациях. И размещение функций - не членов в пространстве имён позволяет содержать в чистоте глобальное пространство имён.

Против:

Иногда разумнее статические функции класса и функции вне класса сгруппировать в одном месте, в новом классе. Например, когда у них сложные зависимости от всего или им нужен доступ к внешним ресурсам.

Вердикт:

Иногда полезно объявить функцию, не привязанную к экземпляру класса. И можно сделать либо статическую функцию в классе, либо внешнюю (вне класса) функцию. Желательно, чтобы функция-вне-класса не использовала внешних переменных и находилась в пространстве имён. Не создавайте классы только для группировки статических членов: это всё равно, что дать именам некий префикс и группировка становится лишней.

Если определяется функция вне класса и она используется только в этом `.cc`-файле, то используйте внутреннее связывание для ограничения области видимости.

Локальные переменные

Объявляйте переменные внутри функции в наиболее узкой области видимости, инициализируйте такие переменные при объявлении.

Язык C++ позволяет объявлять переменные в любом месте функции. Однако рекомендуется делать это в наиболее вложенной области видимости и по возможности ближе к первому использованию переменной. Это облегчает поиск объявлений, проще узнать тип переменной и её начальное значение. Также рекомендуется использовать инициализацию, а не объявление с присваиванием. Примеры:

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

```
int j = f();  // Хорошо -- объявление с инициализацией.
```

```
int jobs = NumJobs();
// ... другой код ...
f(jobs);      // Плохо -- декларация отделена от использования.
```

```
int jobs = NumJobs();
f(jobs);      // Хорошо -- декларация прямо перед использованием (или близко к нему).
```

```
std::vector<int> v;
v.push_back(1);  // Желательно инициализировать с помощью {}.
v.push_back(2);
```

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

Переменные, необходимые только внутри кода `if`, `while` и `for` лучше объявлять внутри условий, тогда область их видимости будет ограничена только соответствующим блоком кода:

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

Однако учитывайте одну тонкость: если переменная есть экземпляр объекта, то при каждом входе в область видимости будет вызываться конструктор, и, соответственно, при выходе будет вызываться деструктор.

```
// Неэффективная реализация:
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // Конструктор и деструктор Foo вызовутся по 1000000 раз каждый.
  f.DoSomething(i);
}
```

Возможно было бы более эффективно такую переменную (которая используется внутри цикла) объявить вне цикла:

```
Foo f;  // Конструктор и деструктор Foo вызовутся по разу.
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

Переменные: статические и глобальные

Объекты в статической области видимости/действия запрещены, кроме тривиально удаляемых. Фактически это означает, что деструктор должен ничего не делать (включая вложенные или базовые типы). Формально это можно описать, что тип не содержит пользовательского или виртуального деструктора и что все базовые типы и не-статические члены ведут себя аналогично (т.е. являются тривиально удаляемыми). Статические переменные в функциях могут быть динамически инициализированными. Использование же динамической инициализации для статических членов класса или переменных в области пространства имён (namespace) в целом не рекомендуется, однако допустимо в ряде случаев (см. ниже).

Эмпирическое правило: если глобальную переменную (рассматривая её изолированно) можно объявить как `constexpr`, значить она соответствует вышеуказанным требованиям.

Определение:

Каждый объект имеет тот или иной тип *времени жизни / storage duration*, и, очевидно, это влияет на время жизни объекта. Объекты статического типа доступны с момента их инициализации до момента завершения программы. Такие объекты могут быть переменными в пространстве имён ("глобальные переменные"), статическими членами классов, локальными переменными внутри функций со спецификатором `static`. Статические переменные в функциях инициализируются, когда поток выполнения кода проходит в первый раз через объявление; все остальные объекты статического типа инициализируются в фазе старта (start-up) приложения. Все объекты статического типа удаляются в фазе завершения программы (до обработки незавершённых(unjoined) потоков).

Инициализация может быть *динамическая*, т.е. во время инициализации делается что-то нетривиальное: например, конструктор выделяет память, или переменная инициализируется идентификатором процесса. Также инициализации может быть *статической*. Сначала выполняется статическая инициализация: для *всех* объектов статического типа (объект инициализируется либо заданной константой, либо заполняется нулями). Далее, если необходимо, выполняется динамическая инициализация.

За:

Глобальные и статические переменные бывают очень полезными: константные имена, дополнительные структуры данных, флаги командной строки, логирование, регистрирование, инфраструктура и др.

Против:

Глобальные и статические переменные с динамической инициализацией или нетривиальным деструктором могут сильно усложнить код и привести к трудно обнаруживаемым багам. Порядок динамической инициализации (и разрушения) объектов может быть различным когда есть несколько единиц трансляции. И, например, когда одна из инициализаций ссылается на некую переменную статического типа, то возможна ситуация доступа к объекту до корректного начала его жизненного цикла (до полного конструирования), или уже после окончания жизненного цикла. Если программа создаёт несколько потоков, которые не завершаются к моменту выхода из программы, то они могут попытаться получить доступ к объектам, которые уже разрушены.

Вердикт:

Когда деструктор тривиальный, тогда порядок разрушения в принципе не важен. В противном случае есть риск обратиться к объекту после его разрушения. Поэтому, настоятельно рекомендуется использовать только переменные со статическим типом размещения (конечно, если они имеют тривиальный деструктор). Фундаментальные типы (указатели или `int`), как и массивы из них, являются тривиально разрушаемыми. Переменные с типом constexp также тривиально разрушаемые.

```
const int kNum = 10;  // Допустимо

struct X { int n; };
const X kX[] = {{1}, {2}, {3}};  // Допустимо

void foo() {
  static const char* const kMessages[] = {"hello", "world"};  // Допустимо
}

// Допустимо: constexpr гарантирует, что деструктор будет тривиальный
constexpr std::array<int, 3> kArray = {1, 2, 3};
```

```
// Плохо: нетривиальный деструктор
const std::string kFoo = "foo";

// Плохо по тем же причинам (хотя kBar и является ссылкой, но
// правило применяется и для временных объектов в расширенным временем жизни)
const std::string& kBar = StrCat("a", "b", "c");

void bar() {
  // Плохо: нетривиальный деструктор
  static std::map<int, int> kData = {{1, 0}, {2, 0}, {3, 0}};
}
```

Отметим, что ссылка не есть сам объект, и, следовательно, к ним не применяются ограничения по разрушению объекта. Хотя ограничения на динамическую инициализацию остаются в силе. В частности, внутри функции допустим следующий код `static T& t = *new T;`.

Тонкости инициализации

Инициализация может быть запутанной: мало того, что конструктору нужно (желательно правильно) отработать, так есть ещё и предварительные вычисления:

```
int n = 5;    // Отлично
int m = f();  // ? (Зависит от f)
Foo x;        // ? (Зависит от Foo::Foo)
Bar y = g();  // ? (Зависит от g и Bar::Bar)
```

На выполнение всех выражений, кроме первого, может повлиять порядок инициализации, который может быть разным/неопределённым (или зависимым от ...).

Рассмотрим идею *константной инициализации*, как она обозначается в стандарте C++. Это означает, что инициализационное выражение - константное, и если при создании объекта вызывается конструктор, то он (конструктор) тоже должен быть заявлен как `constexpr`:

```
struct Foo { constexpr Foo(int) {} };

int n = 5;  // Отлично, 5 - константное выражение
Foo x(2);   // Отлично, 2 - константное выражение и вызывается constexpr конструктор
Foo a[] = { Foo(1), Foo(2), Foo(3) };  // Отлично
```

Константная инициализация является рекомендуемой для большинства случаев. Константную инициализацию переменных со статическим размещением рекомендуется помечать как `constexpr` или `constinit`. Любую не-локальную переменную со статическим размещением и без указанной выше маркировки следует считать динамически инициализируемой (и тщательно проверять на ревью кода).

Например, следующие инициализации могут привести к проблемам:

```
// Объявления
time_t time(time_t*);      // Не constexpr !
int f();                   // Не constexpr !
struct Bar { Bar() {} };
```

```
// Проблемные инициализации
time_t m = time(nullptr);  // Инициализационное выражение не константное
Foo y(f());                // Те же проблемы
Bar b;                     // Конструктор Bar::Bar() не является constexpr
```

Динамическая инициализация переменных вне функций не рекомендуется. В общем случае это запрещено, однако, это можно делать если никакой код программы не зависит от порядка инициализации этой переменной среди других: в этом случае изменение порядка инициализации не может что-то поломать. Например:

```
int p = getpid();  // Допустимо, пока другие статические переменные
                   // не используют p в своей инициализации
```

Динамическая же инициализация статических переменных в функциях (локальных) допустима и является широко распространённой практикой.

Стандартные практики

- Глобальные строки: если требуется именованная глобальная или статическая строковая константа, то рекомендуется использовать переменную `constexpr string_view`, символьный массив или указатель на первый символ строкового литерала. Строковые литералы обычно находятся в статическом (по времени жизни) размещении и этого в большинстве случаев достаточно. Смотрите также: TotW #140.
- Динамические контейнеры (map, set и т.д.): если требуется статическая коллекция с фиксированными данными (например, таблицы значений для поиска), то не используйте динамические контейнеры из стандартной библиотеки как тип для статической переменной, т.к. у этих контейнеров нетривиальный деструктор. Вместо этого попробуйте использовать массивы простых (тривиальных) типов, например массив из массивов целых чисел (вместо `std::map<int, int>`) или, например, массив структур с полями `int` и `const char*`. Учтите, что для небольших коллекций линейный поиск обычно вполне приемлем (и может быть очень эффективным благодаря компактному размещению в памяти). Также можете воспользоваться алгоритмами absl/algorithm/container.h для стандартных операций. Также возможно создавать коллекцию данных уже отсортированной и использовать алгоритм бинарного поиска. Если без динамического контейнера не обойтись, то попробуйте использовать статическую переменную-указатель, объявленную в функции (см. ниже).
- Умные указатели (`std::unique_ptr`, `std::shared_ptr`): умные указатели освобождают ресурсы в деструкторе и поэтому использовать их нельзя. Попробуйте применить другие практики/способы, описанные в разделе. Например, одно из простых решений это использовать обычный указатель на динамически выделенный объект и далее никогда не удалять его (см. последний вариант списка).
- Статические переменные пользовательского типа: если требуется статический и константный пользовательский тип, заполненный данными, то можете объявить у этого типа тривиальный деструктор и `constexpr` конструктор.
- Если все другие способы не подходят, то можно создать динамический объект и никогда не удалять его. Объект можно создать с использованием статического указателя или ссылки, объявленной в функции, например: `static const auto& impl = *new T(args...);`.

Потоковые переменные (thread_local)

Потоковые переменные (`thread_local`), объявленные вне функций должны быть инициализированы константой, вычисляемой во время компиляции. И это должно быть сделано с помощью атрибута `constinit`. В целом, для определения данных, специфичных для каждого потока, использование `thread_local` является наиболее предпочтительным.

Определение:

Переменные можно объявлять со спецификатором `thread_local`:

```
thread_local Foo foo = ...;
```

Каждая такая переменная представляется собой коллекцию объектов. Разные потоки работают с разными экземплярами переменной (каждый со своим экземпляром). По поведению переменные `thread_local` во многом похожи на Переменные со статическим типом размещения. Например, они могут быть объявлены в пространстве имён, внутри функций, как статические члены класса (как обычные члены класса - нельзя).

Инициализация потоковых переменных очень напоминает статические переменные, за исключением, что это делается для каждого потока. В том числе это означает, что безопасно объявлять `thread_local` переменные внутри функции. Однако в целом `thread_local` переменные подвержены тем же проблемам, что и статические переменные (различный порядок инициализации и т.д.).

У переменных `thread_local` есть тонкость, связанная с порядком удаления: при завершении потока переменные `thread_local` будут разрушены в порядке, обратном порядку инициализации (это штатное поведение в C++). Если же код, вызываемый деструктором одной `thread_local` переменной, использует другую, уже разрушенную, `thread_local` переменную этого потока, то можно получить трудно диагностируемую проблему использование-после-удаления (use-after-free).

За:

- Потоковые переменные в принципе защищены от эффекта гонок (т.к. каждый поток обычно имеет доступ только к своему экземпляру), что очень полезно для программирования потоков.
- Переменные `thread_local` являются единственным стандартизованным средством для создания локальных потоковых данных.

Против:

- Операции с `thread_local` переменными могут неявно вызвать выполнение другого кода, который не предполагался и его размер также может быть различным. Такое выполнение кода может произойти как при старте потока, так и при первом использовании в данном потоке.
- Переменные `thread_local` являются по сути глобальными переменными со всеми их недостатками (за исключением потокобезопасности).

- Выделяемая для `thread_local` память зависит от количества потоков, и её размер может быть значительным.
- Нестатичные (не `static`) члены данных не могут быть `thread_local`.
- Могут возникнуть ошибки использование-после-освобождения (use-after-free), если у `thread_local` переменных сложный, запутанный деструктор. В идеале, деструктор любой такой переменной не должен вызывать любой код (даже через вложенные вызовы), ссылающийся на потенциально удалённые `thread_local` переменные. Хотя это может быть трудно реализуемым.
- Подходы к устранению проблем использования-после-удаления (use-after-free), которые работают для глобального или статического контекстов, могут неработать для случая `thread_local`. В частности, намеренное отсутствие деструкторов для глобальных и статических переменных ещё можно допустить, так как жизненный цикл таких переменных завершается при закрытии программы. Таким образом, любая утечка памяти или ресурсов будет сразу же исправлена операционной системой. В противоположность этому, отсутствие деструктора для `thread_local` переменных может привести к утечке ресурсов, тем большей, чем больше завершается потоков в процессе работы программы.

Вердикт:

Переменные `thread_local` в области видимости класса или пространства имён должны быть инициализированы константой времени компиляции (т.е. не должна использоваться динамическая инициализация). Поэтому переменные `thread_local` в классе или пространстве имён должны быть аннотированы как `constinit` (или `constexpr`, однако это скорее как исключение):

```
   constinit thread_local Foo foo = ...;
```

У переменных `thread_local`, заявленных внутри функций, нет проблем с инициализацией, однако ещё остаются риски использования-после-удаления (use-after-free) при завершении потока. Отметим, что возможно использовать объявленную внутри функции переменную `thread_local` и вне функции. Для этого нужна функция доступа к переменной:

```
Foo& MyThreadLocalFoo() {
  thread_local Foo result = ComplicatedInitialization();
  return result;
}
```

Учитывайте, что переменные `thread_local` разрушаются при завершении потока. Если деструктор любой такой переменной использует какую-либо другую (потенциально удалённую) переменную `thread_local`, это может привести к появлению трудно диагностируемых ошибок использования-после-удаления (use-after-free). Более предпочтительным является использование тривиальных типов или таких, в которых не выполняется пользовательский код в деструкторе, чтобы минимизировать возможность доступа к другим `thread_local` переменным.

Переменные `thread_local` должны быть предпочтительным способом определения потоко-специфичных данных.

Классы

Классы являются основным строительным блоком в C++. И, конечно же, используются они часто. В этой секции описаны основные правила и запреты, которым нужно следовать при использовании классов.

Код в конструкторе

Не вызывайте виртуальные методы в конструкторе. Избегайте инициализации, которая может завершиться ошибкой (а способа сигнализировать об ошибке не предусмотрено. Прим.: учтите, что Гугл не любит исключения).

Определение:

Вообще в конструкторе можно выполнять любые инициализации (т.е. всю инициализацию сделать в конструкторе).

За:

- Не нужно беспокоиться об неинициализированном классе.
- Объекты, которые полностью инициализируются в конструкторе, могут быть константными (`const`) и также их легче использовать в стандартных контейнерах и алгоритмах.

Против:

- Если в конструкторе вызываются виртуальные функции, то не вызываются реализации из производного класса. Даже если сейчас класс не имеет потомков, в будущем это может обернуться проблемой.
- Нет простого способа проинформировать об ошибке без крэша программы (что не всегда допустимо) или выбрасывания исключений (которые запрещены).
- Если возникла ошибка, то у нас есть частично (обычно - неправильно) инициализированный объект. Очевидное действие: добавить механизм проверки состояния `bool IsValid()`. Однако про эту проверку легко забыть.
- Вы не можете пользоваться адресом конструктора. Поэтому нет, например, простого способа передать выполнение конструктора в другой поток.

Вердикт:

Конструкторы не должны вызывать виртуальные функции. В ряде случаев (если это позволительно) обработка ошибок конструирования возможна через завершение программы. В иных случаях рассмотрите паттерн Фабричный Метод или используйте `Init()` (подробнее здесь: TotW #42). Используйте `Init()` только в случае, если у объекта есть флаги состояния, разрешающие вызывать те или иные публичные функции (т.к. сложно полноценно работать с частично сконструированным объектом).

Неявные преобразования

Не объявляйте неявные преобразования. Используйте ключевое слово `explicit` для операторов преобразования типа и конструкторов с одним аргументом.

Определение:

Неявные преобразования позволяют объект одного типа (*source type*) использовать там, где ожидается другой тип (*destination type*), например передача аргумента типа `int` в функцию, ожидающую `double`.

Помимо неявных преобразований, задаваемых языком программирования, можно также определять свои пользовательские, добавляя соответствующие члены в объявление класса (как источника, так и получателя). Неявное преобразование на стороне источника объявляется как оператор + тип получателя (например, `operator bool()`). Неявное преобразование на стороне получателя реализуется конструктором, принимающим тип источника как единственный аргумент (помимо аргументов со значениями по умолчанию).

Ключевое слово `explicit` может применяться к конструктору или к оператору преобразования для явного указания, что функция может применяться только при явном соответствии типов (например, после операции приведения). Это применяется не только для неявного преобразования, но и для списков инициализации:

```
class Foo {
  explicit Foo(int x, double y);
  ...
};

void Func(Foo f);
```

```
Func({42, 3.14});  // Ошибка
```

Этот пример кода технически не является неявным преобразованием, но язык трактует это как будто подразумевается `explicit`.

За:

- Неявные преобразования могут сделать тип более удобным в использовании, не требуя явного указания типа в очевидных случаях.
- Неявные преобразования могут быть упрощённой альтернативой для перегрузки, например когда одна функция с аргументом типа `string_view` применяется вместо отдельных версий для `std::string` и `const char*`.
- Применение списка инициализации является компактным и понятным способом инициализации объектов.

Против:

- Неявные преобразования могут скрывать баги с несоответствием типов, когда получаемый тип не соответствует ожиданиям пользователя (если он вообще предполагал приведение типа).
- Неявные преобразования могут усложнить чтение кода, особенно при наличии перегруженных функций: становится неясно, какой код действительно будет вызван.
- Конструкторы с одним аргументом могут быть случайно использованы как неявное преобразование, даже если это не предполагалось.
- Когда конструктор с одним аргументом не объявлен как `explicit` нельзя с уверенностью сказать: то ли это такое неявное преобразование, то ли автор забыл ключевое слово.
- Неявные преобразования могут приводить к ошибке двусмысленности вызова, особенно когда есть несколько вариантов преобразования. Такое может быть вызвано, например, наличием двух типов, оба допускают неявные преобразования. Или (в случае одно типа) одновременным наличием неявного конструктора и неявного оператора преобразования.
- Использование списка инициализации также может добавить проблем, если целевой тип задан неявно и, особенно, если сам список состоит только из одного элемента.

Вердикт:

Операторы преобразования типа и конструкторы с одним аргументом должны объявляться с ключевым словом `explicit`. Есть и исключение: конструкторы копирования и перемещения могут объявляться без `explicit`, т.к. они не выполняют преобразование типов.

Также неявные преобразования могут быть необходимы (вполне подходящими) для типов, которые проектируются как взаимозаменяемые: например, когда объекты двух типов являются различными представлениями одного и того же внутреннего значения. В этом случае для игнорирования этого важного правила обязательно запросите разрешение у вышестоящего руководства.

Конструкторы, которые нельзя вызвать с одним аргументом, можно объявлять без `explicit`. Конструкторы, принимающие единственный `std::initializer_list` также должны объявляться без `explicit` для поддержки инициализации копированием (например, `MyType m = {1, 2};`).

Копируемые и перемещаемые типы

Открытый интерфейс класса должен явно указывать на возможность копирования и/или перемещения, или наоборот всё запрещать. Поддерживайте копирование и/или перемещение, только если эти операции имеют смысл для вашего типа.

Определение:

Перемещаемый тип - тот, что может быть инициализирован или присвоен из временных значений.

Копируемый тип - может быть инициализирован или присвоен из другого объекта того же типа (т.е. также, как и перемещаемый), с условием, что исходный объект остаётся неизменным. Например, `std::unique_ptr<int>` - это перемещаемый, но не копируемый тип (т.к. значение исходного `std::unique_ptr<int>` объекта должно измениться при присвоении целевому объекту). `int` и `std::string` - примеры перемещаемый типов, которые также можно копировать: для `int` операции перемещения и копирования одинаковые, для `std::string` операция перемещения требует меньше ресурсов, чем копирование.

Для пользовательских типов копирование задаётся конструктором копирования и оператором копирования. Перемещение задаётся либо конструктором перемещения с оператором перемещения, либо (если их нет) соответствующими функциями копирования.

Конструкторы копирования и перемещения могут неявно вызываться компилятором, например при передаче объектов по значению.

За:

Объекты копируемых и перемещаемых типов могут быть переданы и получены по значению, что делает API проще, безопаснее, универсальнее. В этом случает нет проблем с владением объекта, его жизненным циклом, изменением значения и т.п., а также не требуется указывать их в "контракте" (всё это в отличие от передачи объектов по указателю или ссылке). Также предотвращается отложенное взаимодействие между клиентом и реализацией, что существенно облегчает понимание и поддержку кода, а также его оптимизацию компилятором. Такие объекты могут использоваться как аргументы других классов, требующих передачу по значению, (например, большинство контейнеров), и вообще они гибче (например, при использовании в паттернах проектирования).

Конструкторы копирования/перемещения и соответствующие операторы присваивания обычно легче определить, чем альтернативы наподобие `Clone()`, `CopyFrom()` или `Swap()`, т.к. компилятор может сгенерировать требуемые функции (неявно или посредством `= default`). Они (функции) легко объявляются и можно быть уверенным, что все члены класса будут скопированы. Конструкторы (копирования и перемещения) в целом более эффективны, т.к. не требуют выделения памяти, отдельной инициализации, дополнительных присвоений, хорошо оптимизируются (см. copy elision).

Операторы перемещения позволяют эффективно (и неявно) управлять ресурсами rvalue объектов. Иногда это упрощает кодирование.

Против:

Некоторым типам не требуется быть копируемыми, и поддержка операций копирования может противоречить логике или привести к некорректной работе. Типы для синглтонов (`Registerer`), объекты для очистки (например, при выходе за область видимости) (`Cleanup`) или содержащие уникальные данные (`Mutex`) по своему смыслу являются некопируемыми. Также, операции копирования для базовых классов, имеющих наследников, могут привести к "разделению объекта" object slicing. Операции копирования по умолчанию (или неаккуратно написанные) могут привести к ошибкам, которые тяжело обнаружить.

Конструкторы копирования вызываются неявно и это легко упустить из виду (особенно для программистов, которые раньше писали на языках, где передача объектов производится по ссылке). Также можно снизить производительность, делая лишние копирования.

Вердикт:

Открытый интерфейс каждого класса должен явно указывать, какие операции копирования и/или перемещения он поддерживает. Обычно это делается в секции public в виде явных деклараций нужных функций или объявлением их как delete.

В частности, копируемый класс должен явно объявлять операции копирования; только перемещаемый класс должен явно объявить операции перемещения; некопируемый/неперемещаемый класс должен явно запретить операции копирования. Копируемый класс может объявить операции перемещения, если перемещение является целесообразным. Явная декларация или удаление всех четырёх функций копирования и перемещения допустима, но не является обязательной. Если вы реализуете оператор копирования и/или перемещения, то обязательно нужно сделать соответствующий конструктор.

```
class Copyable {
 public:
  Copyable(const Copyable& other) = default;
```

```
  Copyable& operator=(const Copyable& other) = default;

  // Неявное определение операций перемещения будет запрещено (т.к. объявлено копирование)
  // Допустимо явно определить операции перемещения (при целесообразности)
};

class MoveOnly {
 public:
  MoveOnly(MoveOnly&& other) = default;
  MoveOnly& operator=(MoveOnly&& other) = default;

  // Неявно определённые операции копирования удаляются. Но (если хотите) можно это записать явно:
  MoveOnly(const MoveOnly&) = delete;
  MoveOnly& operator=(const MoveOnly&) = delete;
};

class NotCopyableOrMovable {
 public:
  // Такое объявление запрещает и копирование и перемещение
  NotCopyableOrMovable(const NotCopyableOrMovable&) = delete;
  NotCopyableOrMovable& operator=(const NotCopyableOrMovable&)
      = delete;

  // Хотя операции перемещения запрещены (неявно), можно записать это явно:
  NotCopyableOrMovable(NotCopyableOrMovable&&) = delete;
  NotCopyableOrMovable& operator=(NotCopyableOrMovable&&)
      = delete;
};
```

Описываемые объявления или удаления функций можно опустить в очевидных случаях:

- Если класс не содержит секции `private` (например, структура struct или класс-интерфейс), то копируемость и перемещаемость можно заявить через аналогичное свойство любого открытого члена.
- Если базовый класс явно некопируемый и неперемещаемый, наследные классы будут такими же. Однако, если базовый класс не объявляет это операции, то этого будет недостаточно для прояснения свойств наследуемых классов.
- Заметим, что если (например) конструктор копирования объявлен/удалён, то нужно и явно объявить/удалить оператор копирования (т.к. его статус неочевиден). Аналогично и для объявления/удаления оператора копирования. Аналогично и для операций перемещения.

Тип не следует объявлять копируемым/перемещаемым, если для обычного программиста не понятна необходимость этих операций или если операции очень требовательны к ресурсам и производительности. Операции перемещения для копируемых типов это всегда оптимизация производительности, но с другой стороны - это потенциальный источник багов и усложнений. Поэтому не объявляйте

операции перемещения, если они не дают значительного выигрыша по производительности по сравнению с копированием. Вообще желательно (если для класса заявляются операции копирования) всё спроектировать так, чтобы использовались функции копирования по-умолчанию. И обязательно проверьте корректность работы любых операций по-умолчанию.

Для исключения риска "слайсинга" предпочтительным будет сделать базовый класс абстрактным, сделав конструкторы protected, объявив деструкторы protected или добавив одну/несколько чистых виртуальных функций. Рекомендуется избегать наследования от обычных классов.

Структуры vs Классы

Используйте структуры (`struct`) только для пассивных объектов, хранящих данные. В других случаях используйте классы (`class`).

Ключевые слова `struct` и `class` практически идентичны в C++. Однако, у нас есть собственное понимание для каждого ключевого слова, поэтому используйте то, которое подходит по назначению и смыслу.

Структуры должны использоваться для пассивных объектов, передающих данные. Также они могут содержать ассоциированные константы. Все поля должны быть открытыми (public). Структуры не должны содержать инварианты, которые основаны на зависимости между различными полями структуры, так как возможность напрямую изменять поля может сделать инвариант невалидным. Конструкторы, деструкторы и вспомогательные методы могут использоваться в структурах, однако эти методы не должны требовать или использовать любые инварианты.

Если требуется дополнительная функциональность в обработке данных или инварианты, если известная структура планируется к расширению, то предпочтительно применение классов (`class`). Если сомневаетесь, что выбрать - лучше используйте классы.

В ряде случаев (шаблонные мета-функции, traits, некоторые функторы) для единообразия с STL допускается использование структур вместо классов.

Не забудьте, что переменные в структурах и классах именуются разными стилями.

Структуры vs пары (pair) и кортежи (tuple)

Если отдельные элементы в блоке данных могут осмысленно называться, то желательно использовать структуры вместо пар или кортежей.

Хотя использование пар и кортежей позволяет не изобретать велосипед с собственным типом и сэкономит много времени при *написании* кода, поля с осмысленными именами (вместо `.first`, `.second` или `std::get<X>`) будут более понятны при *чтении* кода. И хотя C++14 для кортежей в дополнение к доступу по индексу добавляется доступ по типу (`std::get<Type>`, а тип должен быть уникальным), имя поля намного более информативно нежели тип.

Пары и кортежи являются подходящими в коде, где нет специального различия между элементами пары или кортежа. Также они требуются для работы с существующим кодом или API.

Наследование

Часто композиция класса является более подходящей, чем наследование. Когда используйте наследование, делайте его открытым (`public`).

Определение:

Когда дочерний класс наследуется от базового, он включает определения всех данных и операций от базового. "Наследование интерфейса" - это наследование от чистого абстрактного базового класса (в нём не определены состояние или методы). Всё остальное - это "наследование реализации".

За:

Наследование реализации уменьшает размер кода благодаря повторному использованию частей базового класса (который становится частью нового класса). Т.к. наследование является декларацией времени компиляции, это позволяет компилятору понимать структуру и находить ошибки. Наследование интерфейса может быть использовано чтобы класс поддерживал требуемый API. И также, компилятор может находить ошибки, если класс не определяет требуемый метод наследуемого API.

Против:

В случае наследования реализации, код начинает размазываться между базовым и дочерним классом и это может усложнить понимание кода. Также, дочерний класс не может переопределять код не-виртуальных функций (не может менять их реализацию).

Множественное наследование ещё более проблемное, а также иногда приводит к уменьшению производительности. Часто просадка производительности при переходе от одиночного наследования к множественному может быть больше, чем переход от обычных функций к виртуальным. Также от множественного наследования один шаг до ромбического, а это уже ведёт к неопределённости, путанице и, конечно же, багам.

Вердикт:

Любое наследование должно быть открытым (`public`). Если хочется сделать закрытое (`private`), то лучше добавить новый член с экземпляром базового класса. Также допустимо использовать `final` для классов, которые не планируются использовать как базовые классы.

Не злоупотребляйте наследованием реализации. Композиция классов часто более предпочтительна. Попытайтесь ограничить использование наследования семантикой "Является": `Bar` можно наследовать от `Foo`, если можно сказать, что `Bar` "Является" `Foo` (т.е. там, где используется `Foo`, можно также использовать и `Bar`).

Защищёнными (`protected`) делайте лишь те функции, которые должны быть доступны для дочерних классов. Обратите внимание, что данные должны быть закрытыми (`private`).

Явно декларируйте переопределение виртуальных функций/деструктора с помощью спецификаторов: либо `override`, либо (если требуется) `final`. Не используйте спецификатор `virtual` при переопределении функций. Объяснение: функция или деструктор, помеченные `override` или `final`, но не являющиеся виртуальными, просто не скомпилируются (что помогает обнаружить общие ошибки). Также спецификаторы работают как документация; а если спецификаторов нет, то программист будет вынужден проверить всю иерархию, чтобы уточнить виртуальность функции.

Множественное наследование допустимо, однако множественное наследование *реализации* не рекомендуется от слова совсем.

Перегрузка операторов

Перегружайте операторы в рамках разумного. Не используйте пользовательские литералы.

Определение:

C++ позволяет пользовательскому коду переопределять встроенные операторы используя ключевое слово `operator` и пользовательский типа как один из параметров; также `operator` позволяет определять новые литералы, используя `operator""`; также можно создавать функции приведения типов, наподобие `operator bool()`.

За:

Использование перегрузки операторов для пользовательских типов (по аналогии со встроенными типами) может сделать код более сжатым и интуитивным. Перегружаемые операторы соответствуют определённым операциям (например, ==, <, = и <<) и если код следует логике применения этих операций, то пользовательские типы можно сделать понятнее и использовать при работе с внешними библиотеками, которые опираются на эти операции.

Пользовательские литералы - очень эффективный способ для создания пользовательских объектов.

Против:

- Написать комплект операторов для класса (корректных, согласованных и логичных) - это может потребовать известных усилий и, при недоработанном коде, это может обернуться труднопонимаемыми багами.
- Излишняя перегрузка операторов может усложнить понимание кода, особенно если код не соответствует логике операции.
- Все недостатки, связанные с перегрузкой функций, присущи и перегрузке операторов.
- Перегрузка операторов обмануть других программистов, ожидающих простую и быструю встроенную операцию, а получающие нечто ресурсоёмкое.
- Поиск мест вызова перегруженных операторов может быть нетривиальной задачей, и это явно сложнее обычного текстового поиска.
- При ошибках в типах аргументов вы можете вместо сообщений об ошибке/предупреждении от компилятора (по которым легко найти проблему и исправить её), получить "корректный" вызов другого оператора. Например, код для `foo < bar` может сильно отличаться от кода для `&foo < &bar`; немного напутав в типах получим ошибочный вызов.
- Перегрузка некоторых операторов является в принципе рискованным занятием. Перегрузка унарного & может привести к тому, что один и тот же код будет трактоваться по-разному в зависимости от видимости декларации этой перегрузки. Перегрузка

операторов &&, || и , (запятая) может поменять порядок (и правила) вычисления выражений.
- Часто операторы определяются вне класса, и есть риск использования разных реализаций одного и того же оператора. Если оба определения будут слинкованы в один бинарный файл, можно получить неопределённое поведение и хитрые баги.
- Пользовательские литералы (UDL) позволяют создавать новые синтаксические формы, незнакомые даже продвинутым C++ программистам. Например: `"Hello World"sv` как сокращение для `std::string_view("Hello World")`. Исходная нотация может быть более понятной, хотя и не такой компактной.
- Т.к. для UDL не указывается пространство имён, потребуется либо использовать using-директиву (которая запрещена) или using-декларацию (которая также запрещена (в заголовочных файлах), кроме случая когда импортируемые имена являются частью интерфейса, показываемого в заголовочном файле). Для таких заголовочных файлов лучше бы избегать суффиксов UDL, и желательно избегать зависимости между литералами, которые различны в заголовочном и исходном файле.

Вердикт:

Определяйте перегруженные операторы только если их смысл очевиден, понятен, и соответствует общей логике. Например, используйте | в смысле операции ИЛИ; реализовывать же вместо этого логику канала (pipe) - не очень хорошая идея.

Определяйте операторы только для ваших собственных типов, делайте это в тех же заголовочных файлах, исходном коде (`.cc` файлы), и в том же пространстве имён. В результате операторы будут доступны там же, где и сами типы, а риск множественного определения минимален. По возможности избегайте определения операторов как шаблонов, т.к. придётся соответствовать любому набору шаблонных аргументов. Если вы определяете оператор, также определяйте "родственные" к нему. И позаботьтесь о согласованности выдаваемых ими результатов.

Желательно определять не изменяющие значения бинарные операторы как внешние функции (не-члены). Если же бинарный оператор объявлен членом класса, неявное преобразование может применяться к правому аргументу, но не к левому. А это может слегка расстроить программистов, если (например) код `a + b` будет компилироваться, а `b + a` - нет.

Для типа T, значения которого могут сравниваться на равенство, определите внешний (не член класса) оператор `operator==` и задокументируйте, когда два значения типа T будут считаться равными. Если есть единственное (и очевидное) понимание, когда значение `t1` типа T меньше другого значения `t2`, то допустимо определить `operator<=>`, который не должен противоречить `operator==`. Предпочтительно не перегружать другие операторы сравнения или упорядочивания.

Не нужно пытаться обойти переопределение операторов. Если требуется сравнение (или присваивание и функция вывода), то лучше определить == (или = и <<) вместо своих функций `Equals()`, `CopyFrom()` и `PrintTo()`. И наоборот: не нужно переопределять оператор только потому, что внешние библиотеки ожидают этого. Например, если тип данных нельзя упорядочить и хочется хранить его в `std::set`, то лучше сделайте пользовательскую функцию сравнения и не пользуйтесь оператором <.

Не переопределяйте &&, ||, , (запятая) или унарный &. Не переопределяйте `operator""`, т.е. не стоит вводить собственные литералы. Не используйте такие литералы, определённые во внешнем коде (включая стандартную библиотеку).

Дополнительная информация:
Преобразование типов описано в секции неявные преобразования. Оператор = расписан в конструкторе копий. Тема перегрузки << для работы со стримами освещена в потоках/streams. Также можно ознакомиться с правилами из раздела перегрузка функций, которые также подходят и для операторов.

Доступ к членам класса

Данные класса делайте всегда закрытыми `private`, кроме константа. Это упрощает использование инвариантов путём добавления простейших (часто - константных) функций доступа.

Допустимо (по техническим причинам) объявлять данные соответствующего тестового класса, определённого в `.cc` файле, как `protected` при использовании Google Test. Если тестовый класс определён вне используемого `.cc` файла, например в `.h` файле, то объявляйте члены данных как `private`.

Порядок объявления

Располагайте похожие объявления в одном месте, выносите секции `public` наверх.

Определение класса обычно начинается с секции `public:`, далее идёт `protected:` и затем `private:`. Пустые секции не указывайте.

Внутри каждой секции группируйте вместе подобные декларации. Предпочтителен следующий порядок:

1. Типы и псевдонимы (`typedef`, `using`, `enum`, вложенные структуры и классы, и `friend`-типы)
2. (Опционально, только для структур) не-`static` члены данных
3. Статические константы
4. Фабричные методы
5. Конструкторы и операторы присваивания
6. Деструкторы
7. Все остальные функции (`static` и не-`static` функции, `friend` функции)
8. Все другие члены данных (статические и не-статические)

Не размещайте в определении класса громоздкие определения методов. Обычно только тривиальные, очень короткие или критичные по производительности методы "встраиваются" в определение класса. См. также Встраиваемые функции.

Функции

Входные и Выходные Параметры

Результат выполнения C++ функции выдаётся через возвращаемое значение и иногда через выходные параметры (или входные/выходные (in/out) параметры).

Предпочтительно использовать именно возвращаемое значение (вместо выходных параметров): это формирует понятный код, производительность обычно не страдает (иногда она даже становится ещё лучше).

Предпочтительно возвращать результат по значению или, если не удаётся, по ссылке. Не возвращайте результат через обычный (сырой) указатель, за исключением случаев, когда указатель может принимать нулевое значение.

Параметры могут быть входными, выходными или и тем и другим (in/out). Обязательные входные параметры следует передавать либо как значения, либо как `const` ссылки. Обязательные выходные параметры или in/out лучше передавать как ссылки (которые не могут быть null). В целом, для опциональных параметров используйте либо `std::optional` для передачи входных параметров по значению, либо используйте `const` указатели (вместо ссылок в случае обязательных параметров). Используйте не-`const` указатели для представления опциональных выходных или in/out параметров.

Не объявляйте функции, которые требуют, чтобы ссылочный параметр сохранял валидность и после вызова функции. В некоторых случаях ссылочные параметры могут указывать на временные сущности, это может стать причиной ошибок с временем жизни объектов. Вместо этого постарайтесь убрать требования к жизненному циклу (например, скопировав параметр) или передавайте долгоживущие параметры через указатель и задокументируйте требования к времени жизни и непустому значению.

Когда объявляете параметры функции, то сначала указывайте входные параметры. Выходные параметры указывайте в конце. В частности, если нужно добавить новый входной параметр, то размещайте его перед выходными параметрами (не ставьте его в конец только потому, что он новый). Однако, порядок объявления параметров не является жёстким правилом. Иногда порядок in/out параметров диктуется сигнатурами соседних функций. В общем, правило есть, но иногда нужно проявить гибкость. Функции с переменным количеством аргументов тоже могут потребовать изменения порядка аргументов.

Пишите короткие функции

Желательно писать маленькие и сфокусированные функции.

Понятно, что в ряде случаев одна длинная функция лучше нескольких коротких, поэтому нельзя установить жёсткую границу. Однако, если длина функции превышает 40 строк, подумайте о возможности разбить её на части (без ущерба для логики программы).

Даже если длинная функция отлично работает сейчас, через месяц (или год) в неё могут добавить новый функционал. И это может привести к багам, которые трудно обнаружить. И наоборот, если функции короткие и простые, то другим людям проще их читать и модифицировать. Также короткие функции проще тестировать.

Когда вы работаете с чьим-то кодом, возможно встретятся длинные и запутанные функции. Не нужно боятся модифицировать существующий код: если работа с такой функцией становится запутанной, её сложно отлаживать или необходимо использовать куски этой функции в другом контексте, то попробуйте её разбить на несколько более маленьких кусочков, с которыми проще работать.

Перегрузка функций

Используйте перегрузку функций (включая конструкторы) только если по коду вызова можно понять, что будет вызываться, без детального изучения вариантов перегружаемых функций.

Определение:

Можно написать функцию, которая принимает аргумент `const std::string&`. И, например, перегрузить её другой функцией, принимающей `const char*` (хотя в этом случае лучше использовать `std::string_view`).

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

За:

Перегрузка может сделать код более интуитивным и понятным, позволяя создать функции с одинаковым именем, но принимающих разные аргументы. Такая возможность очень востребована при программировании шаблонов и может быть удобна при использовании паттерна Посетитель (Visitor).

Перегрузка с использованием различных вариаций константных (`const`) и ссылочных аргументов может сделать код более удобным и эффективным (См. TotW 148).

Против:

Когда функция перегружается только по типу аргументов (не по количеству), то необходимо понимать правила C++ по сопоставлению типов. Также перегрузка может быть запутанной, если наследуемый класс переопределяет только некоторые из вариантов перегружаемых функций исходного класса.

Вердикт:

Перегружайте функцию, если нет семантических различий между её вариантами. В этом случае допустимо изменять как типы аргументов, так и квалификаторы (const и др.) или количество аргументов. Делайте перегрузку так, чтобы не было нужды разбираться, какая именно версия функции будет вызвана (достаточно того, что **какая-нибудь** версия будет вызвана). Если же ещё все варианты функции можно будет описать одним комментарием в заголовочном файле, то (да, сделайте так) это признак хорошего дизайна программного интерфейса.

Аргументы по-умолчанию

Аргументы по-умолчанию допустимы в не-виртуальных функциях; также само значение не должно изменяться. Здесь ограничения аналогичны Перегрузке функций и, в целом, использование перегрузки является предпочтительным. Аргументы же по-умолчанию желательно использовать, только если преимущества от их использования (читабельность кода) перевешивают указанные ниже недостатки.

За:

Часто есть функция, в которой используются типовые значения, но иногда требуется для них задать другую величину. Параметры по-умолчанию предлагают удобный способ это сделать, не определяя несколько функций для редко используемых значений. По сравнению с перегрузкой функций, аргументы по-умолчанию позволяют написать более чистый код, с меньшим количеством дублирования и явным разделением аргументов на обязательные и опциональные.

Против:

Аргументы по-умолчанию по сути являются другим способом получить семантику перегруженных функций, поэтому все причины отказаться от перегрузки функций здесь также актуальны.

Значение для аргументов по-умолчанию в случае виртуальных функций определяется типом целевого объекта и нет никакой гарантии, что все объявления данной функции (в наследниках) содержат одно и то же значение.

Параметры по-умолчанию вычисляются для каждой точки вызова заново. И это может увеличить объём генерируемого кода. Однако, обычно ожидается, что значение по-умолчанию всегда одинаковое (т.е. не меняется от вызова к вызову).

Указатели на функции с аргументами по-умолчанию также могут сбивать с толку, т.к. сигнатура функции часто не соответствует форме вызова. И перегрузка функции позволяет решить эти проблемы.

Вердикт:

Аргументы по-умолчанию под запретом для виртуальных функций (где они могут работать некорректно) и для случаев, когда значение для аргумента может измениться. Например, не пишите такой код: `void f(int n = counter++);`.

В ряде случаев аргументы по-умолчанию могут сильно улучшить читабельность кода (даже с учётом вышеуказанных недостатков) и тогда их можно использовать. Если же есть сомнения, то используйте перегрузку функций.

Синтаксис указания возвращаемого типа в конце

Указывайте возвращаемый тип в конце, только если обычный синтаксис (возвращаемый тип вначале) неудобен или нечитабелен.

Определение:

В C++ есть две формы декларации функций. В старой форме возвращаемый тип указывается перед именем функции:

```
int foo(int x);
```

Новая форма использует `auto` перед именем функции и завершается возвращаемым типом, указываемым после списка аргументов. Например, предыдущую декларацию можно записать как:

```
auto foo(int x) -> int;
```

Основное отличие двух форм в том, что завершающий тип находится уже в области видимости самой функции. Конечно, для простых типов (например, `int`) большой разницы нет. Однако для сложных типов (использование типов, объявленных в области видимости класса; или объявленных через параметры функции) это может иметь значение.

За:

Возвращаемый тип, указанный в конце, является единственным способом явно его задать для лямбда-выражения. В ряде случаев компилятор может самостоятельно вывести возвращаемый тип лямбды, однако это возможно не всегда. И даже в случае умного компилятора, иногда требуется явно указать тип (для читабельности или др.)

Иногда намного проще и удобнее указать возвращаемый тип в конце, после списка параметров. Особенно в случаях, когда возвращаемый тип зависит от параметров шаблона. Например:

```
    template <typename T, typename U>
    auto add(T t, U u) -> decltype(t + u);
```

понятнее, чем:

```
    template <typename T, typename U>
    decltype(declval<T&>() + declval<U&>()) add(T t, U u);
```

Против:

Указание возвращаемого типа в конце является новым относительно синтаксисом, у которого нет аналогов в таких (C++-подобных) языках, как C и Java. Поэтому такая форма записи может показаться чуждой.

Существующие кодовые базы содержат огромное количество деклараций в обычном стиле. И они не будут переписываться под новый синтаксис. Поэтому на практике выбор такой: либо использовать только обычный стиль, либо смесь обычного и нового. Далее учитываем: унификация стиля это есть хорошо; единая версия синтаксиса более унифицирована, чем две разные версии. В общем, вы поняли.

Вердикт:

Используйте обычный (более старый) стиль декларации функции, когда возвращаемый тип указывается перед именем функции. Новую же форму (возвращаемый тип в конце) используйте либо по явной необходимости (лямбды), либо для улучшения читабельности кода. Причём последний вариант (читабельность) часто свидетельствует о чересчур сложных шаблонах, лучше их избегать.

Специфика Google

Есть различные трюки и средства, которые используются, чтобы сделать код на C++ более надёжным. И да, они могут отличаться от того, что используют в других компаниях.

Владение и умные указатели

Предпочтительно, чтобы динамически созданный объект имел одного (выделенного) владельца. Передачу такого "владения" желательно проводить через умные указатели.

Определение:

"Владение" это технология учёта, используемая для управления динамически выделенной памятью или другими ресурсами. Владелец динамической сущности (объекта) это объект или функция, которые ответственны за удаление сущности, когда она будет не нужна. Владение может быть распределённым, и в этом случае обычно последний оставшийся владелец отвечает за удаление. Даже если владение не является распределённым, этот механизм может использоваться, чтобы передать владение от одного объекта (или кода) другому.

"Умные" указатели это классы, которые функционируют как обычные указатели; например, в них перегружены операторы * и →. Некоторые типы умных указателей можно использовать для автоматического управления "владением": учёт владельцев, удаление объектов. `std::unique_ptr` это тип умного указателя, который реализует эксклюзивное владение динамически созданного объекта; объект удаляется в случае выхода из области видимости экземпляра `std::unique_ptr`. `std::unique_ptr` не может быть скопирован, однако его можно передать (*move*) другому `std::unique_ptr`, что фактически есть передача владения. `std::shared_ptr` это умный указатель, реализующий распределённое владение. `std::shared_ptr` можно копировать, при этом владение распределяется между всеми копиями. Управляемый объект удаляется, когда разрушается последняя копия `std::shared_ptr`.

За:

- Очень тяжело управлять динамической памятью без какого-либо механизма учёта владения.

- Передача владения может быть проще и быстрее, чем копирование объекта. Также не все объекты можно скопировать.
- Передача владения может быть проще, чем копирование указателя или использование ссылок, т.к. нет необходимости согласовывать жизненный цикл объекта между различными частями кода.
- Умные указатели могут улучшить читабельность кода, сделать его логику более понятной, самодокументируемой и непротиворечивой.
- Умные указатели могут исключить ручное управление владением, упростить код и избежать большого количества ошибок.
- Для константных (`const`) объектов распределённое владение может быть простой и эффективной альтернативой против полного копирования.

Против:

- Владение должно представляться и передаваться через указатели (либо умные, либо обычные). Семантика указателей сложнее работы со значениями, особенно в API: помимо владения необходимо беспокоиться о правильности используемых типов (aliasing), времени жизни, изменяемости объектов и т.д.
- Затраты по производительности при копировании значений часто завышены, поэтому прирост производительности при передаче владения (против простого копирования) в ряде случаев не может оправдать ухудшение читабельности и увеличение сложности кода.
- API управления владением могут накладывать свои ограничения на модель (порядок) управления памятью.
- При использовании умных указателей нет чёткого понимания где именно (в коде) будет производиться освобождение ресурсов.
- `std::unique_ptr` реализует передачу владения через семантику перемещения, которая является новой и может затруднить понимание кода.
- Распределённое владение с одной стороны позволяет аккуратно управлять владением, с другой стороны может усложнить архитектуру системы.
- Распределённое владение требует операций учёта во время выполнения, это может отразиться на производительности.
- В ряде случаев (например, при создании циклических ссылок) объекты с распределённым владением никогда не удалятся.
- Умные указатели не всегда могут заменить обычные указатели.

Вердикт:

Если необходима работа с динамической памятью, то предпочтительно чтобы код, выделяющий память, ею же и владел. Если другой код хочет получить доступ к этой памяти, то можно передать копию, указатель или ссылку (и всё это без передачи владения). Предпочтительно использовать `std::unique_ptr` для явной передачи владения. Например:

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr);
```

Без веской причины не создавайте код с распределённым владением. Как вариант, это может быть желание избежать "тяжелой" операции копирования, однако обязательно убедитесь, что выигрыш будет существенным и разделяемый объект - неизменяемый (т.е. `std::shared_ptr<const Foo>`). Если же требуется именно распределённое владение, то используйте `std::shared_ptr`.

Никогда не используйте `std::auto_ptr`. В качестве замены есть `std::unique_ptr`.

cpplint

Для проверки кода на ошибки стиля используйте `cpplint.py`.

`cpplint.py` - утилита, которая читает файл с кодом и определяет многие ошибки в стиле. Конечно, она не идеальна, иногда выдаёт ложно-положительные и ложно-отрицательные ошибки, однако это всё равно полезная утилита.

Иногда в проекте есть инструкция, откуда брать и как пользоваться `cpplint.py`. Если в вашем проекте такой нет, то можно просто скачать `cpplint.py`.

Ещё возможности C++

Rvalue-ссылки

Используйте rvalue-ссылки только в перечисленных ниже специальных случаях.

Определение:

Rvalue-ссылка является ссылочным типом, привязанным к временному объекту. По синтаксису похожа на обычную ссылку. Например, `void f(std::string&& s);` объявляет функцию с аргументом rvalue-ссылка на `std::string`.

Когда суффикс '&&' (без дополнительных квалификаторов) используется с шаблонным аргументом функции, то применяются специальные правила определения типа аргумента. И такая ссылка имеет название передающей (forwarding reference).

За:

- Определение конструктора перемещения (принимающего rvalue-ссылку на тип класса) даёт возможность переместить (move) класс вместо его копирования. Например, если `v1` это `std::vector<std::string>`, то код `auto v2(std::move(v1))` скорее всего выполнит несколько операций с указателями вместо копирования большого объёма данных. И в большинстве случаев это приведёт к существенному увеличению производительности кода.
- Rvalue-ссылки позволяют реализовать типы, которые можно перемещать, а не копировать. Это полезно для типов, которые нельзя копировать, но которые хочется передавать в функцию как аргумент, хранить в контейнере и т.д.

- Функция `std::move` необходима для эффективного использования некоторых типов стандартной библиотеки, таких как `std::unique_ptr`.
- Передающий ссылки, использующие объявление rvalue-ссылки, позволяют написать единую обёртку, перемещающую аргумент в другую функцию. И это одинаково работает вне зависимости от того, временный объект или нет, константный он или нет. Это и называется 'идеальная передача'.

Против:

- Rvalue-ссылки не все хорошо понимают. Особенности, наподобие схлопывания ссылок или специальных правил определения типа, всё сильно усложняют.
- Rvalue-ссылки часто используют неправильно, т.к. они интуитивно нелогичны: обычно ожидается, что аргумент будет иметь валидное состояние после вызова функции (и операции перемещения выполняться не будут).

Вердикт:

Не используйте rvalue-ссылки (и не применяйте && квалификатор в методах), за исключением следующего:

- Вы можете использовать их, чтобы объявить конструктор перемещения и перемещающий оператор присваивания (как описано в Копируемые и перемещаемые типы).
- Вы можете использовать их для определения &&-квалифицированных методов, которые по логике "поглощают" `*this`, оставляя его в состоянии неиспользуемого или пустого. Заметим, что это применимо только для квалификаторов методов (которые идут после закрывающих скобок сигнатуры функции); если вам хочется "поглотить" обычный параметр функции, то лучше передайте его по значению.
- Вы можете использовать передающие ссылки в связке с `std::forward`, чтобы получить "идеальную передачу" (perfect forwarding).
- Вы можете использовать их, чтобы объявить пары перегружаемых функций, одна с `Foo&&`, другая с `const Foo&`. Обычно предпочтительно передавать аргументы по значению, однако использование пары функций может дать лучшую производительность если, например, функции при определённых условиях не используют передаваемые значения. И не забывайте: если ради производительности пишется более сложный код, то проверьте и убедитесь, что это действительно помогает.

Дружественные сущности

В ряде случаев допустимо использовать классы и функции как `friend`.

Дружественные типы обычно определяются в том же файле, поэтому нет необходимости открывать другой файл, чтобы разобраться с использованием закрытых членов класса. Обычное использование `friend`: когда класс `FooBuilder` объявляется дружественным (friend) классу `Foo`, так что `FooBuilder` может корректно настроить внутреннее состояние `Foo` без необходимости открывать это состояние всем остальным. В ряде случаев удобно сделать класс unit-тестов дружественным исходному классу.

Дружественность расширяет (но не ломает) инкапсуляцию класса. В ряде случаев, когда требуется дать доступ к внутреннему состоянию только одному классу, лучше объявить его как friend, чем делать члены класса открытыми (`public`). Однако, остальные

классы должны взаимодействовать только через открытые функции.

Исключения (программные)

Мы НЕ используем исключения C++.

За:

- Исключения позволяют обрабатывать ситуации типа "это невозможно" не в месте возникновения ошибки, а на более верхнем уровне, позже. И всё это без копания в исходниках и составления таблиц с кодами ошибок.
- Исключения используются в большинстве современных языков программирования и их использование в C++ позволяет писать код, концептуально схожий с Python, Java и др.
- Некоторые библиотеки C++ используют исключения в своей работе и отказ от них может существенно усложнить интеграцию с этими библиотеками.
- Исключения являются единственным способом проинформировать о проблемах в конструкторе класса. Конечно, это можно обойти, используя фабричные методы или метод `Init()`, однако это потребует либо дополнительного выделения памяти или, соответственно, обработку специального "невалидного" состояния.
- Исключения очень удобны при использовании в тестировании (в тестовых фреймворках).

Против:

- Если используется `throw` в функции, то вы должны проверить всех, кто эту функцию вызывает. Должна быть базовая гарантия безопасности при исключениях. Или же код никогда не ловит исключения и тогда программа может внезапно завершиться. Например есть код, где `f()` вызывает `g()`, который вызывает `h()`. Если `h` выбрасывает исключение, которое отлавливает `f`, то `g` нужно писать аккуратно, иначе могут быть утечки ресурсов и т.д.
- Обычно использование исключений усложняет отслеживание последовательности выполнения кода. Например, функции могут завершаться в неожиданных местах. Это затрудняет поддержку кода и его отладку. Вы можете улучшить ситуацию, следуя (своим) правилам когда и где можно использовать исключения. Однако, очень желательно, чтобы и другие разработчики знали об этих правилах.
- Безопасное использование исключений требует использования дополнительных принципов кодирования, например, RAII. И их количество (принципов кодирования) может быть значительным. Например, чтобы разработчик не разбирался в тонкостях всей цепочки вызовов неизвестного ему кода, желательно выделить код сохранения данных в хранилище в отдельную фазу-фиксацию. Такое выделение может нести как плюсы, так и минусы (э-э-э, корпоративная политика смешивания кода в одну кучу для удобства обсфуркации?). В любом случае, использование исключений уменьшает количество доступных вариантов.
- Использование исключений ведёт к распуханию бинарного файла программы, увеличивает время компиляции (иногда только чуть-чуть) и вообще может привести к проблемам с адресным пространством.
- Само наличие исключений может провоцировать разработчиков выбрасывать их по поводу и без повода, даже когда это не нужно или может привести к сложностям в обработке. Например, если пользователь вводит неподходящий текст, это не должно приводить к выбрасыванию исключения. И вообще, если всё это расписывать, то никакого документа не хватит!

Вердикт:

Для новых проектов преимуществ от использования исключений обычно больше, чем недостатков. Однако, для уже существующего кода введение исключений может повлиять на весь код. Также могут возникнуть проблемы стыка нового кода и старого (без исключений) кода.

Т.к. большинство C++ кода в Google не использует исключений, то очень проблематично будет внедрять новый код, который будет генерировать исключения. Существующий код в Google не может корректно работать с исключениями, поэтому цена внедрения исключений намного выше, чем реализация любого нового проекта. Переписывание существующего кода под обработку исключений - это будет очень медленный процесс, с большим количеством ошибок. Поэтому лучше использовать альтернативу в виде возврата кода ошибки и assert-ов: это не так сложно.

Этот запрет также распространяется на возможности, такие как `std::exception_ptr` и `std::nested_exception`.

Однако, для кода под Windows есть послабления.

noexcept

Указывайте `noexcept`, если это корректно и будет полезно.

Определение:

Спецификатор `noexcept` используется для указания, что функция не будет выбрасывать исключения. Если же функция с таким спецификатором всё же выбросит исключение, то произойдёт крэш программы через `std::terminate`.

Также есть оператор `noexcept`. Он выполняет проверку: объявлено ли выражение как "не выбрасывающее исключений". Проверка проводится на этапе компиляции.

За:

- Спецификация конструктора перемещения как `noexcept` может улучшить производительность в ряде случаев, например `std::vector<T>::resize()` скорее переместит объект нежели скопирует его, если конструктор перемещения для типа T заявлен как `noexcept`.
- Указание `noexcept` для функций может разрешить дополнительную оптимизацию, например компилятор может не генерировать код по раскрутке стека. Конечно, это имеет смысл, если в целом исключения разрешены.

Против:

- Если проекты следуют данному руководству и в них исключения запрещены, то очень сложно проверить правильность спецификатора `noexcept`, если вообще применимо понятие правильности.
- Вы не можете безболезненно удалить ранее прописанный спецификатор `noexcept`, т.к. это аннулирует гарантию, на которую может полагаться другой код, причём полагаться весьма замысловатыми способами.

Вердикт:

Используйте `noexcept`, когда это может улучшить производительность и точно отражает семантику функции (если из функции вылетело исключение, то это явная, фатальная ошибка). Считается, что `noexcept` на конструкторе перемещения может существенно улучшить производительность, учитывайте это. Если вы рассчитывайте на значительный прирост производительности от применения `noexcept` для других функций, пожалуйста, сначала проконсультируйтесь с руководителем проекта.

Используйте безусловный `noexcept`, если исключения полностью запрещены (т.е. в типовом проекте C++ в Google). В ином случае, используйте спецификатор `noexcept` с условиями (желательно простыми), которые становятся false в тех редких случаях, когда функция может всё-таки выбросить исключение. Эти тесты могут пользоваться проверками на характеристики типов (например, `std::is_nothrow_move_constructible` для объектов создаваемых через конструктор перемещения) или аллокаторов (например, `absl::default_allocator_is_nothrow`). Отметим, что наиболее частая причина исключений - невозможность выделения памяти (и да, мы верим, что это не относится к конструкторам перемещения - они не должны выбрасывать исключений из-за ошибок выделения памяти) и есть много приложений, для которых эта ситуация означает фатальную ошибку, которую даже не имеет смысла обрабатывать. И даже в других, потенциально ошибочных, ситуациях рекомендуется делать упор на простоту интерфейса, нежели на поддержку всех сценариев обработки ошибок: например, вместо написания накрученного `noexcept` с зависимостью от внешней хэш-функции (выбрасывает она исключения или нет), можно просто задокументировать, что разрабатываемый компонент не поддерживает хэш-функции, которые выбрасывают исключения. И, после этого, использовать `noexcept` без всяких дополнительных условий.

Информация о типе во время выполнения (RTTI)

Не используйте информацию о типе во время выполнения (RTTI).

Определение:

RTTI позволяет запросить информацию о C++ классе объекта во время выполнения. Делается через `typeid` или `dynamic_cast`.

За:

Типовые альтернативы вместо RTTI (описано ниже) требуют модификации или редизайна иерархии классов, участвующих в запросах. Иногда такую модификацию очень тяжело сделать, или она нежелательна, особенно в коде, который уже используется в других проектах.

RTTI может быть полезен для юнит-тестов. Например, можно тестировать классы-фабрики на правильность сгенерированного типа. Также это полезно в выстраивании связей между объектами и их макетами (mock).

RTTI бывает полезно при работе с абстрактными объектами. Например:

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast<Derived*>(other);
  if (that == nullptr)
```

```
    return false;
  ...
}
```

Против:

Часто сам запрос типа объекта в процессе выполнения означает проблемы с дизайном приложения, показывает наличие изъянов в иерархии классов.

Бесконтрольное использование RTTI усложняет поддержку кода. Это может привести в развесистым условиям, которые зависят от типа объекта, которые рассыпаны по всему коду. И которые придётся досконально изучать если будет необходимость что-то изменить в этом коде.

Вердикт:

Использование RTTI может легко привести к злоупотреблениям, поэтому будьте аккуратны. Старайтесь ограничить использование RTTI только юнит-тестами. Рекомендуется отказаться от RTTI в новом коде. Если же требуется написать код, который ведёт себя по разному в зависимости от типа объекта, возможно следующие альтернативы будут более подходящими:

- Виртуальные методы. Это предпочтительный способ для выполнения различного кода в зависимости от типа объекта. И вся работа (код) делается в самом объекте.
- Если код должен находится вне объектов, то можно использовать подходы, аналогичные двойной диспетчеризации, например шаблон проектирования Посетитель/Visitor. Это позволит внешнему коду самому определять тип объектов (используя систему типов).

Когда логика программы гарантирует, что полученный указатель на базовый класс фактически есть указатель на определённый производный класс, тогда можно свободно использовать `dynamic_cast`. Правда, в этом случае лучше использовать `static_cast`.

Большое количество условий, основанных на типе объекта, есть показатель явных проблем в коде.

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

Подобный код может рассыпаться, когда добавляется новый дочерний класс в иерархию. И вообще, очень тяжело модифицировать большое количество разрозненных кусков кода в случае небольших изменений в свойствах или методах дочерних классов.

И пожалуйста, не изобретайте собственный велосипед на замену RTTI. Аргументы против собственного решения будут такие же (см. выше), да и разбираться в чужих велосипедах обычно сложнее.

Приведение типов / Casting

Рекомендуется использовать приведение типов в C++-стиле: `static_cast<float>(double_value)`. Также можно использовать инициализацию значением в скобках для преобразования арифметических типов: `int64_t y = int64_t{1} << 42`. Не используйте приведение вида `(int)x`, кроме приведения к `void`. Также допустимо использовать формат приведения вида `T(x)` только в случае, когда T это класс.

Определение:

В C++ приведение типов расширяется по сравнению с чистым C путём добавления операций приведения.

За:

Основная проблема с приведением типов в чистом C - неоднозначность операции. Например, одинаково записанная операция: `(int)3.5` и `(int)"hello"` сильно отличается по смыслу. Инициализация в скобках и операции в стиле C++ часто помогают избежать такой неоднозначности. Дополнительный плюс: операции приведения в стиле C++ легче искать по коду.

Против:

C++-стиль довольно громоздкий.

Вердикт:

В общем случае, не используйте приведение типов в стиле чистого C. Вместо этого используйте стиль C++, когда требуется явное преобразование типов.

- Используйте инициализацию в скобках для преобразования арифметических типов (`int64_t{x}`). Это самый безопасный способ, т.к. в других случаях (при потере информации при конвертации) код может не скомпилироваться. Плюс лаконичный синтаксис.
- Используйте `absl::implicit_cast` для безопасного приведения вверх по иерархии: например, приведение `Foo*` к `SuperclassOfFoo*`; или приведение `Foo*` к `const Foo*`. C++ обычно делает это автоматически, однако в некоторых ситуациях требуется явное приведение "вверх": например, использование оператора `?:`.
- Используйте `static_cast` как эквивалент преобразований в C-стиле или когда необходимо преобразовать указатель на дочерний класс в указатель на базовый или обратно: указатель на базовый класс преобразовать в указатель на дочерний. В последнем случае вы должны быть уверены, что объект является экземпляром дочернего класса.
- Используйте `const_cast` чтобы убрать квалификатор `const` (см. const).
- Используйте `reinterpret_cast` чтобы небезопасно преобразовать указатели к целым числам и обратно; или преобразовать к указателям другого типа, в том числе и `void*`. Используйте эту операцию только если вы знаете, что делаете и понимаете проблемы с выравниванием. Также, рассмотрите возможность разыменования указателя (без приведения типов) и использования `std::bit_cast` чтобы привести полученное значение к нужному типу.
- Используйте `std::bit_cast` для приведения "сырых" битов в другой тип такого же размера (например, если требуется интерпретировать `double` как `int64_t`).

Также может быть полезным раздел RTTI с описанием `dynamic_cast`.

Потоки / Streams

Используйте потоки в подходящих случаях, особенно если их использование упрощает код. Перегружайте операцию << только для типов-значений и выводите в поток собственно данные (доступные пользователю). Не выводите в поток внутренние переменные (инварианты и т.д.) и другие детали реализации.

Определение:

Потоки являются стандартной абстракцией для ввода/вывода в C++, (см. стандартный заголовочный файл `<iostream>`). Они часто используются в коде в Google, особенно для отладочного логирования и диагностики.

За:

Операторы << и >> реализуют форматированный ввод/вывод, они просты в понимании, портабельны, расширяемы и повторно используемы. Противоположность к ним - `printf`, который даже не поддерживает работу с `std::string`. Кроме того не работает с пользовательскими типа и с портабельностью там проблемы. Кроме того, `printf` вынуждает выбирать среди похожих версий одной функции и ориентироваться в десятках форматных символах.

Потоки обеспечивают хорошую поддержку консольного ввода/вывода через `std::cin`, `std::cout`, `std::cerr` и `std::clog`. Функции из C API тоже хорошо работают, однако могут требовать вручную буферизировать ввод.

Против:

- Форматирование потоков можно настроить через манипуляторы, изменяющие состояние потока. Учтите, применение манипуляторов сохраняется во времени и, как результат, поведение кода зависит от истории использования потока (или необходимо будет всё восстанавливать в известное состояние после каждого случая возможного применения манипуляторов). Кроме того, пользовательский код может (помимо модификации встроенных параметров) создавать собственные манипуляторы, добавлять переменные состояния и изменять поведение через функционал register_callback.
- Проблематично полностью контролировать вывод потока: смена настроек (см. выше), вывод мешанины из параметров и собственно данных, использование перегрузки операторов (и компилятор может выбрать не ту перегрузку, которая предполагалась) - всё это не добавляет управляемости.
- Формирование вывода посредством вызова цепочки операторов << затрудняет локализацию, т.к. при этом жёстко, прямо в коде, фиксируется порядок слов. И средства поддержки локализации лишаются ценности.
- Потоковый API может быть сложным, с тонкостями. И программисты должны уметь с ним работать, чтобы код был эффективным.
- Выбор правильной версии оператора << из большого количества аналогичных является для компилятора затратным. При частом их использовании в большой кодовой базе, время парсинга и анализа может отъедать до 20% времени.

Вердикт:

Используйте потоки только если они являются наилучшим решением. Обычно это варианты ввода/вывода в человекочитаемый формат, предназначенный для разработчиков, а не для конечного пользователя. Также не забывайте, что в проекте уже могут быть устоявшиеся методы ввода/вывода - старайтесь использовать их. В частности, библиотеки для логирования и диагностического вывода обычно предпочтительнее нежели `std::cerr` или `std::clog`. И вместо `std::stringstream` лучше использовать `absl/strings` или их эквивалент.

Не рекомендуется использовать потоки для ввода/вывода в случае обмена данными с конечными пользователями или где возможно нарушение формата или валидности данных. В таких случаях используйте подходящие (шаблонные) библиотеки, которые корректно обрабатывают интернационализацию, локализацию, проверяют корректность данных и формата.

Если потоки всё же используются, старайтесь избегать API работы с состояниями, кроме состояния ошибки. Т.е. не используйте `imbue()`, `xalloc()` и `register_callback()`. Рекомендуется использовать явные функции форматирования (такие, как `absl::StreamFormat()`) вместо манипуляторов или флагов форматирования для таких вещей как смена основания системы счисления, точности или набивка нулями до нужного размера чисел.

Перегружайте оператор << только для типа-значения с тем, чтобы оператор выводил человеко-читаемое представление. Не выводите в поток детали реализации или внутренние переменные. Если же требуется отладочная печать внутреннего состояния, то используйте обычные функции-методы (например, метод класса `DebugString()` - подходящий вариант).

Преинкремент и предекремент

Используйте префиксные формы (`++i`) инкремента и декремента; постфиксную форму используйте только при явной необходимости.

Определение:

Когда переменная инкрементируется (`++i`, `i++`) или декрементируется (`--i`, `i--`), а возвращаемое значение не используется, то необходимо чётко понимать: использовать префиксную форму (`++i`, `--i`) или постфиксную (`i++`, `i--`).

За:

Выражение с постфиксным инкрементом/декрементом оценивается для значения, *которое было до модификации*. В результате код можно сделать компактнее, однако сложнее для чтения. Префиксную форму обычно легче читать, и она часто более эффективна (как минимум такая же), т.к. не требуется создавать копию значения до выполнения операции.

Против:

Традиционно раньше в разработке (особенно на языке C) использовалась постфиксная форма (даже если возвращаемое значение не использовалось), особенно для циклов `for`.

Вердикт:

Используйте префиксную форму инкремента/декремента. Допустимо использовать постфиксную форму только для случаев, когда используется возвращаемое значение постфиксной операции.

Использование const

В API используйте `const` когда это имеет смысл. В ряде случаев `constexpr` будет лучшей альтернативой const.

Определение:

При объявлении переменных или параметров вначале может быть указано `const`, чтобы показать что переменные не изменяются (например, `const int foo`). Функции класса могут быть с квалификатором `const`, чтобы показать, что эта функция не изменяет состояние членов класса (например, `class Foo { int Bar(char c) const; };`).

За:

Позволяет легко понять, как использовать переменные. Компиляторам даёт возможность полнее контролировать типы и, теоретически, генерировать лучший код. Использование констант даёт дополнительную защиту (уверенность) в корректности кода: функции не могут модифицировать переменные, изменять состояние класса и, как результат, можно безопасно работать без локов в многопоточном окружении.

Против:

Использование `const` оно "заразное": если передаётся `const` переменная в функцию, то она должна в прототипе иметь указание на `const` (или придётся делать `const_cast`). И это может быть проблемой при вызове библиотечных функций.

Вердикт:

Настоятельно рекомендуется использовать `const` в API (параметры функций, методы, не-локальные переменные), где это имеет смысл. Такой подход даёт понятное (и верифицируемое компилятором) описание как можно модифицировать объекты. Чёткое разделение на модифицирующие (запись) и не-модифицирующие (только чтение) операции очень полезно, особенно для написания потокобезопасного кода. В частности:

- Если функция не меняет аргумент, переданный по ссылке или указателю, то он должен быть ссылкой на константу (`const T&`), либо указателем на константу (`const T*`).
- Если параметр передаётся по значению, то использование `const` не даёт никакого эффекта. Поэтому не рекомендуется объявлять константный параметр. См. также TotW #109.
- Старайтесь объявлять члены класса константными (`const`) если они не изменяют логическое состояние объекта (и не дают возможности другим что-то менять, например через возврат не-константной ссылки). В противном случае эти методы могут быть небезопасными в многопоточном окружении.

Использование `const` для локальных переменных отдаётся на усмотрение программиста: можно использовать, можно - нет.

Все `const` операции класса должны работать корректно при одновременном вызове нескольких функций. Если это не выполняется, то класс должен быть явно описан как "потоко-не-безопасный".

Местоположение const

Иногда используется форма `int const *foo` вместо `const int* foo`. Обосновывается это тем, что первая форма более логична: `const` следует за описываемым объектом. Однако, такая "логичность" имеет мало смысла (и обычно не применяется в коде с несколькими вложенными маркерами "указатель"), т.к. чаще всего есть только один `const` для базового значения. В таком случае нет необходимости специально заботиться о логичности. Размещение же `const` вначале делает код более читабельным и согласованным с английским языком: прилагательное (`const`) стоит перед существительным (`int`).

Так что расположение `const` вначале является предпочтительным. Однако, это не жёсткое условие и если остальной код в проекте использует другой порядок - следуйте за кодом!

Использование constexpr, constinit, consteval

Используйте `constexpr` для определения констант или чтобы сделать константную инициализацию. Используйте `constinit` для инициализации константой не-константной переменной.

Определение:

Переменные можно объявлять как `constexpr` для указания на константу, значение которой определяется во время компиляции или линковки. Также можно объявлять функции и конструкторы как `constexpr`, чтобы их можно было использовать для определения переменной с `constexpr`. Функции могут быть объявлены с `consteval`, чтобы ограничить их использование только временем компиляции.

За:

`constexpr` позволяет определять выражения с плавающей запятой (помимо литералов), использовать константы для пользовательских типов и вызовов функций.

Против:

Использование `constexpr` может вызвать проблемы с поддержкой кода (или миграцией), если константность нужно будет позже убрать. Текущие требования на допустимые вещи в `constexpr` функциях или конструкторах могут потребовать реализации дополнительных обходных путей в коде.

Вердикт:

`constexpr` позволяет определить неизменяемые части интерфейса. Используйте `constexpr` чтобы определить константы и функции для задания им значений. `consteval` может использоваться для кода, который не должен вызваться во время выполнения (в runtime). Не используйте `constexpr`, если это потребует усложнения кода. Не используйте `constexpr` или `consteval` чтобы сделать код "встраиваемым" (inlining).

Целочисленные типы

Если требуется встроенный целочисленный тип в C++ - используйте `int`. Если в программе требуется целочисленный тип другого размера, то используйте целочисленные типы с явным указанием длины из `<cstdint>`, такие как `int16_t`. Если используются значения, которые могут быть равны или превышать 2^31 (2 Гб), используйте 64-битный тип, такой как `int64_t`. При оценке размера не забудьте, что в `int` должен укладываться не только результат, но и промежуточные значения при вычислениях. И, если сомневаетесь, используйте тип подлиннее.

Определение:

C++ не определяет точный размер целочисленных типов, таких как `int`. В современных архитектурах обычно считается, что `short` содержит 16 битов, `int` - 32, `long` может быть как 32, так и 64 бита; `long long` содержит 64 бита. Однако, различные платформы могут задавать типы разного размера, особенно это касается `long`.

За:

Унификация в коде.

Против:

Размеры целочисленных типов в C++ могут изменяться в зависимости от компилятора и архитектуры.

Вердикт:

Заголовочный файл стандартной библиотеки `<cstdint>` определяет различные типы: `int16_t`, `uint32_t`, `int64_t` и т.д. Если требуются целочисленные типы фиксированного размера, то следует использовать именно эти (см. выше) типы. Не используйте `short`, `unsigned long long` и им подобные в качестве типов фиксированного размера. Для таких типов желательно опускать префикс `std::`, т.к. лишние 5 символов только замусорят код. Из целочисленных типов языка C можно использовать только `int`. Также, в соответствующих случаях, используйте псевдонимы стандартных типов, например `size_t` и `ptrdiff_t`.

Тип `int` используется очень часто, особенно для небольших значений, например как счётчики в циклах. Можете считать, что `int` содержит минимум 32 бита (но не больше). Если требуется 64 битный целочисленный тип, то используйте `int64_t` или `uint64_t`.

Для типа, который может хранить "большие значения" используйте `int64_t`.

Старайтесь не использовать беззнаковые числа (например, `uint32_t`). Допустимое применение беззнаковых чисел это использование битовых представлений или использование переполнения (по модулю 2^N) в расчётах. Отметим, что также не рекомендуется использовать беззнаковый тип чтобы указать на отсутствие отрицательных значений: в этом случае используйте assert-ы.

Если код возвращает размер контейнера, то убедитесь, что его тип (размера) является достаточным для любого возможного использования. И если сомневаетесь, используйте тип подлиннее.

Будьте внимательны при конвертировании целочисленных типов. Может появится неопределённое поведение (UB), ведущее к багам безопасности и другим проблемам.

Беззнаковые целые числа

Беззнаковые целые числа отлично подходят для работы с битовыми полями и модульной арифметики. Так сложилось, что стандарт C++ использует беззнаковые числа и для возврата размера контейнеров (хотя многие члены организации по стандартизации и считают это ошибкой; в любом случае, сейчас это уже не изменить). Ситуация, что беззнаковая арифметика по поведению является модульной (заворачивание значений при переполнении) и отличается от обычной (знаковой), не позволяет компилятору диагностировать большое количество ошибок. Фактически, такое поведение затрудняет оптимизацию.

С другой стороны, совместное использование беззнаковых и знаковых целых чисел создаёт ещё больше проблем. Лучшее решение: старайтесь использовать итераторы вместо указателей и явных размеров; не мешайте беззнаковые числа вместе со знаковыми; избегайте беззнаковых чисел (кроме работы с битовыми полями и для модульной арифметики); не используйте беззнаковые числа только чтобы показать, что переменная неотрицательная.

Совместимость с 64-бит

Написанный код должен быть совместим как с 64-битной, так и с 32-битной архитектурой. Особое внимание обращайте на печать в консоль, операции сравнения и выравнивание структур.

- Форматные символы `printf()` для некоторых целочисленных типов основываются на макроопределениях (например, `PRI` из `<cinttypes>`). И такой подход является непрактичным, нежелательным и т.д. Поэтому очень желательно не использовать функцию `printf` (и подобные) в своём коде, или даже переписать существующий код. Вместо неё можно использовать библиотеки, поддерживающие безопасное форматирование числовых значений, такие как `StrCat` или `Substitute` для простых преобразований, или `std::ostream`.

  К сожалению, макросы `PRI` являются единственным переносимым способом указать формат для типов данных с задаваемым размером (`int64_t`, `uint64_t`, `int32_t`, `uint32_t` и т.д.). По возможности не передавайте аргументы таких типов в функции,

основанные на `printf`. Исключение составляют типы, для которых есть свой выделенный модификатор длины, например `size_t` (`z`), `ptrdiff_t` (`t`) и `maxint_t` (`j`).

- Помните, что `sizeof(void*) != sizeof(int)`. Используйте `intptr_t` в случае, если требуется целочисленный тип размером, равным размеру указателя.
- Будьте аккуратны с выравниванием структур, особенно тех что записываются на диск. Обычно классы и структуры, в которых есть член типа `int64_t` или `uint64_t`, по умолчанию выравниваются на границу 8 байт в 64-битных системах. Если в коде есть подобные структуры, они сохраняются на диске и используются 32-битным и 64-битным кодом, то обязательно проверьте, что структуры упаковываются одинаково на обеих архитектурах. Большинство компиляторов позволяют задать выравнивание структур. В gcc можно использовать `__attribute__((packed))`. В MSVC - `#pragma pack()` и `__declspec(align())`.
- Используйте инициализацию в фигурных скобках, если требуется создать 64-битную константу. Например:

```
int64_t my_value{0x123456789};
uint64_t my_mask{uint64_t{3} << 48};
```

Макросы препроцессора

Избегайте определения макросов, особенно в заголовочных файлах. Вместо этого используйте встраиваемые функции, перечисления или переменные-константы. Если используете макросы, то в имени используйте префикс - название проекта. Не используйте макросы, чтобы переопределить или дополнить C++ API.

Использование макросов подразумевает, что программист видит один код, а компилятор - другой. Это может вызвать неожиданные последствия, особенно если макросы глобальные.

Ситуация может усугубиться, когда макросы используются для переопределения C++ или другого публичного API. При любых ошибках в использовании API потребуется разбираться в логике макросов; увеличивается время разбора кода инструментами рефакторинга или анализаторами. Как результат, использование макросов в таких случаях запрещено. Например, откажитесь от подобного кода:

```
class WOMBAT_TYPE(Foo) {
  // ...

 public:
  EXPAND_PUBLIC_WOMBAT_API(Foo)

  EXPAND_WOMBAT_COMPARISONS(Foo, ==, <)
};
```

К счастью, в C++ зависимость от макросов поменьше, чем в C. Вместо макросов для высокопроизводительного кода можно использовать встраиваемые функции. Для хранения констант есть `const` переменные. Чтобы для удобства "укоротить" длинное имя

переменной используйте ссылки. Вместо применения макросов для условной компиляции кода используйте ... лучше не используйте условную компиляцию (конечно же это не касается защиты от повторного включения заголовочных файлов через `#define`). Тем более условная компиляция затрудняет тестирование кода.

С другой стороны, есть приёмы кодирования, которые делаются только через макросы. Обычно это можно увидеть в низко-уровневых библиотеках. Также есть приёмы (преобразование в строку, объединение строк и т.д.), которые нельзя сделать средствами самого языка напрямую. В любом случае, перед использованием макросов попробуйте найти способ реализации без макросов. Если же необходимо использовать макросы для определения интерфейса то предварительно обязательно проконсультируйтесь с руководством.

Следующие правила позволят избежать ряда проблем с макросами. По возможности следуйте им:

- Не определяйте макросы в заголовочных (`.h`) файлах.
- Определяйте (`#define`) макросы по возможности ближе к месту первого использования. И когда макросы уже не используются, то делайте `#undef`.
- Не делайте `#undef` существующих макросов с последующим их переопределением своей версией. Вместо этого лучше придумайте для своих макросов уникальное имя и используйте его.
- Постарайтесь не использовать макросы, которые раскрываются в большие и неустойчивые конструкции C++. По крайней мере, документируйте такое поведение.
- Старайтесь не использовать ## для генерации имени функции/класса/переменной.

Настоятельно не рекомендуется экспортировать макросы из заголовочных файлов (т.е. определять макрос и не делать `#undef` его в конце заголовочного файла). Если макрос экспортируется из заголовочного файла, то он должен иметь глобальное уникальное имя. Как вариант, добавьте префикс с именем пространства имён проекта (заглавными буквами).

0 и nullptr/NULL

Используйте `nullptr` для указателей и `'\0'` для char-ов (не используйте `0` для этих целей).

Для указателей (адресов) используйте `nullptr`, это улучшает безопасность типов.

Используйте `'\0'` в качестве символа конца строки (пустого символа). Это улучшает читабельность кода.

sizeof

Рекомендуется использовать `sizeof(`***переменная***`)` вместо `sizeof(`***тип***`)`.

Используйте `sizeof(`***переменная***`)` если необходим размер определённой переменной. `sizeof(`***переменная***`)` будет возвращать корректное значение даже если в дальнейшем изменится тип переменной. `sizeof(`***тип***`)` можно использовать, когда код не работает с конкретной переменной, например в случае форматирования/разбора данных, где соответствующий тип C++ не подходит.

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

```
memset(&data, 0, sizeof(MyStruct)); // Плохо
```

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

Вывод типов, включая auto

Используйте вывод типов только если это сделает код более читабельным или более безопасным. Не используйте его только из-за неудобства написания полного типа.

Определение:

Есть ряд ситуаций, когда типы в C++ коде могут (или даже необходимо) быть выведены компилятором и это более предпочтительно, чем явно их прописывать:

вывод типов аргументов шаблонной функции
: Шаблонная функция может вызываться без указания явных шаблонных типов. Компилятор выводит эти типы из аргументов функции:

  ```
  template <typename T>
  void f(T t);

  f(0);  // Вызывается f<int>(0)
  ```

переменная с `auto` типом
: Декларация переменной может использовать `auto` вместо типа. Компилятор выводит тип из выражения инициализации, следуя правилам, аналогичным для шаблонной функции (во всяком случае, пока не используются фигурные скобки вместо круглых).

```
auto a = 42;  // a типа int
auto& b = a;  // b типа int&
auto c = b;   // c типа int
auto d{42};   // d типа int, а не std::initializer_list<int>
```

`auto` может использоваться совместно с `const`, в составе указателя или ссылки, и (начиная с C++17) для задания аргумента шаблона, не являющегося типом. Изредка можно увидеть использование `decltype(auto)` вместо `auto` и в этом случае выводимый тип является результатом применения `decltype` к переданному выражению инициализации.

вывод типа возвращаемого значения функции

`auto` (и `decltype(auto)`) можно использовать для указания возвращаемого значения функции. Компилятор выводит тип возвращаемого значения из выражения `return` в теле функции, следуя тем же правилам, что и при объявлении переменной:

```
auto f() { return 0; }  // Возвращаемый f тип - int
```

Возвращаемый тип лямбды может быть выведен аналогичным способом (хотя это делается при отсутствии возвращаемого типа, а не в случае использования `auto`). Синтаксис указания возвращаемого типа в конце для функций также использует `auto`, но это не вывод типа, скорее альтернативный синтаксис для явно указанного возвращаемого типа.

общие (generic) лямбды

В описании лямбды можно использовать `auto` в качестве одного или нескольких типов параметров. Как результат, оператор вызова лямбды будет шаблонной функцией вместо обычной, с отдельными параметрами шаблона по одному на каждый `auto`:

```
// Сортируем `vec` по убыванию
std::sort(vec.begin(), vec.end(), [](auto lhs, auto rhs) { return lhs > rhs; });
```

инициализация захватываемых переменных лямбды

В лямбде в секции захвата можно явно прописать новые переменные, инициализированные значением:

```
[x = 42, y = "foo"] { ... }  // тип x - int, y - const char*
```

Синтаксис не позволяет указать тип новой переменной, он выводится аналогично `auto` переменным.

вывод аргументов шаблонного класса

См. соответствующий раздел.

структурная привязка

При объявлении кортежей, структур или массивов с использованием `auto` можно указать имена отдельных элементов вместо имени полного объекта. Эти имена называются "структурная привязка", а декларация - соответственно "декларация структурной привязки". Синтаксис не позволяет задать тип ни полного объекта, ни отдельных имён:

```
auto [iter, success] = my_map.insert({key, value});
if (!success) {
```

```
  iter->second = value;
}
```

`auto` можно использовать с квалификаторами `const`, & и &&. Отметим, что эти квалификаторы формально применяются к анонимному кортежу/структуре/массиву, а не к отдельным привязкам. Правила определения конечного типа привязок довольно запутанные, однако в большинстве случаев всё довольно логично. Можно только отметить, что тип привязки обычно не может быть ссылочным, даже если в декларации указана ссылка (хотя поведение всё равно может быть как у ссылки).

В приведённом выше описании не указаны многие детали, для дополнительной информации используйте приведённые ссылки.

За:

- Название типов в C++ может быть длинным и громоздким, особенно при использовании шаблонов и пространств имён.
- Когда название типа C++ повторяется несколько раз внутри небольшого куска кода или декларации, это повторение не улучшает читабельность.
- В ряде случаев вывод типов безопаснее, т.к. позволяет избежать случайных копирований или преобразований типов.

Против:

При явном указании типов код C++ становится более ясным и понятным, особенно если вывод типов опирается на информацию из совершенно другой части кода. В выражении наподобие:

```
auto foo = x.add_foo(); // Плохо. Что есть foo?
auto i = y.Find(key);
```

может быть неочевидно какие типы выводятся для переменных, особенно если y не является хорошо известным типом или объявлен намного раньше по коду.

Необходимо разбираться, выдаётся ли ссылка при выводе типа, производится ли копирование (особенно если оно не предполагалось).

Если выводимые типы используется как часть интерфейса, то любые незначительные изменения в выражениях могут привести к радикальным изменениям API.

Вердикт:

Основное правило: используйте вывод типов только если это сделает код более ясным и безопасным. Не используйте вывод типов только, чтобы избежать неудобств при написании явного типа. При оценке понятности кода учитывайте, что читатели кода могут быть в другой команде и не знакомы с этим проектом. Поэтому, хотя явные типы могут считаться понятным, очевидным и избыточными для одних, они могут содержать полезную информацию для других. Например, можно полагать, что возвращаемый тип `make_unique<Foo>()` очевиден. Однако, в случае `MyWidgetFactory()` лучше считать по-другому.

Эти принципы применяются для всех видов вывода типов. Однако, существуют тонкости, описанные ниже.

Вывод аргументов шаблонной функции

Вывод аргументов шаблонной функции это практически всегда хорошо. Это стандартный и ожидаемый способ работы с шаблонными функциями, т.к. вызов такой функции напоминает работу с обычной (но перегруженной для различных аргументов) функцией. И шаблонные функции желательно проектировать так, чтобы вывод аргументов был корректный и безопасный, или функция не должна компилироваться.

Вывод типов локальных переменных

В случае локальных переменных можно использовать вывод типов, чтобы сделать код более простым, убрав очевидную или нежелательную информацию о типах, и читатель может сконцентрироваться на важных частях кода. Сравните примеры кода:

```
std::unique_ptr<WidgetWithBellsAndWhistles> widget =
    std::make_unique<WidgetWithBellsAndWhistles>(arg1, arg2);
absl::flat_hash_map<std::string,
                    std::unique_ptr<WidgetWithBellsAndWhistles>>::const_iterator
    it = my_map_.find(key);
std::array<int, 6> numbers = {4, 8, 15, 16, 23, 42};
```

```
auto widget = std::make_unique<WidgetWithBellsAndWhistles>(arg1, arg2);
auto it = my_map_.find(key);
std::array numbers = {4, 8, 15, 16, 23, 42};
```

Типы иногда содержат смесь как полезной, так и формальной информации. Например в примере выше очевидно, что `it` это итератор. Вообще в большинстве аналогичных случаев информация о контейнере и даже типе ключа не особо полезна, однако тип значения обычно важен. В этом случае можно определить локальную переменную с явно указанным (полезным для читателя) типом:

```
if (auto it = my_map_.find(key); it =/= my_map_.end()) {
  WidgetWithBellsAndWhistles& widget = *it->second;
  // Do stuff with `widget`
}
```

Если тип является шаблонным, у которого параметры неинформативны, но тип самого шаблона является важным, то можно использовать вывод аргументов шаблонного класса чтобы избавиться от формального (ненужного) кода. Отметим что случаи, когда

это является полезным, довольно редки. Также учтите, что при использовании вывода аргументов шаблонного класса желательно следовать специальным рекомендациям.

Не используйте `decltype(auto)` при наличии более простых альтернатив, т.к. результат использования не всегда легко предсказуем.

Вывод типа возвращаемого значения

Используйте вывод типа возвращаемого значения (как для функция, так и для лямбд), только если тело функции содержит небольшое количество `return` и сам код небольшой. В противном случае читатель не сможет понять возвращаемый тип без его исследования. Также применяйте такой вывод типов только для функций или лямбд с очень маленькой областью видимости, т.к. функции не могут установить границы абстракции: сама реализация *формирует* интерфейс. В частности, публичные функции в заголовочных файлах никогда не должны использовать вывод типа возвращаемого значения.

Вывод типов параметров

Параметры с `auto` в лямбдах должны использовать с осторожностью, т.к. реальный тип определяется кодом лямбды, а не её объявлением. И, конечно, явное указание типа обычно более понятно, за исключением случаев, когда лямбда определяется рядом с местом её использования (можно одновременно видеть и определение лямбды и её вызов) или лямбда передаётся в настолько известный интерфейс, что используемые аргументы очевидны (например, см. вызов `std::sort` выше).

Инициализация переменных захвата лямбды

При инициализации переменных захвата предпочтительны специальные рекомендации, которые в целом подменяют общие правила для использования вывода типов.

Структурные привязки

В отличие от других форм вывода типов, структурные привязки могут дать дополнительную информацию за счёт правильного именования элементов полного объекта. Т.е. декларация структурной привязки может улучшить читабельность кода по сравнению с использованием явного типа (даже если `auto` не рекомендуется). Структурные привязки хорошо подходят при работе с парами или

кортежами (см. пример использования `insert` выше), потому что в последних нет "говорящих" названий полей. С другой стороны, в целом не рекомендуется использовать пары и кортежи, пока внешний API, наподобие `insert`, явно этого не потребует.

Если объектом привязки является структура, иногда может быть полезно указать имена, лучше подходящие для данного кода. Однако учитывайте, что такие имена могут быть менее понятны читателям кода, чем штатные имена полей. Рекомендуется использовать комментарии для указания имён полей, если они отличаются от имён привязок. Используйте синтаксис, аналогичный комментариям к параметрам функций:

```
auto [/*field_name1=*/ bound_name1, /*field_name2=*/ bound_name2] = ...
```

Также, как и с параметрами функций, комментарии могут помочь внешним инструментам определить ошибки в порядке указания полей.

Вывод аргументов шаблонного класса

Используйте вывод аргументов только для тех шаблонных классов, что явно поддерживают это.

Определение:

Вывод аргументов шаблонного класса (CTAD) проявляется, когда переменная декларируется с типом шаблона, но без указания аргументов (даже без угловых скобок):

```
std::array a = {1, 2, 3};  // Тип `a`: std::array<int, 3>
```

Компилятор выводит аргументы из выражения инициализации, используя "гайд" для шаблонов, который может быть явный и неявный.

Явный гайд походит на декларацию функции с возвращаемым типом в конце, только без `auto` вначале, и имя функции есть имя шаблона. Например, ранее приведённый пример опирается на следующий гайд для `std::array`:

```
namespace std {
template <class T, class... U>
array(T, U...) -> std::array<T, 1 + sizeof...(U)>;
}
```

Конструкторы в основном определении шаблона (т.е. не в специализации) также неявно определяют "гайд".

Когда объявляется переменная, использующая CTAD, компилятор выбирает "гайд" на основе правил выбора (разрешения) перегруженного конструктора, и возвращаемый гайдом тип становится типом переменной.

За:

CTAD иногда может уменьшить количество формального кода.

Против:

Неявные "гайды", получаемые из конструкторов, могут реализовывать нежелательное или даже неправильное поведение. Эта проблема может часто проявляться для конструкторов, написанных до появления CTAD в C++17, т.к. авторы кода просто не знали о тех проблемах, которые вызовет CTAD. И далее, добавление явных "гайдов" для исправления проблемы может поломать любой существующий код, который использует неявные гайды.

У CTAD есть много недостатков, аналогичных недостаткам `auto`, т.к. оба механизма выводят (полный или частичный) тип переменной на основе инициализации. CTAD выдаёт больше информации, чем `auto`, однако всё равно не содержит явного указания, если информация была пропущена.

Вердикт:

Не используйте CTAD на шаблонных классах, пока не будет поддержки механизма и обеспечен хотя бы один явный "гайд" (предполагается, что в пространстве имён `std` всё поддерживается). Желательно, чтобы недопустимое использование CTAD приводило к предупреждениями компилятора.

В любом случае, использование CTAD должно следовать общим правилам при выводе типов.

Назначенная Инициализация

Используйте назначенную инициализацию только в C++20-совместимом формате.

Определение:

Назначенная инициализация это синтаксис, который позволяет инициализировать агрегат (простая структура данных, POD) по явному указанию имён полей:

```
struct Point {
  float x = 0.0;
  float y = 0.0;
  float z = 0.0;
};

Point p = {
  .x = 1.0,
  .y = 2.0,
```

```
    // z будет 0.0
};
```

Явно перечисленные поля будут инициализированы указанными значениями, остальные инициализируются аналогично традиционному выражению инициализации агрегата, наподобие `Point{1.0, 2.0}`.

За:

Назначенная инициализация может добавить удобства и понятности выражениям с агрегатами, особенно для структур с более запутанным, чем пример с `Point` выше, порядком полей.

Против:

Хотя назначенная инициализация уже давно является частью стандарта C и поддерживается компиляторами C++ как расширение, она (прим. официально) не поддерживается в версиях C++ перед C++20.

Правила в C++ стандарте более строгие, чем в C и расширениях компиляторов. Требуется указывать назначенную инициализацию в том же порядке, как поля появляются в определении структуры. Поэтому, в примере выше, для C++20 допустимо инициализировать `x` и затем `z`, а вот `y` и затем `x` - нет.

Вердикт:

Используйте назначенную инициализацию только в форме, совместимой со стандартом C++20: инициализаторы располагаются в том же порядке, что и соответствующие поля в объявлении структуры.

Лямбды

Используйте лямбды в подходящих случаях. Желательно использовать явный захват переменных, если лямбда будет выполнена вне текущей области видимости.

Определение:

Лямбды это лаконичный и удобный способ создания объектов - анонимных функций. Особенно они полезны, когда нужно передавать функцию как аргумент. Например:

```
std::sort(v.begin(), v.end(), [](int x, int y) {
  return Weight(x) < Weight(y);
});
```

Лямбды также позволяют захватывать переменный из текущей области видимости либо (явно) по имени, либо (неявно) через захват по-умолчанию. Явный захват предписывает перечислить все требуемые переменные: либо как значения, либо как ссылки:

```
int weight = 3;
int sum = 0;
// Захват `weight` по значению и `sum` по ссылке.
std::for_each(v.begin(), v.end(), [weight, &sum](int x) {
  sum += weight * x;
});
```

Захват по-умолчанию применяется ко всем переменным, используемым в теле лямбды, в том числе и к `this` (если используются члены класса):

```
const std::vector<int> lookup_table = ...;
std::vector<int> indices = ...;
// Захват `lookup_table` по ссылке, сортировка `indices` по значению
// ассоциированных элементов из `lookup_table`.
std::sort(indices.begin(), indices.end(), [&](int a, int b) {
  return lookup_table[a] < lookup_table[b];
});
```

Захват переменной может быть также с инициализатором, что можно использовать для перемещения (move) переменных по значению или для других случаев, не подпадающих под обычный захват по значению или ссылке:

```
std::unique_ptr<Foo> foo = ...;
[foo = std::move(foo)] () {
  ...
}
```

Такой тип захвата (часто называемый init или generalized) нужен скорее не для собственно "захвата" переменных (или даже имён) из текущей области видимости. Этот синтаксис нужен для определения членов объекта лямбды:

```
[foo = std::vector<int>({1, 2, 3})] () {
  ...
}
```

И тип такой переменной с инициализатором выводится согласно правилам, аналогичным использованию `auto`.

За:

- Лямбды это очень удобный и лаконичный способ определения объектов-функций (например, для передачи в алгоритмы STL), что может улучшить читабельность.
- Правильное использование захвата по-умолчанию может устранить избыточность и выявить важные исключения при захвате.

- Лямбды, `std::function` и `std::bind` можно использовать совместно как механизм обратного вызова общего назначения. Упрощается написание кода, принимающего/использующего функции как аргументы.

Против:

- Захват переменных может быть источником ошибок с недействительными указателями, особенно если лямбда выходит за текущую область видимости.
- Захват по умолчанию может вводить в заблуждение, т.к. он не защищает от ошибок недействительных указателей. Захват указателя на объект по значению не приводит к созданию копии самого объекта. При этом часто возникают те же особенности жизненного цикла, что и при захвате по ссылке. Учтите, что `this` также захватывается по значению и это может привести к проблемам, т.к. он часто используется неявно.
- Захват фактически объявляет новые переменные (вне зависимости от наличия инициализатора), хотя это отличается от обычного синтаксиса C++. В частности, нет указания типа переменной, даже в виде `auto` (хотя захват с инициализацией может сделать это неявно через приведение типов), и это отличается от типовой декларации.
- Захват с инициализацией использует вывод типов, и подвержен тем же проблемам, что и `auto`, плюс сам синтаксис не содержит указания, что происходит вывод типов.
- В ряде случаев использование лямбд может сильно затруднить понимание кода (очень длинные функции) или потребовать дополнительного контроля.

Вердикт:

- Используйте лямбды в подходящих случаях. Применяйте форматирование, описанное ниже.
- Рекомендуется явно указывать захват переменных, если лямбда может выйти за текущую область видимости. Например, вместо:

```
{
  Foo foo;
  ...
  executor->Schedule([&] { Frobnicate(foo); })
  ...
}
// ПЛОХО! При беглом просмотре можно упустить, что лямбда использует
// ссылку на foo и this (если Frobnicate является членом класса).
// Если лямбда вызывается после возврата из текущей функции, то
// это приведёт к проблемам, т.к. foo и другие объекты могут быть
// уже разрушены.
```

лучше написать:

```
{
  Foo foo;
  ...
  executor->Schedule([&foo] { Frobnicate(foo); })
```

```
  ...
}
// ЛУЧШЕ - Компилятор выдаст ошибку, если Frobnicate является методом класса.
// Также явно указано, что foo захватывается по ссылке.
```

- Используйте захват по-умолчанию по ссылке (`[&]`), только если жизненный цикл лямбды явно короче чем у любой захватываемой переменной.
- Используйте захват по-умолчанию по значению (`[=]`), только как средство привязки нескольких переменных для короткой лямбды, где состав захвата очевиден и не будет неявного захвата `this`. (Подразумевается, что лямбда, которая используется в не-статической функции класса и внутри себя ссылается на нестатические члены класса, должна захватывать `this` либо явно, либо через `[&]`.) Не рекомендуется писать лямбды с объёмным и сложным кодом, используя захват по-умолчанию по значению.
- Используйте захват только существующих переменных из текущей области видимости. Не используйте захват с инициализаторами, чтобы ввести новые имена или подменить существующие. Вместо этого можно объявить обычную новую переменную и передать её в лямбду, или вместо лямбды определить отдельную полноценную функцию.
- Вопросы по указанию параметров и возвращаемому типу рассмотрены в разделе Вывод типов.

Метапрограммирование на шаблонах

Не используйте сложные/запутанные шаблоны в коде.

Определение:

Метапрограммирование на шаблонах это семейство техник, использующих возможности по инстанцированию шаблона в C++, которое Тьюринг-полное, для выполнения вычислений на этапе компиляции.

За:

Метапрограммирование позволяет создавать очень гибкие интерфейсы, у которых высокая производительность и отличная типобезопасность. Например, Google Test, `std::tuple`, `std::function` и Boost.Spirit были бы невозможны без таких средств.

Против:

Техники метапрограммирования часто непонятны всем, кроме экспертов языка. Код, использующий шаблоны запутанными/сложными способами, часто нечитабелен, его сложно отлаживать и поддерживать.

Метапрограммирование часто приводит к появлению очень скудной и непонятной информации об ошибках компиляции: даже если сам интерфейс простой, то его сложная реализация всё равно проявляется, когда пользователь делает что-то неправильно.

Метапрограммирование усложняет проведение рефакторинга, затрудняя работу инструментов. Во-первых, код шаблона раскрывается в различных контекстах и трудно проверить, что в каждом из них код остаётся корректным. Во-вторых, ряд инструментов работает уже с AST (прим.: абстрактное синтаксическое дерево), которое описывает структуру кода только после раскрытия шаблонов. И в этом

случае может быть тяжело обнаруженные проблемы отобразить на исходные конструкции в коде и определить, что требуется переписать.

Вердикт:

Метапрограммирование в ряде случаев позволяет создать понятные и простые в использовании интерфейсы (которые были бы сложнее в ином случае), однако есть соблазн всё сделать чересчур заумным. Поэтому его лучше использовать в небольшом количестве низкоуровневых компонентов, чтобы сложность поддержки компенсировалась полезностью и широтой применения.

Дважды подумайте перед тем, как использовать метапрограммирование или другие сложные техники на шаблонах: может ли средний программист в команде понять и поддерживать такой код (особенно после переключения с другого проекта); сможет ли не-C++ программист (или другой случайный читатель) понять сообщения об ошибках или отладить выполнение функции, которую он вызывает. Если используются рекурсивное инстанцирование шаблонов, список типов, метафункции, шаблоны выражений или используется SFINAE или трюк с `sizeof` для разрешения перегрузки функции - скорее всего вы зашли слишком далеко.

Если используется метапрограммирование, то готовьтесь приложить усилия для минимизации сложности, а также её изоляции. По возможности скрывайте код с метапрограммированием внутри реализации, чтобы сделать пользовательские заголовочные файлы более читабельными. Код с метапрограммированием должен быть очень хорошо откомментирован: следует подробно задокументировать, как использовать код и на что будет похож "сгенерированный" результат. Обратите особое внимание на сообщения об ошибках, которые выдаёт компилятор, когда делается что-то ошибочное. Учтите, что сообщения об ошибках являются частью вашего пользовательского интерфейса и код следует доработать так, чтобы сообщения об ошибках были понятными для пользователя и объясняли, что нужно делать для их исправления.

Концепты и Ограничения

Используйте концепты редко. В общем, концепты и ограничения следует использовать только там, где использовались шаблоны до C++20. Не добавляйте концепты в заголовочные файлы, за исключением внутренних файлов внутри библиотек. Не определяйте концепты, которые не поддерживаются компилятором. Предпочтительно использовать ограничения вместо метапрограммирования, также избегайте синтаксиса вида `template<Concept T>`. Вместо этого используйте подход `requires(Concept<T>)`.

Определение:

Ключевое слово `concept` определяет новый механизм для задания требований к параметрам шаблонов, такие как признаки (traits) для типов или спецификации интерфейсов. Ключевое слово `requires` даёт механизмы для назначения анонимных ограничений на шаблоны и проверки соответствия этим ограничениям во время компиляции. Концепты и Ограничения часто используются совместно, но допустимо и независимое применение.

За:

- Концепты позволяют компиляторам выдавать более осмысленные сообщения об ошибках при использовании шаблонов, уменьшить путаницу и значительно улучшить разработку кода.
- Концепты могут уменьшить количество одинакового кода, необходимого для определения и использования ограничений на этапе компиляции, что часто улучшает результирующий код.

- Ограничения предоставляют некоторые возможности, которые трудно реализовать через шаблоны или подход SFINAE.

Против:

- Также, как и с шаблонами, концепты могут значительно усложнить код, сделать его трудным для понимания.
- Синтаксис концептов может сбить с толку, так как описание концептов похоже на использование классов.
- Концепты, особенно в API, увеличивают связанность кода, усложняют его изменение.
- Концепты и Ограничения могут повторять логику, которая уже есть в теле функций, и это приводит к дублированию кода и осложняет его сопровождение.
- Концепты искажают смысл изначальных контрактов. Концепты становятся самостоятельными сущностями, которые используются в различных местах и могут обособленно изменяться. Это может привести к расхождению заявленных и предполагаемых требований с течением времени.
- Концепты и Ограничения влияют новыми и неочевидными способами на механизм разрешения перегрузок.
- Как и в случае с SFINAE, Ограничения затрудняют масштабный рефакторинг кода.

Вердикт:

Предопределённые концепты стандартной библиотеки более предпочтительны для признаков (traits) типов, даже при наличии аналогов (например, если `std::is_integral_v` использовался до C++20, то следует использовать `std::integral` в коде под C++20). Аналогично, более предпочтительным является современный синтаксис ограничений через `requires(`***`Condition`***`)`. Избегайте устаревших конструкций метапрограммирования (таких, как `std::enable_if<`***`Condition`***`>`) и синтаксиса `template<`***`Concept`*** `T>`.

Не переделывайте вручную любые существующие концепты или признаки (traits). Например, используйте `requires(std::default_initializable<T>)` вместо `requires(requires { T v; })` или подобных.

Декларации новых (пользовательских) концептов (`concept`) следует избегать, за исключением случаев внутреннего использования в библиотеках без выставления во внешний API. В целом, не используйте концепты и ограничения в тех местах, где вы не стали бы применять эквивалентные шаблоны под C++17.

Не определяйте концепты, дублирующие тело функции, или накладывают ограничения, которые незначительны, очевидны при чтении кода функции или понятны из сообщений об ошибках. Например, избегайте такого кода:

```
template <typename T>     // Плохо - избыточность и мало пользы
concept Addable = std::copyable<T> && requires(T a, T b) { a + b; };
template <Addable T>
T Add(T x, T y, T z) { return x + y + z; }
```

Вместо этого оставьте код шаблона как он есть. Добавляйте концепты только если очевидно значительное улучшение понятности именно этого кода, например через сообщения об ошибках об вложенных или неочевидных требованиях.

Концепты должны быть статически верифицируемы компилятором. Не используйте концепты, основная польза от которых будет в семантических (или других неисполняемых) ограничениях. Требования, которые не обрабатываются во время компиляции, должны оформляться другими средствами, такими как комментарии, assert-ы или тесты.

Boost

Используйте только одобренные библиотеки из коллекции Boost.

Определение:

Boost это популярная коллекция проверенных, бесплатных и открытых библиотек C++.

За:

В целом код Boost является высококачественным, портируемым и во многом дополняется стандартную библиотеку C++, например, в таких областях как свойства типов или улучшенные связыватели (binder).

Против:

Некоторые библиотеки Boost поощряют создание кода, который ухудшает читабельность: используется метапрограммирование или другие продвинутые техники на шаблонах, а также чрезмерно "функциональный" стиль.

Вердикт:

Чтобы читабельность кода оставалась высокой для всех, кто осуществляет его поддержку, разрешены к использованию только некоторые библиотеки из коллекции Boost. В настоящее время это:

- Call Traits из `boost/call_traits.hpp`
- Compressed Pair из `boost/compressed_pair.hpp`
- Boost Graph Library (BGL) из `boost/graph`, за исключением сериализации (`adj_list_serialize.hpp`) и параллельных/распределённых алгоритмов и структур данных (`boost/graph/parallel/*` и `boost/graph/distributed/*`).
- Property Map из `boost/property_map`, за исключением параллельных/распределённых (`boost/property_map/parallel/*`).
- Iterator из `boost/iterator`
- Часть Polygon, которая работает с построением диаграмм Вороного и не зависит от остальной части Polygon: `boost/polygon/voronoi_builder.hpp`, `boost/polygon/voronoi_diagram.hpp`, and `boost/polygon/voronoi_geometry_type.hpp`
- Bimap из `boost/bimap`
- Statistical Distributions and Functions из `boost/math/distributions`
- Special Functions из `boost/math/special_functions`
- Root Finding & Minimization Functions из `boost/math/tools`
- Multi-index из `boost/multi_index`
- Heap из `boost/heap`
- flat-контейнеры библиотеки Container: `boost/container/flat_map` и `boost/container/flat_set`
- Intrusive из `boost/intrusive`
- Библиотека `boost/sort`

- Preprocessor из `boost/preprocessor`

В настоящее время прорабатывается вопрос о добавлении других библиотек Boost в этот список, так что он может в будущем дополняться.

Ещё возможности C++

Некоторые расширения современного C++, также как и Boost, содержат код с плохой читабельностью, с удалённой добавочной информацией о типах, с использованием метапрограммирования. Другие же расширения дублируют существующий функционал, что может привести к путанице и дополнительной конвертации кода.

Вердикт:

Настоятельно не рекомендуется использовать следующие возможности C++:

- Рациональные числа (относящиеся к времени компиляции) (`<ratio>`), т.к. за интерфейсом может стоять сложный шаблон.
- Заголовочные файлы `<cfenv>` и `<fenv.h>`, т.к. многие компиляторы не поддерживают корректную работу этого функционала.
- Заголовочный файл `<filesystem>`, который недостаточно протестирован, и подвержен уязвимостям в безопасности.

Нестандартные расширения

Нестандартные расширения C++ не рекомендуется использовать, пока явно не указано обратное.

Определение:

Компиляторы поддерживают различные расширения, не являющиеся частью стандартного C++. Например, это GCC `__attribute__`, внутренние (intrinsic) функции (`__builtin_prefetch` или SIMD), `#pragma`, ассемблерные вставки, `__COUNTER__`, `__PRETTY_FUNCTION__`, составные выражения (`foo = ({ int x; Bar(&x); x })`, массивы переменной длины и `alloca()`, а также "оператор Элвис" `a ?: b`.

За:

- Нестандартные расширения могут предоставить полезные возможности, которых нет в стандартном C++.
- Сделать указания компилятору по оптимизации производительности кода можно только используя расширения.

Против:

- Нестандартные расширения работают не на всех компиляторах, поэтому их использование ухудшает переносимость кода.

- Даже если расширения поддерживаются всему требуемыми компиляторами, их спецификация может быть не полной и возможны различия в поведении у разных компиляторов.
- Для понимания кода нужно знать нестандартные расширения в дополнение к языку программирования.
- Нестандартные расширения потребуют дополнительных усилий при портировании на другие архитектуры.

Вердикт:

Не используйте нестандартные расширения. Можно использовать портируемые обёртки кода, которые реализованы с использованием нестандартных расширений и (желательно) содержатся в одном заголовочном файле (portability header).

Псевдонимы/Alias

Публичные псевдонимы предназначены для использования с API и должны быть хорошо документированы.

Определение:

Есть несколько способов для создания имён, являющихся псевдонимами для других сущностей:

```
using Bar = Foo;
typedef Foo Bar;  // Но предпочтительнее использовать `using` в C++ коде.
using ::other_namespace::Foo;
using enum MyEnumType;  // Создаются псевдонимы для всех констант из MyEnumType.
```

При написании нового кода рекомендуется использовать `using`, а не `typedef`. Это обеспечивает более согласованный синтаксис с остальным C++ кодом и поддерживает работу с шаблонами.

Аналогично другим декларациям, псевдонимы, введённые в заголовочном файле, обычно являются частью публичного API этого файла. Исключения касаются случаев, когда псевдонимы объявлены внутри определения функции, private секции класса или явно отмеченном внутреннем пространстве имён. Такие псевдонимы, а также введённые в .cc файлах, являются "деталями реализации" (т.к. клиентский код к ним не обращается) и не подпадают под действие этих правил.

За:

- Псевдонимы могут улучшить читабельность кода, сокращая длинные и сложные имена.
- Псевдонимы могут уменьшить дублирование: например, сделав именование типа, часто использующегося в API, в дальнейшем это *может* облегчить изменение этого типа на другой.

Против:

- Псевдонимы, размещённые в заголовочном файле (где клиентский код может обращаться к ним), увеличивают количество сущностей в этом файле API и его сложность.
- Клиентский код может полагаться на особенности публичных псевдонимов, и это усложняет внесение изменений.

- Есть соблазн создавать публичные псевдонимы, которые будут использоваться только во внутренней реализации. Такой подход может влиять на сам API и усложнит его поддержку.
- Псевдонимы увеличивают риск создания дубликата для другого имени.
- Псевдонимы могут ухудшить читабельность кода, если хорошо известным сущностям будут давать незнакомые имена.
- Псевдонимы типов могут создать "ненадёжное" соглашение по API: то ли псевдоним всегда будет идентичным указанному типу (API не изменится, тип можно использовать любым способом); то ли тип у псевдонима может поменяться (тогда рекомендуется использовать только небольшую часть возможностей).

Вердикт:

Не вводите псевдонимы в публичный API только для облегчения написания кода в реализации; псевдоним должен прежде всего быть полезен для клиентского кода.

Определяя публичный псевдоним, обязательно опишите назначение нового имени, будет ли псевдоним всегда соответствовать текущему типу или будет более ограниченная совместимость. С одной стороны, это позволит узнать, является ли псевдоним заменителем типа или необходимо следовать более специфическим правилам. С другой стороны, это может дать определённую свободу при изменении псевдонима в дальнейшем.

Не объявляйте публичные псевдонимы на пространства имён в своём API. (См. также Пространство имён).

Например, следующие псевдонимы описывают способ их использования в клиентском коде:

```
namespace mynamespace {
// Используется для хранения измерений. DataPoint может меняться с Bar* на другой
// внутренний тип, его следует трактовать как абстрактный указатель.
using DataPoint = ::foo::Bar*;

// Набор измерений. Добавлен для удобства пользователя.
using TimeSeries = std::unordered_set<DataPoint, std::hash<DataPoint>, DataPointComparator>;
}  // namespace mynamespace
```

Приведённые ниже псевдонимы не документируют способы использования и часть из них вообще не предназначения для использования в клиентском коде:

```
namespace mynamespace {
// Плохо: непонятно, как это использовать.
using DataPoint = ::foo::Bar*;
using ::std::unordered_set;  // Плохо: это только для внутреннего удобства
using ::std::hash;           // Плохо: это только для внутреннего удобства
typedef unordered_set<DataPoint, hash<DataPoint>, DataPointComparator> TimeSeries;
}  // namespace mynamespace
```

Однако, локальные псевдонимы очень удобны внутри определения функций, `private` секций классов, внутренних пространств имён и в `.cc` файлах:

```
// В .cc файле
using ::foo::Bar;
```

Оператор Switch

Если в качестве сравниваемого выражения используется не-перечисление, то оператор switch всегда должен иметь секцию `default` (в случае же использования перечисления компилятор будет выдавать предупреждения, если какие-то из значений не обрабатываются). Если при этом секция default не должна выполняться, то оформляйте это как ошибочное поведение. Например:

```
switch (var) {
  case 0: {
    ...
    break;
  }
  case 1: {
    ...
    break;
  }
  default: {
    LOG(FATAL) << "Invalid value in switch statement: " << var;
  }
}
```

Если объединяется код для нескольких меток (из одного case проваливаемся в другой), то это должно быть помечено атрибутом `[[fallthrough]];`. Причём `[[fallthrough]];` ставится в месте "проваливания" в следующий case. Исключением является случай указания нескольких меток для одного общего блока кода - здесь никакой аннотации не требуется.

```
switch (x) {
  case 41:  // Аннотация не требуется.
  case 43:
    if (dont_be_picky) {
      // Используйте атрибут вместо (или совместно) с комментарием-описанием.
      [[fallthrough]];
```

```
    } else {
      CloseButNoCigar();
      break;
    }
  case 42:
    DoSomethingSpecial();
    [[fallthrough]];
  default:
    DoSomethingGeneric();
    break;
}
```

Инклюзивный Язык

Во всём коде, включая имена и комментарии, используйте инклюзивный язык и избегайте терминов, которые другие программисты могут посчитать невежливыми или оскорбительными, даже если слова имеют вполне нейтральный смысл (такие, как "master" и "slave", "blacklist" и "whitelist", "redline"). Аналогично, используйте нейтральный язык: например, используйте "they"/"them"/"their" для указания на людей без уточнения пола (в том числе и для одного человека), и "it"/"its" для программного обеспечения, компьютеров и т.п. Исключением могут быть случаи указания на конкретного человека (с использованием соответствующих местоимений).

Именование

Основные правила стиля кодирования приходятся на именование. Вид имени сразу же (без поиска объявления) говорит нам что это: тип, переменная, функция, константа, макрос и т.д.

Правила именования могут быть произвольными, однако важна их согласованность, и правилам нужно следовать.

Общие принципы именования

Используйте имена, который будут понятны даже людям из другой команды.

Имя должно говорить о цели или применимости объекта. Не экономьте на длине имени, лучше более длинное и более понятное (даже новичкам) имя. Поменьше аббревиатур, особенно если они незнакомы вне проекта. Используйте только известные аббревиатуры (Википедия о них знает?). Не сокращайте слова. В целом, длина имени должна соответствовать размеру области видимости. Например, n - подходящее имя внутри функции в 5 строк, однако при описании класса это может быть коротковато.

```
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int n = 0;  // Чёткий смысл для небольшой области видимости
    for (const auto& foo : foos) {
      ...
      ++n;
    }
    return n;
  }
  void DoSomethingImportant() {
    std::string fqdn = ...;  // Известная аббревиатура полного доменного имени
  }
 private:
  const int kMaxAllowedConnections = ...;  // Чёткий смысл для контекста
};
```

```
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int total_number_of_foo_errors = 0;  // Слишком подробное имя для короткой функции
    for (int foo_index = 0; foo_index < foos.size(); ++foo_index) {  // Лучше использовать `i`
      ...
      ++total_number_of_foo_errors;
    }
    return total_number_of_foo_errors;
  }
  void DoSomethingImportant() {
    int cstmr_id = ...;  // Сокращённое слово (удалены буквы)
  }
 private:
  const int kNum = ...;  // Для целого класса очень нечёткое имя
};
```

Отметим, что типовые имена также допустимы: `i` для итератора или счётчика, `T` для параметра шаблона.

В дальнейшем при описании правил "word" / "слово" это всё, что пишется на английском без пробелов. Сюда относятся аббревиатуры, такие как акронимы и буквенные сокращения. Для имён, написанных в смешанном стиле (иногда называемый "верблюжий регистр" или "стиль паскаля"), в котором первая буква каждого слова пишется заглавной, предпочтительно аббревиатуры писать как одно слово, т.е. `StartRpc()` лучше, чем `StartRPC()`.

Параметры шаблона также следуют правилам своих категорий: type names / имена типов для типов, variable names / имена переменных для переменных.

Имена файлов

Имена файлов должны быть записаны только строчными буквами, для разделения можно использовать подчёркивание (`_`) или дефис (`-`). Используйте тот разделитель, который используется в проекте. Если единого подхода нет - используйте "_".

Примеры подходящих имён:

- `my_useful_class.cc`
- `my-useful-class.cc`
- `myusefulclass.cc`
- `myusefulclass_test.cc // _unittest and _regtest are deprecated.`

C++ файлы должны заканчиваться на `.cc`, заголовочные - на `.h`. Файлы, включаемые как текст должны заканчиваться на `.inc` (см. также секцию Независимые заголовочные файлы).

Не используйте имена, уже существующие в `/usr/include`, такие как `db.h`.

Старайтесь давать файлам специфичные имена. Например, `http_server_logs.h` лучше чем `logs.h`. Когда файлы используются парами, лучше давать им одинаковые имена. Например, `foo_bar.h` и `foo_bar.cc` (и содержат класс `FooBar`).

Имена типов

Имена типов начинаются с прописной буквы, каждое новое слово также начинается с прописной буквы. Подчёркивания не используются: `MyExcitingClass`, `MyExcitingEnum`.

Имена всех типов - классов, структур, псевдонимов, перечислений, параметров шаблонов - именуются в одинаковом стиле. Имена типов начинаются с прописной буквы, каждое новое слово также начинается с прописной буквы. Подчёркивания не используются.

Например:

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

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

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

// enums
enum class UrlTableError { ...
```

Имена Концептов

Концепты именуются по тем же правилам, что и Имена Типов.

Имена переменных

Имена переменных (включая параметры функций) и членов данных пишутся змеиным стилем (`snake_case` - строчными буквами с подчёркиванием между словами). Члены данных классов (но не структур) дополняются подчёркиванием в конце имени. Например: `a_local_variable`, `a_struct_data_member`, `a_class_data_member_`.

Имена обычных переменных

Например:

```
std::string table_name;  // OK - змеиный стиль
```

```
std::string tableName;   // Плохо - смешанный стиль
```

Члены данных класса

Члены данных классов, статические и нестатические, именуются как обычные переменные с добавлением подчёркивания в конце.

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

Члены данных структуры

Члены данных структуры, статические и нестатические, именуются как обычные переменные. К ним не добавляется символ подчёркивания в конце.

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

См. также Структуры vs Классы, где описано когда использовать структуры, когда классы.

Имена констант

Объекты объявляются как `constexpr` или `const`, чтобы значение не менялось в процессе выполнения. Имена таких объектов начинаются с символа "k", далее идёт имя в смешанном стиле (прописные и строчные буквы). Подчёркивание может быть использовано в редких случаях когда прописные буквы не могут использоваться для разделения. Например:

```
const int kDaysInAWeek = 7;
const int kAndroid8_0_0 = 24;  // Android 8.0.0
```

Все аналогичные константные объекты со статическим типом хранилища (т.е. статические или глобальные, подробнее тут: Storage Duration) именуются также. Это относится и к шаблонам, в которых возможно получение различных значений в зависимости от инстанциирования. Соглашение является необязательным для переменных в других типах хранилища (например, автоматические переменные); в противном случае применяются обычные правила именования переменных. Например:

```
void ComputeFoo(absl::string_view suffix) {
  // Допустим любой из вариантов
  const absl::string_view kPrefix = "prefix";
  const absl::string_view prefix = "prefix";
  ...
}
```

```
void ComputeFoo(absl::string_view suffix) {
  // Плохо - разные аргументы при вызове ComputeFoo создают kCombined с разными значениями
  const std::string kCombined = absl::StrCat(kPrefix, suffix);
  ...
}
```

Имена функций

Обычные функции именуются в смешанном стиле (прописные и строчные буквы); функции доступа к переменным (accessor и mutator) должны иметь стиль, похожий на целевую переменную.

Обычно имя функции начинается с прописной буквы и каждое слово в имени пишется с прописной буквы.

```
AddTableEntry()
DeleteUrl()
OpenFileOrDie()
```

(Аналогичные правила применяются для констант в области класса или пространства имён (namespace) которые представляют собой часть API и должны выглядеть как функции (и то, что они не функции - некритично))

Accessor-ы и mutator-ы (функции get и set) могут именоваться наподобие соответствующих переменных. Они часто соответствуют реальным переменным-членам, однако это не обязательно. Например, `int count()` и `void set_count(int count)`.

Именование пространства имён (namespace)

Пространство имён называйте строчными буквами, слова разделяйте подчёркиваниями. Пространство имён верхнего уровня формируйте на основе имени проекта. Избегайте коллизий ваших имён и других, хорошо известных, пространств имён.

Пространство имён верхнего уровня - это обычно название проекта или команды (которая делала код). Код должен располагаться в директории (или поддиректории) с именем, соответствующим пространству имён.

Не забывайте правило не использовать аббревиатуры - к пространствам имён это также применимо. Коду внутри вряд ли потребуется упоминание пространства имён, поэтому аббревиатуры - это лишнее.

Избегайте использовать для вложенных пространств имён известные названия. Коллизии между именами могут привести к сюрпризам при сборке. В частности, не создавайте вложенных пространств имён с именем `std`. Рекомендуются уникальные идентификаторы проекта (`websearch::index`, `websearch::index_util`) вместо небезопасных к коллизиям `websearch::util`. Также старайтесь избегать излишней вложенности пространств имён (TotW #130).

Для `internal / внутренних` пространств имён коллизии могут возникать при добавлении другого кода (внутренние хелперы имеют свойство повторяться у разных команд). В этом случае хорошо помогает использование имени файла для именования пространства имён. (`websearch::index::frobber_internal` для использования в `frobber.h`).

Имена перечислений

Перечисления (как с ограничениями на область видимости (scoped), так и без (unscoped)) должны именоваться как константы, но не как макросы. Т.е. используйте `kEnumName`, но не `ENUM_NAME`.

```
enum class UrlTableError {
  kOk = 0,
  kOutOfMemory,
  kMalformedInput,
};
```

```
enum class AlternateUrlTableError { // Плохо (прим. пер)
  OK = 0,
  OUT_OF_MEMORY = 1,
  MALFORMED_INPUT = 2,
};
```

Вплоть до января 2009 года стиль именования значений перечисления был как у макросов. Это создавало проблемы дублирования имён макросов и значений перечислений. Таким образом, сейчас применение стиля констант становится логичным. Новый код должен использовать для именования стиль констант.

Имена макросов

Вы ведь не собираетесь определять макросы? На всякий случай (если собираетесь), они должны выглядеть так: `MY_MACRO_THAT_SCARES_SMALL_CHILDREN_AND_ADULTS_ALIKE`.

Пожалуйста прочтите как определять макросы; Обычно, макросы *не* должны использоваться. Однако, если они вам абсолютно необходимы, именуйте их прописными буквами с символами подчёркивания, указывая название проекта как префикс.

```
#define MYPROJECT_ROUND(x) ...
```

Исключения из правил именования

Если вам нужно именовать что-то, имеющее аналоги в существующем C или C++ коде, то следуйте используемому в коде стилю.

`bigopen()`
: имя функции, образованное от `open()`

`uint`
: похож на стандартный тип

`bigpos`
: `struct` или `class`, образованный от `pos`

`sparse_hash_map`
: STL-подобная сущность; следуйте стилю STL

`LONGLONG_MAX`
: константа, такая же как `INT_MAX`

Комментарии

Комментарии являются обязательными для кода (если вы планируете его читать). Следующие правила описывают, что вы должны комментировать и как. Но помните: хотя комментарии очень важны, идеальный код сам себя документирует. Использование "говорящих" имён для типов и переменных намного лучше, чем непонятные имена, которые потом требуется расписывать в комментариях.

Комментируйте код с учётом его следующих читателей: программистов, которым потребуется разбираться в вашем коде. Учтите, что следующим читателем можете стать вы!

Стиль комментариев

Используйте либо `//` либо `/* */`, пока не нарушается единообразие.

Вы можете использовать либо `//` либо `/* */`, однако `//` *намного* предпочтительнее. Однако, всегда согласовывайте ваш стиль комментариев с уже существующим кодом.

Комментарии в шапке файла

В начало каждого файла вставляйте шапку с лицензией.

Если файл с кодом (например `.h` файл) объявляет несколько абстракций для "внешнего" использования (общие функции, связанные классы и т.п.), то добавляйте комментарий в шапку, описывающий эту коллекцию абстракций. Должно быть достаточно деталей, чтобы другие авторы могли понять, что соответствует и не соответствует этой коллекции. Не размещайте в этом комментарии детальное описание отдельных абстракций.

Например, если пишется комментарий в шапке файла `frobber.h`, то не нужно добавлять такие комментарии в `frobber.cc` или `frobber_test.cc`. С другой стороны, если вы запрограммировали коллекцию классов в файле `registered_objects.cc`, у которого нет ассоциированного заголовочного файла, то нужно добавить комментарий в шапку файла `registered_objects.cc`.

Правовая информация и список авторов

Каждый файл должен содержать информацию о лицензии. Формат описания зависит от лицензии, используемой в проекте. У каждой лицензии (Apache 2.0, BSD, LGPL, GPL, др.) могут быть свои требования к оформлению.

Если вы делаете значительные изменения в файле, подумайте над удалением прежнего списка авторов. Обновлённые файлы могут уже не содержать упоминание об авторских правах и список авторов.

Комментарии классов или структур

Каждое (неочевидное) объявление класса или структуры должно сопровождаться комментарием, для чего эта сущность и как ей пользоваться.

```
// Перебор содержимого GargantuanTable.
// Пример:
//    std::unique_ptr<GargantuanTableIterator> iter = table->NewIterator();
//    for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//      process(iter->key(), iter->value());
//    }
class GargantuanTableIterator {
  ...
};
```

Комментарии классов

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.

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

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

Комментарии функции

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

Объявление функции

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 (например, функции получения значений переменных класса). Закрытые методы и функции, объявленные в `.cc` файле, следуют общим правилам. Комментарии должны соответствовать теме *об этой функции* и должны начинаться с глагольной фразы: например, "Opens the file"/"Открывает файл" более предпочтительно, чем "Open the file"/"Открыть файл". Обычно такие комментарии не описывают, как функция выполняет свою задачу: для этого используются комментарии в определении функции.

В комментарии к объявлению функции обратите внимание на следующее:

- Что подаётся на вход функции, что возвращается в результате. Если имена аргументов функции записаны в 'апострофах', то это может облегчить работу генераторам документации.
- Для функции-члена класса: сохраняет ли экземпляр ссылки или указатели на аргументы вне функции. Такое поведение является обычным делом для указателей/ссылок аргументов в конструкторах.
- Возможно ли использование null для значения (каждого) аргумента-указателя; и что произойдёт в этом случае.
- Для каждого выходного или in/out аргумента укажите, что будет с исходным значением в аргументе (например, значение будет дополнено или переписано).
- Влияние работы функции на производительность.

Пример:

```
// Возвращает итератор по таблице, указывающий на первый элемент, который
// лексически больше или равен start_word. Если такого элемента не существует, то
// возвращается пустой указатель. Клиент не должен использовать итератор,
// если исходный объект GargantuanTable был разрушен.
//
// Этот метод эквивалентен следующему:
//     std::unique_ptr<Iterator> iter = table→NewIterator();
//     iter→Seek(start_word);
//     return iter;
std::unique_ptr<Iterator> GetIterator(absl::string_view start_word) const;
```

Однако не стоит разжёвывать очевидные вещи.

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.

Комментируя конструкторы и деструкторы, учитывайте, что читатель кода знает их назначение. Поэтому комментарий типа "разрушает этот объект" - бестолковый. Можете описывать, что конструктор делает с аргументами (например, изменение владения на указатели) или какие именно операции по очистке делает деструктор. Если всё и так понятно - ничего не комментируйте. Вообще, обычно деструкторы не имеют комментариев (при объявлении).

Определение функций

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 более лучшие альтернативы). Можете описать принципы синхронизации кусков кода (здесь блокируем, а здесь рыбу заворачиваем).

Отметим что 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.

Комментарии к переменным

По хорошему, имя переменной должно сразу говорить что это и зачем. Однако, в некоторых случаях требуются дополнительные комментарии.

Член данных класса

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.

С другой стороны, полезными будут 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 num_total_entries_;
```

Глобальные переменные

Ко всем глобальным переменным следует писать комментарий о том, что это за переменные, их назначение и (если не очевидно) почему они должны быть глобальными. Например:

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

Комментарии к реализации

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

Описательные комментарии

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

Комментарии к аргументам функций

Когда назначение аргумента функции неочевидно, подумайте о следующих вариантах:

- Если аргумент есть фиксированное значение (literal constant) и это значение используется в разных блоках кода (и подразумевается, что это значение суть одно и тоже), вам следует создать константу и явно использовать её.
- Возможно следует заменить аргумент типа `bool` на перечисление (`enum`). Это сделает аргумент само-определяющим.
- Для функций, использующих несколько конфигурационных опций в аргументах, можно создать отдельный класс (или структуру), объединяющий все опции. И передавать в функцию экземпляр этого класса. Такой подход имеет несколько преимуществ: опции

обозначаются именами, что объясняет их назначение. Уменьшается количество аргументов в функции - код легче писать и читать. И если вам понадобится добавить ещё опций, менять сам вызов функции не придётся.
- Вместо сложных выражений в аргументах используйте промежуточную переменную, которой присвойте выражение.
- В крайнем случае пишите комментарии в месте вызова для прояснения назначения аргументов.

Рассмотрим примеры:

```
// И какое назначение аргументов?
const DecimalNumber product = CalculateProduct(values, 7, false, nullptr);
```

Попробуем причесать код:

```
ProductOptions options;
options.set_precision_decimals(7);
options.set_use_cache(ProductOptions::kDontUseCache);
const DecimalNumber product =
    CalculateProduct(values, options, /*completion_callback=*/nullptr);
```

Что делать не нужно

Не объясняйте очевидное. В частности, не нужно объяснять вещи, очевидные для человека, знающего C++. Вместо этого, можно описать *зачем* этот код делает так (или вообще сделайте код само-описываемым).

Сравним:

```
// Ищём элемент в векторе.  ⟵ Плохо: очевидно же!
if (std::find(v.begin(), v.end(), element) != v.end()) {
  Process(element);
}
```

С этим:

```
// Обрабатывает (Process) "element" пока есть хоть один
if (std::find(v.begin(), v.end(), element) != v.end()) {
  Process(element);
}
```

Само-описывающий код вообще не нуждается в комментариях. Комментарий на код выше может быть вообще очевидным (и не нужным):

```
if (!IsAlreadyProcessed(element)) {
  Process(element);
}
```

Пунктуация, орфография и грамматика

Обращайте внимание на пунктуацию, орфографию и грамматику: намного проще читать грамотно написанные комментарии.

Комментарии должны быть написаны как рассказ: с правильной расстановкой прописных букв и знаков препинания. В большинстве случаев законченные предложения легче понимаются, нежели обрывки фраз. Короткие комментарии, такого типа как построчные, могут быть менее формальными, но всё равно должны следовать общему стилю.

Хотя излишнее внимание код-ревьюера к использованию запятых вместо точек с запятой может слегка раздражать, очень важно поддерживать высокий уровень читабельности и понятности кода. Правильная пунктуация, орфография и грамматика этому очень сильно способствует.

Комментарии TODO

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

Такой комментарий должен включать строку `TODO` (все буквы прописные), за ней ID дефекта, имя, адрес e-mail или другая информация для идентификации разработчика и сущности проблемы, для которой написан `TODO`.

```
// TODO: bug 12345678 - Удалить после истечения срока совместимости 4 квартал 2047 года
// TODO: example.com/my-design-doc - Вручную подправьте этот код при последующей модификации
// TODO(bug 12345678): Обновите этот список после удаления сервиса Foo
// TODO(John): Используйте "\*" как оператор объединения
```

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").

Форматирование

Стиль кодирования и форматирования являются вещью произвольной, однако проект намного легче управляется, если все следуют одному стилю. Хотя кто-то может не соглашаться со всеми правилами (или пользоваться тем, чем привыкли), очень важно чтобы все следовали единым правилам, чтобы легко читать и понимать чужой код.

Для корректного форматирования мы создали файл настроек для emacs.

Длина строк

Желательно ограничивать длину строк кода 80-ю символами.

Это правило немного спорное, однако масса уже существующего кода придерживается этого принципа, и мы также поддерживаем его.

За:

Приверженцы правила утверждают, что строки длиннее не нужны, а постоянно подгонять размеры окон утомительно. Кроме того, некоторые размещают окна с кодом рядом друг с другом и не могут произвольно увеличивать ширину окон. При этом ширина в 80 символов - исторический стандарт, зачем его менять?.

Против:

Другая сторона утверждает, что длинные строки могут улучшить читабельность кода. 80 символов - пережиток мейнфреймов 1960-х. Современные экраны вполне могут показывать более длинные строки.

Вердикт:

80 символов - максимум.

Строка может превышать предел в 80 символов если:

- комментарий при разделении потеряет в понятности или лёгкости копирования. Например, комментарий с примером команды или URL-ссылкой, длиннее 80 символов.
- строковый литерал, который (легко) не получится перенести на 80 колонке. Это могут быть URI или другие семантически-критичные куски, литерал с текстом на некоем языке, многострочный литерал, где критично расположение переносов строки (например, информационные сообщения). В этих случаях разделение литерала может ухудшить читабельность, усложнить поиск, заблокировать возможность нажать на интернет-ссылку и т.д. За исключением тестового кода, такие литералы должны помещаться в пространство имён и располагаться ближе к началу файла. Если инструментарий, наподобие CLang-Format, не распознаёт неразделяемое содержимое, заблокируйте функции инструментария для требуемого контента.

(Следите за балансом между удобством использования/поиска с одной стороны и читабельностью окружающего кода с другой)
- выражения с include.
- Блокировка от повторного включения
- using декларации

Не-ASCII символы

Не-ASCII символы следует использоваться как можно реже, кодировка должна быть UTF-8.

Вы не должны хардкодить строки для показа пользователю (даже английские), поэтому Не-ASCII символы должны быть редкостью. Однако, в ряде случаев допустимо включать такие слова в код. Например, если код парсит файлы данных (с неанглийской кодировкой), возможно включать в код национальные слова-разделители. В более общем случае, код юнит-тестов может содержать национальные строки. В этих случаях следует использовать кодировку UTF-8, т.к. она понятна большинству утилит (которые понимают не только ASCII).

Кодировка hex также допустима, особенно если она улучшает читабельность. Например, `"\xEF\xBB\xBF"` или `"\uFEFF"` - неразрывный пробел нулевой длины в Юникоде, и который не должен отображаться в правильном UTF-8 тексте.

По возможности избегайте использования префикса `u8`. Начиная с C++20 семантика сильно поменялась (в отличие от C++17) и теперь этот префикс создаёт массивы символов `char8_t` (вместо `char`). Возможно, в C++23 всё опять поменяется.

Избегайте использования символьных типов `char16_t` и `char32_t` т.к. они предназначены для не-UTF-8 строк. По тем же причинам не используйте `wchar_t` (кроме случаев работы с Windows API, использующий `wchar_t`).

Пробелы против Табуляции

Используйте только пробелы для отступов. 2 пробела на один отступ.

Мы используем пробелы для отступов. Не используйте табуляцию в своём коде - настройте свой редактор на вставку пробелов при нажатии клавиши Tab.

Объявления и определения функций

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.

Пример правильного оформления функции:

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
  DoSomething();
  ...
}
```

В случае если одной строки мало:

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1, Type par_name2,
                                             Type par_name3) {
  DoSomething();
  ...
}
```

или, если первый параметр также не помещается:

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // Отступ 4 пробела
    Type par_name2,
    Type par_name3) {
  DoSomething();  // Отступ 2 пробела
  ...
}
```

Несколько замечаний:

- Выбирайте хорошие имена для параметров.
- Имя параметра можно опустить, если он не используется в определении функции.
- Если тип возвращаемого значения и имя функции не помещаются в одной строке, тип оставьте на одной строке, имя функции перенесите на следующую. В этом случае не делайте дополнительный отступ перед именем функции.
- Открывающая круглая скобка всегда находится на одной строке с именем функции.
- Не вставляйте пробелы между именем функции и открывающей круглой скобкой.
- Не вставляйте пробелы между круглыми скобками и параметрами.
- Открывающая фигурная скобка всегда в конце последней строки определения. Не переносите её на новую строку.
- Закрывающая фигурная скобка располагается либо на отдельной строке, либо на той же строке, где и открывающая скобка.
- Между закрывающей круглой скобкой и открывающей фигурной скобкой должен быть пробел.
- Старайтесь выравнивать все параметры.
- Стандартный отступ - 2 пробела.

- При переносе параметров на другую строку используйте отступ 4 пробела.

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

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

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) {}
```

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

```
  ABSL_ATTRIBUTE_NOINLINE void ExpensiveFunction();
[[nodiscard]] bool IsOk();
```

Лямбды

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

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; }
```

Короткие лямбды можно использовать напрямую как аргумент функции.

```
absl::flat_hash_set<int> to_remove = {7, 8, 9};
std::vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
digits.erase(std::remove_if(digits.begin(), digits.end(), [&to_remove](int i) {
               return to_remove.contains(i);
             }),
             digits.end());
```

Числа с плавающей запятой

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.

```
// Плохо (прим. пер)
float f = 1.f;
long double ld = -.5L;
double d = 1248e6;
```

```
float f = 1.0f;
float f2 = 1.0;  // OK
float f3 = 1;    // OK
long double ld = -0.5L;
double d = 1248.0e6;
```

Вызов функции

Следует либо писать весь вызов функции одной строкой, либо размещать аргументы на новой строке. И отступ может быть либо по первому аргументу, либо 4 пробела. Старайтесь минимизировать количество строк, размещайте по несколько аргументов на каждой строке.

Формат вызова функции:

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

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);
```

Допускается размещать аргументы на нескольких строках с отступом в 4 пробела:

```
if (...) {
  ...
  ...
  if (...) {
    bool result = DoSomething(
        argument1, argument2,  // Отступ 4 пробела
        argument3, argument4);
    ...
  }
```

Старайтесь размещать по несколько аргументов в строке, уменьшая количество строк на вызов функции (если это не ухудшает читабельность). Некоторые считают, что форматирование строго по одному аргументу в строке более читабельно и облегчает редактирование аргументов. Однако, мы ориентируемся прежде всего на читателей кода (не редактирование), поэтому предлагаем ряд подходов для улучшения читабельность.

Если несколько аргументов в одной строке ухудшают читабельность (из-за сложности или запутанности выражений), попробуйте создать для аргументов "говорящие" переменные:

```
int my_heuristic = scores[x] * y + bases[x];
bool result = DoSomething(my_heuristic, x, y, z);
```

Или разместите сложный аргумент на отдельной строке и добавьте поясняющий комментарий:

```
bool result = DoSomething(scores[x] * y + bases[x],  // Небольшая эвристика
                          x, y, z);
```

Если в вызове функции ещё есть аргументы, которые желательно разместить на отдельной строке - размещайте. Решение должно основываться улучшении читабельность кода.

Иногда аргументы формируют структуру. В этом случае форматируйте аргументы согласно требуемой структуре:

```
// Преобразование с помощью матрицы 3x3
my_widget.Transform(x1, x2, x3,
                    y1, y2, y3,
                    z1, z2, z3);
```

Форматирование списка инициализации

Форматируйте список инициализации аналогично вызову функции.

Если список в скобках следует за именем (например, имя типа или переменной), форматируйте {} как будто это вызов функции с этим именем. Даже если имени нет, считайте что оно есть, только пустое.

```
// Пример списка инициализации на одной строке.
return {foo, bar};
functioncall({foo, bar});
std::pair<int, int> p{foo, bar};

// Когда хочется разделить на строки.
SomeFunction(
    {"assume a zero-length name before {"},
    some_other_function_parameter);
SomeType variable{
    some, other, values,
    {"assume a zero-length name before {"},
    SomeOtherType{
        "Very long string requiring the surrounding breaks.",
        some, other, values},
```

```
    SomeOtherType{"Slightly shorter string",
                  some, other, values}};
SomeType variable{
    "This is too long to fit all in one line"};
MyType m = {  // Here, you could also break before {.
    superlongvariablename1,
    superlongvariablename2,
    {short, interior, list},
    {interiorwrappinglist,
     interiorwrappinglist2}};
```

Операторы цикла и ветвления

В общем, операторы цикла и ветвления состоят из следующих **компонентов**:

- Одно или несколько **ключевых слов** операции (например, `if`, `else`, `switch`, `while`, `do` или `for`).
- Одно **условие** или **итерация**, заключённые в круглые скобки.
- Один или несколько **команд** или блоков команд.

Особенности форматирования:

- Отдельные компоненты должны разделяться одиночным пробелом (не переводом строки).
- Внутри условия или итерации размещайте один пробел (или перевод строки) между каждой точкой с запятой и следующей лексемой, за исключением закрывающих скобок и других точек с запятой.
- Внутри условия или итерации не добавляйте пробел после открывающей скобки, также не добавляйте перед закрывающей скобкой.
- Помещайте команды внутрь блока (т.е. в фигурные скобки).
- Внутри блока команд сделайте перевод строки сразу же после открывающей скобки, ещё один перевод строки прямо перед закрывающей скобкой.

```
if (condition) {                  // Хорошо - никаких пробелов внутри скобок, пробел перед скобкой
  DoOneThing();                   // Хорошо - отступ два пробела
  DoAnotherThing();
} else if (int a = f(); a == 3) {  // Хорошо - закрывающая скобка на новой строке, else на той же строке
  DoAThirdThing(a);
} else {
  DoNothing();
}
```

```
// Хорошо - применение тех же правил для цикла
while (condition) {
  RepeatAThing();
}

// Хорошо - применение тех же правил для цикла
do {
  RepeatAThing();
} while (condition);

// Хорошо - применение тех же правил для цикла
for (int i = 0; i < 10; ++i) {
  RepeatAThing();
}
```

```
if(condition) {}    // Плохо - отсутствует пробел после `if`
else if ( condition ) {}           // Плохо - пробел между скобками и условием
else if (condition){}              // Плохо - пропущен пробел перед `{`
else if(condition){}               // Плохо - пропущено несколько пробелов

for (int a = f();a == 10) {}       // Плохо - пропущен пробел после точки с запятой

// Плохо - команда между `if` и `else` не заключена в фигурные скобки
if (condition)
  foo;
else {
  bar;
}

// Плохо - всё выражение не укладывается в одну/две строки (прим. см. исключение ниже)
if (condition)
  // Comment
  DoSomething();

// Плохо - всё выражение не укладывается в одну/две строки (прим. см. исключение ниже)
if (condition1 &&
    condition2)
  DoSomething();
```

По историческим причинам допускается одно исключение из вышеприведённых правил: сами фигурные скобки для команды и переводы строк внутри блока можно опустить если всё выражение будет записано либо одной строкой (в этом случае между закрывающей круглой скобкой и командой ставится пробел), либо на двух строках (тогда вставляется перевод строки после закрывающей круглой скобки и фигурные скобки не ставятся).

```
// OK - всё записано единственной строкой
if (x == kFoo) { return new Foo(); }

// OK - фигурные скобки опциональны
if (x == kFoo) return new Foo();

// OK - условие расположено на первой строке, команда на второй
if (x == kBar)
  Bar(arg1, arg2, arg3);
```

Это исключение неприменимо для выражений с несколькими ключевыми словами, такими как `if ... else` или `do ... while`.

```
// Плохо - пропущены фигурные скобки в конструкции `if ... else`
if (x) DoThis();
else DoThat();

// Плохо - пропущены фигурные скобки в конструкции `do ... while`
do DoThis();
while (x);
```

Используйте такой стиль только когда выражения короткие. Учитывайте, что сложные условия или блоки команд лучше читаются, когда есть разделение скобками. Также, в некоторых проектах всегда требуются фигурные скобки.

Блоки `case` в операторе `switch` могут быть как в фигурных скобках, так и без них - в зависимости от личных предпочтений. Если фигурные скобки используются, то размещайте их согласно примеру:

```
switch (var) {
  case 0: {  // Отступ 2 пробела
    Foo();   // Отступ 4 пробела
    break;
  }
  default: {
    Bar();
  }
}
```

Пустой цикл должен быть оформлен либо как пара скобок, либо как `continue` без скобок. Не используйте одиночную точку с запятой.

```
while (condition) {}  // Хорошо - `{}` указывает на отсутствие внутренней логики
while (condition) {
  // Комментарий также допустим
}
while (condition) continue;  // Хорошо - `continue` указывает на отсутствие внутренней логики
```

```
while (condition);  // Плохо - выглядит как часть цикла `do-while`
```

Указатели и ссылки

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

Ниже приведены примеры правильного форматирования выражений с указателями и ссылками:

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

Отметим:

- Не окружайте пробелами функции доступа '.' и '->'.
- Операторы * или & не отделяются пробелами.

Когда описывается указатель или ссылка (объявления или определения переменных, аргументов, возвращаемых значений, параметров шаблонов и т.п.) допустимо вставлять пробел как перед, так и после '*'/'&'. Если пробел ставится в конце, то в некоторых случаях его можно опустить (параметры шаблонов и т.п.).

```
// Отлично, пробел до *, &
char *c;
const std::string &str;
int *GetPointer();
std::vector<char *>

// Отлично, пробел после *, & (или опущен)
```

```
char* c;
const std::string& str;
int* GetPointer();
std::vector<char*>  // Заметьте, нет пробела между '*' и '>'
```

Старайтесь использовать единый стиль в файле. При модификации существующего файла применяйте используемое форматирование.

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;  // Запрещено - не используйте & или * в множественном объявлении
int* x, *y;  // Запрещено - не используйте & или * в множественном объявлении;  различное расположение пр
char * c;  // Плохо - пробелы с обеих сторон *
const std::string & str;  // Плохо - пробелы с обеих сторон &
```

Логические выражения

Если логическое выражение очень длинное (превышает типовое значение), используйте единый подход к разбивке выражения на строки.

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

```
if (this_one_thing > this_other_thing &&
    a_third_thing == a_fourth_thing &&
    yet_another && last_one) {
  ...
}
```

Отметим, что разбиение кода (согласно примеру) производится так, чтобы && и оператор AND завершали строку. Такой стиль чаще используется с коде Google, хотя расположение операторов в начале строки тоже допустимо. Также, можете добавлять дополнительные круглые скобки для улучшения читабельности (но не чрезмерно). Учтите, что использование операторов в виде пунктуации (такие как && и ~) более предпочтительно, чем операторы в виде слов `and` и `compl`.

Возвращаемые значения

Не заключайте простые выражения `return` в скобки.

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!
```

Инициализация переменных и массивов

Вы можете выбирать между вариантами =, `()` и `{}`. Следующие примеры кода корректны:

```
int x = 3;
int x(3);
int x{3};
std::string name = "Some Name";
std::string name("Some Name");
std::string name{"Some Name"};
```

Будьте внимательны при использовании списка инициализации `{...}` для типа, у которого есть конструктор с `std::initializer_list`. Компилятор предпочтёт использовать конструктор `std::initializer_list` при наличии *списка в фигурных скобках*. Заметьте, что пустые фигурные скобки `{}` - это особый случай и будет вызван конструктор по-умолчанию (если он доступен). Для явного использования конструктора без `std::initializer_list` применяйте круглые скобки вместо фигурных.

```
std::vector<int> v(100, 1);  // Вектор из сотни единиц
std::vector<int> v{100, 1};  // Вектор из 2-х элементов: 100 и 1
```

Также конструирование с фигурными скобками запрещает ряд преобразований целых типов (преобразования с уменьшением точности). И можно получить ошибки компиляции.

```
int pi(3.14);  // Ок: pi == 3
int pi{3.14};  // Ошибка компиляции: "сужающее" преобразование
```

Директивы препроцессора

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

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 (lopsided_score) {
    #if DISASTER_PENDING  // Неправильно! "#if" должна быть в начале строки
    DropEverything();
    #endif                // Неправильно! Не делайте отступ для "#endif"
    BackToNormal();
  }
```

Форматирование классов

Размещайте секции в следующем порядке: `public`, `protected` и `private`. Отступ - один пробел.

Ниже описан базовый формат для класса (за исключением комментариев, см. описание Комментирование класса):

```
class MyClass : public OtherClass {
 public:      // Отступ 1 пробел
  MyClass();  // Обычный 2-х пробельный отступ
  explicit MyClass(int var);
  ~MyClass() {}

  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

  void set_some_var(int var) { some_var_ = var; }
  int some_var() const { return some_var_; }

 private:
  bool SomeInternalFunction();

  int some_var_;
  int some_other_var_;
};
```

Замечания:

- Имя базового класса пишется в той же строке, что и имя наследуемого класса (конечно, с учётом ограничения в 80 символов).
- Ключевые слова `public:`, `protected:`, и `private:` должны быть с отступом в 1 пробел.
- Перед каждым из этих ключевых слов должна быть пустая строка (за исключением первого упоминания). Также в маленьких классах пустые строки можно опустить.
- Не добавляйте пустую строку после этих ключевых слов.
- Секция `public` должна быть первой, за ней `protected` и в конце секция `private`.
- См. Порядок объявления для выстраивания деклараций в каждой из этих секций.

Списки инициализации конструктора

Списки инициализации конструктора могут быть как в одну строку, так и на нескольких строках с 4-х пробельным отступом.

Ниже представлены правильные форматы для списков инициализации:

```
// Всё в одну строку
MyClass::MyClass(int var) : some_var_(var) {
  DoSomething();
}

// Если сигнатура и список инициализации не помещается на одной строке,
// нужно перенести двоеточие и всё что после него на новую строку
MyClass::MyClass(int var)
    : some_var_(var), some_other_var_(var + 1) {
  DoSomething();
}

// Если список занимает несколько строк, то размещайте каждый элемент на
// отдельной строке и всё выравниваем
MyClass::MyClass(int var)
    : some_var_(var),             // Отступ 4 пробела
      some_other_var_(var + 1) {  // Выравнивание по предыдущему
  DoSomething();
}

// Как и в других случаях, фигурные скобки могут размещаться на одной строке
MyClass::MyClass(int var)
    : some_var_(var) {}
```

Форматирование пространств имён

Содержимое в пространстве имён пишется без отступа.

Пространство имён не добавляет отступов. Например:

```
namespace {

void foo() {  // Хорошо. Без дополнительного отступа
  ...
```

```
}

}  // namespace
```

Не делайте отступов в пространстве имён:

```
namespace {

  // Ошибка! Сделан отступ там, где не нужно
  void foo() {
    ...
  }

}  // namespace
```

Горизонтальная разбивка

Используйте горизонтальную разбивку в зависимости от ситуации. Никогда не добавляйте пробелы в конец строки.

Общие принципы

```
int i = 0;  // Два пробела перед комментарием в конце строки.
void f(bool b) {  // Перед открывающей фигурной скобкой всегда ставьте пробел
  ...
int i = 0;  // Обычно перед точкой с запятой нет пробела
// Пробелы внутри фигурных скобок для списка инициализации можно добавлять на ваш выбор.
// Если вы добавляете пробелы, то ставьте их с обеих сторон
int x[] = { 0 };
int x[] = {0};

// Пробелы вокруг двоеточия в списках наследования и инициализации
class Foo : public Bar {
 public:
```

```
  // Для inline-функции добавляйте
  // пробелы внутри фигурных скобок (кроме пустого блока)
  Foo(int b) : Bar(), baz_(b) {}  // Пустой блок без пробелов
  void Reset() { baz_ = 0; }  // Пробелы разделяют фигурные скобки и реализацию
  ...
```

Добавление разделительных пробелов может мешать при слиянии кода. Поэтому: Не добавляйте разделительных пробелов в существующий код. Вы можете удалить пробелы, если уже модифицировали эту строку. Или сделайте это отдельной операцией (предпочтительно, чтобы с этим кодом при этом никто не работал).

Циклы и условия

```
if (b) {          // Пробел после ключевого слова в условии или цикле
} else {          // Пробелы вокруг else
}
while (test) {}   // Внутри круглых скобок обычно не ставят пробел
switch (i) {
for (int i = 0; i < 5; ++i) {
// Циклы и условия могут могут внутри быть с пробелам. Но это редкость.
// В любом случае, будьте последовательны
switch ( i ) {
if ( test ) {
for ( int i = 0; i < 5; ++i ) {
// В циклах после точки с запятой всегда ставьте пробел
// Также некоторые любят ставить пробел и перед точкой с запятой, но это редкость
for ( ; i < 5 ; ++i) {
  ...

// В циклы по диапазону всегда ставьте пробел до двоеточия и после
for (auto x : counts) {
  ...
}
switch (i) {
  case 1:         // Перед двоеточием в case нет пробела
    ...
  case 2: break;  // После двоеточия есть пробел, если дальше (на той же строке) идёт код
```

Операторы

```
// Операторы присваивания всегда окружайте пробелами
x = 0;

// Другие бинарные операторы обычно окружаются пробелами,
// хотя допустимо умножение/деление записывать без пробелов.
// Между выражением внутри скобок и самими скобками не вставляйте пробелы
v = w * x + y / z;
v = w*x + y/z;
v = w * (x + z);

// Унарные операторы не отделяйте от их аргумента
x = -5;
++x;
if (x && !y)
  ...
```

Шаблоны и приведение типов

```
// Не ставьте пробелы внутри угловых скобок (< и >),
// перед <, между >( в приведении
std::vector<std::string> x;
y = static_cast<char*>(x);

// Пробелы между типом и знаком указателя вполне допустимы. Но смотрите на уже используемый формат кода
std::vector<char *> x;
```

Вертикальная разбивка

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.

Несколько полезных замечаний о пустых строках:

- Пустая строка в начале или в конце функции не улучшит читабельность.
- Пустые строки в цепочке блоков if-else могут улучшить читабельность.
- 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.
- Пустые строки вначале декларации пространства имён (или блока пространств имён) могут помочь читабельности благодаря визуальному разделению смыслового контента и (часто не-семантических) обёрток. И в случае, когда первое объявление внутри пространств(а) имён предваряется комментарием, это становится особым случаем указанного выше правила, помогая "объединить" комментарий с последующим объявлением.

Исключения из правил

Соглашения по кодированию, описанные выше являются обязательными. Однако, как и в любых правилах, иногда в них есть исключения, которые сейчас и обсудим.

Существующий код, не соответствующий стилю

Допустимо отклоняться от правил, если производится работа с кодом, не соответствующим этому руководству.

Если модифицируется код, написанный другим стилем, допустимо отклоняться от требований этого руководства и использовать "местный" стиль, чтобы получить согласованный код. Если сомневаетесь - спросите автора кода (или того, кто это поддерживает). Помните, что *согласованность* включает также и текущий стиль кода.

Программирование под Windows

Программисты под Windows могут использовать особенный набор соглашений о кодировании, основанный на стиле заголовочных файлов в Windows и другом коде от Microsoft. Так как хочется сделать, чтобы код был понятным для всех, то рекомендуется использовать единое руководство по стилю в C++, одинаковое для всех платформ.

Повторим несколько рекомендаций, которые отличают данное руководство от стиля Windows:

- Не используйте венгерскую нотацию (например, именование целочисленной переменной как `iNum`). Вместо этого используйте соглашения об именовании от Google, включая расширение `.cc` для файлов с исходным кодом.
- Windows определяет собственные синонимы для базовых типов, такие как `DWORD`, `HANDLE` и др. Понятно, что при вызове Windows API рекомендуется использовать именно их. И всё равно, старайтесь определять типы, максимально похожие на C++. Например, используйте `const TCHAR *` вместо `LPCTSTR`.
- При компиляции кода с помощью Microsoft Visual C++ установите уровень предупреждений 3 или выше. Также установите настройку, чтобы трактовать все предупреждения как ошибки.
- Не используйте `#pragma once`. Вместо этого используйте стандартную защиту от повторного включения, описанную в руководстве от Google. Компоненты пути в имени для макроопределения защиты должны быть относительными к корню проекта.
- Вообще, не используйте нестандартные расширения, такие как `#pragma` и `__declspec` (исключение для случаев крайней необходимости). Использование `__declspec(dllimport)` и `__declspec(dllexport)` допустимо, однако следует оформить их как макросы `DLLIMPORT` и `DLLEXPORT`: в этом случае их можно легко заблокировать, если код будет распространяться.

С другой стороны, есть правила, которые можно нарушать при программировании под Windows:

- Обычно рекомендуется не использовать множественное наследование реализации; однако это требуется при использовании COM и некоторых классов ATL/WTL. В этом случае (при реализации COM или ATL/WTL) нарушение правила допустимо.
- Хотя использование исключений в собственном коде не рекомендуется, они интенсивно используются в ATL и некоторых STL (в том числе и в варианте библиотеки от Visual C++). Когда используете ATL, следует определить `_ATL_NO_EXCEPTIONS`, чтобы запретить исключения. Желательно разобраться, можно ли запретить исключения и в STL, однако, если это не реализуемо, то допустимо разрешить исключения в компиляторе. (Учтите, что это разрешено только для компиляции STL. Пользовательский код всё равно не должен содержать обработчики исключений).
- Типичный способ работы с прекомпилированными заголовочными файлами - включить такой файл первым в каждый файл исходников, обычно с именем `StdAfx.h` или `precompile.h`. Чтобы не создавать проблем при распространении кода, лучше избегать явного включения такого файла (за исключением `precompile.cc`). Используйте опцию компилятора `/FI` для автоматического включения такого файла.
- Заголовочный файлы ресурсов (обычно `resource.h`), содержащий только макросы, может не следовать рекомендациям этого руководства.

Перевод

Кислов Евгений, 2019-2021, 2023, 2024
email: style@evgenykislov.com

evgenykislov.com

Актуальная версия перевода:
Руководство Google по стилю в C++
