

# Оглавление

Избранные рекомендации по C++ .....	14
Предисловие .....	17
Вступление .....	18
О книге.....	20
Код примеров.....	23
Благодарности .....	24
Об авторах .....	26
От издательства .....	28

## ЧАСТЬ I BIKESHEDDING — ЭТО ПЛОХО

<b>Глава 1.1.</b> Р.2. Придерживайтесь стандарта ISO C++ .....	30
Что такое стандарт ISO C++ .....	30
История C++ .....	30
Инкапсуляция вариаций .....	32
Вариации в окружении времени выполнения .....	32
Вариации на уровне языка C++ и компилятора .....	33
Расширения для C++.....	34
Защита заголовочных файлов .....	35
Вариации в основных типах .....	35
Нормативные ограничения .....	36
Изучение старых способов .....	37
Обратная совместимость в C++.....	37
Прямая совместимость и Y2K.....	38
Следите за последними изменениями в стандарте.....	39
IsoCpp.....	39
Конференции.....	40
Другие источники.....	40

<b>Глава 1.2.</b> F.51. Если есть выбор, используйте аргументы по умолчанию вместо перегрузки.....	42
Введение.....	42
Доработка ваших абстракций: дополнительные аргументы или перегрузка? .....	43
Тонкости разрешения перегрузки .....	45
Вернемся к примеру .....	47
Однозначная природа аргументов по умолчанию.....	49
Альтернативы перегрузке .....	50
Иногда без перегрузки не обойтись.....	51
Подведем итог .....	52
<b>Глава 1.3.</b> C.45. Не определяйте конструктор по умолчанию, который просто инициализирует переменные-члены; для этой цели лучше использовать внутриклассовые инициализаторы членов ....	53
Зачем нужны конструкторы по умолчанию .....	53
Как инициализируются переменные-члены.....	55
Что может случиться, если поддерживать класс будут два человека .....	58
Сборная солянка из конструкторов .....	58
Аргументы по умолчанию могут запутать ситуацию в перегруженных функциях.....	60
Подведем итог .....	60
<b>Глава 1.4.</b> C.131. Избегайте тривиальных геттеров и сеттеров .....	62
Архаичная идиома .....	62
Абстракции.....	63
Простая инкапсуляция.....	66
Инварианты класса.....	69
Существительные и глаголы.....	71
Подведем итог .....	72
<b>Глава 1.5.</b> ES.10. Объявляйте имена по одному в каждом объявлении.....	73
Позвольте представить.....	73
Обратная совместимость .....	76
Пишите более ясные объявления.....	77
Структурное связывание .....	78
Подведем итог .....	79

<b>Глава 1.6.</b> NR.2. Функции не обязательно должны иметь только один оператор возврата.....	80
Правила меняются .....	80
Гарантия очистки.....	83
Идиома RAII .....	85
Пишите хорошие функции .....	88
Подведем итог .....	90

## ЧАСТЬ II НЕ НАВРЕДИТЕ СЕБЕ

<b>Глава 2.1.</b> P.11. Инкапсулируйте беспорядочные конструкции, а не разбрасывайте их по всему коду.....	92
Все одним глотком .....	92
Что означает инкапсулировать запутанную конструкцию .....	94
Назначение языка и природа абстракции.....	96
Уровни абстракции.....	100
Абстракция путем рефакторинга и проведения линии .....	101
Подведем итог .....	102
<b>Глава 2.2.</b> I.23. Минимизируйте число параметров в функциях .....	103
Сколько они должны получать? .....	103
Упрощение через абстрагирование.....	105
Делайте так мало, как возможно, но не меньше .....	107
Примеры из реальной жизни .....	109
Подведем итог .....	111
<b>Глава 2.3.</b> I.26. Если нужен кросс-компилируемый ABI, используйте подмножество в стиле C .....	112
Создавайте библиотеки.....	112
Что такое ABI.....	114
Сокращайте до абсолютного минимума.....	115
Распространение исключений.....	118
Подведем итог .....	119
<b>Глава 2.4.</b> C.47. Определяйте и инициализируйте переменные-члены в порядке их объявления.....	121
Подведем итог .....	131

<b>Глава 2.5.</b> SR3. Сведите к минимуму явное совместное использование записываемых данных .....	132
Традиционная модель выполнения.....	132
Подождите, это еще не все.....	134
Предотвращение взаимоблокировок и гонок за данными .....	137
Отказ от блокировок и мьютексов .....	140
Подведем итог .....	143
<b>Глава 2.6.</b> T.120. Используйте метапрограммирование шаблонов, только когда это действительно необходимо .....	144
std::enable_if => requires.....	152
Подведем итог .....	156

### ЧАСТЬ III ПРЕКРАТИТЕ ЭТО ИСПОЛЬЗОВАТЬ

<b>Глава 3.1.</b> I.11. Никогда не передавайте владение через простой указатель (T*) или ссылку (T&).....	158
Использование области свободной памяти .....	158
Производительность интеллектуальных указателей.....	161
Использование простой семантики ссылок .....	163
gsl::owner .....	164
Подведем итог .....	167
<b>Глава 3.2.</b> I.3. Избегайте синглтонов.....	168
Глобальные объекты — это плохо.....	168
Шаблон проектирования «Синглтон» .....	169
Фиаско порядка статической инициализации .....	170
Как скрыть синглтон.....	173
Только один из них должен существовать в каждый момент работы кода .....	174
Подождите минутку.....	176
Подведем итог .....	179
<b>Глава 3.3.</b> C.90. Полагайтесь на конструкторы и операторы присваивания вместо memset и memсru .....	180
В погоне за максимальной производительностью.....	180
Ужасные накладные расходы конструкторов.....	181
Самый простой класс.....	183

О чем говорит стандарт.....	185
А как же метасру? .....	188
Никогда не позволяйте себе недооценивать компилятор.....	189
Подведем итог .....	191
<b>Глава 3.4.</b> ES.50. Не приводите переменные с квалификатором <code>const</code> к неконстантному типу .....	192
Работа с большим количеством данных.....	193
Брандмауэр <code>const</code> .....	195
Реализация двойного интерфейса .....	196
Кэширование и отложенные вычисления .....	198
Два вида <code>const</code> .....	199
Сюрпризы <code>const</code> .....	201
Подведем итог .....	202
<b>Глава 3.5.</b> E.28. При обработке ошибок избегайте глобальных состояний (например, <code>errno</code> ).....	204
Обрабатывать ошибки сложно .....	204
Язык C и <code>errno</code> .....	204
Коды возврата.....	206
Исключения .....	207
<code>&lt;system_error&gt;</code> .....	208
<code>Boost.Outcome</code> .....	209
Почему обрабатывать ошибки так сложно .....	210
Свет в конце туннеля .....	212
Подведем итог .....	214
<b>Глава 3.6.</b> SF.7. Не используйте <code>using namespace</code> в глобальной области видимости в заголовочном файле .....	215
Не делайте этого .....	215
Неоднозначность .....	216
Использование <code>using</code> .....	217
Куда попадают символы.....	219
Еще более коварная проблема .....	222
Решение проблемы операторов разрешения области видимости .....	223
Искушение и расплата .....	225
Подведем итог .....	226

## ЧАСТЬ IV

### ИСПОЛЬЗУЙТЕ НОВУЮ ОСОБЕННОСТЬ ПРАВИЛЬНО

<b>Глава 4.1.</b>	<b>F.21.</b> Для возврата нескольких выходных значений используйте структуры или кортежи.....	228
	Форма сигнатуры функции .....	228
	Документирование и аннотирование .....	230
	Теперь можно вернуть объект .....	231
	Можно также вернуть кортеж .....	234
	Передача и возврат по неконстантной ссылке .....	237
	Подведем итог .....	240
<b>Глава 4.2.</b>	<b>Enum.3.</b> Старайтесь использовать классы-перечисления вместо простых перечислений.....	241
	Константы.....	241
	Перечисления с заданной областью видимости.....	244
	Базовый тип .....	246
	Неявное преобразование .....	247
	Подведем итог .....	249
<b>Глава 4.3.</b>	<b>ES.5.</b> Минимизируйте области видимости .....	250
	Природа области видимости .....	250
	Область видимости блока .....	251
	Область видимости пространства имен.....	253
	Область видимости класса.....	256
	Область видимости параметров функции .....	258
	Область видимости перечисления .....	259
	Область действия параметра шаблона .....	260
	Область видимости как контекст .....	261
	Подведем итог .....	262
<b>Глава 4.4.</b>	<b>Con.5.</b> Используйте <code>constexpr</code> для определения значений, которые можно вычислить на этапе компиляции .....	263
	От <code>const</code> к <code>constexpr</code> .....	263
	C++ по умолчанию.....	265
	Использование <code>constexpr</code> .....	267
	<code>inline</code> .....	271
	<code>constexpr</code> .....	272

constinit .....	273
Подведем итог .....	275
<b>Глава 4.5.</b> T.1. Используйте шаблоны для повышения уровня абстрактности кода .....	276
Повышение уровня абстракции .....	278
Шаблоны функций и абстракция .....	280
Шаблоны классов и абстракция .....	283
Выбор имени — сложная задача .....	285
Подведем итог .....	286
<b>Глава 4.6.</b> T.10. Задавайте концепции для всех аргументов шаблона .....	287
Как мы здесь оказались? .....	287
Ограничение параметров .....	290
Как абстрагировать свои концепции .....	293
Разложение на составляющие через концепции .....	296
Подведем итог .....	297

## ЧАСТЬ V ПИШИТЕ ХОРОШИЙ КОД ПО УМОЛЧАНИЮ

<b>Глава 5.1.</b> P.4. В идеале программа должна быть статически типобезопасной .....	300
Безопасность типов — это средство защиты в C++ .....	300
Объединения .....	302
Приведение .....	304
Целые без знака .....	307
Буферы и размеры .....	310
Подведем итог .....	311
<b>Глава 5.2.</b> P.10. Неизменяемые данные предпочтительнее изменяемых .....	312
Неправильные значения по умолчанию .....	312
const в объявлениях функций .....	315
Подведем итог .....	319
<b>Глава 5.3.</b> I.30. Инкапсулируйте нарушения правил .....	320
Соккрытие неприглядных вещей .....	320
Поддержание видимости, что все в порядке .....	322
Подведем итог .....	327

<b>Глава 5.4.</b> ES.22. Не объявляйте переменные, пока не получите значения для их инициализации .....	329
Важность выражений и операторов .....	329
Объявление в стиле C.....	330
Объявление с последующей инициализацией .....	332
Максимальное откладывание объявления .....	333
Локализация контекстно зависимой функциональности .....	335
Устранение состояния.....	337
Подведем итог .....	339
<b>Глава 5.5.</b> Per.7. При проектировании учитывайте возможность последующей оптимизации .....	340
Максимальная частота кадров.....	340
Работа вдалеке от железа .....	342
Оптимизация через абстракцию.....	346
Подведем итог .....	349
<b>Глава 5.6.</b> E.6. Используйте идиому RAII для предотвращения утечек памяти.....	350
Детерминированное уничтожение .....	350
Утечка файлов .....	353
Почему это так важно .....	356
Все это выглядит чересчур сложным: будущие возможности .....	358
Где все это получить.....	361
Заключение .....	364
Послесловие.....	366



## ГЛАВА 1.2

# F.51. Если есть выбор, используйте аргументы по умолчанию вместо перегрузки

### ВВЕДЕНИЕ

Проектирование API — ценный навык. Разбивая задачу на составляющие, вы должны выделить абстракции и спроектировать для них интерфейс, дав клиенту-пользователю четкие и недвусмысленные инструкции в виде совершенно очевидного набора функций с тщательно подобранными именами. Есть такая рекомендация, которая гласит, что код должен быть самодокументируемым. Несмотря на кажущуюся чрезмерную амбициозность цели, именно в проектировании API вы должны приложить все силы для ее достижения.

Базы кода постоянно растут. Это неизбежно, и от этого никуда не деться. Со временем обнаруживается и кодируется все больше абстракций, решается больше задач, и сама предметная область неуклонно расширяется, чтобы затребовать и затем вместить больше вариантов использования программ. Это совершенно нормально. Это часть типичного процесса разработки и проектирования.

По мере добавления дополнительных абстракций в базу кода свою уродливую голову поднимает проблема однозначного именования. Подбор правильных, говорящих имен — сложная задача. И вы не раз убедитесь в этом в своей карьере программиста. Иногда возникает необходимость позволить клиенту (а часто им являетесь вы сами) сделать вроде бы то же самое, но немного по-другому.

В таких случаях перегрузка может показаться хорошей идеей. Различие между двумя абстракциями может заключаться лишь в передаваемых им аргументах; во всем остальном они семантически идентичны. Перегрузка функций позволяет повторно использовать имя функции, но с другим набором параметров. Однако если они действительно семантически идентичны, то, может быть, лучше выразить эту разницу в форме аргументов по умолчанию? Если это действительно так, то ваш API станет проще для понимания.

Прежде чем начать обсуждение, мы хотим напомнить вам о разнице между параметром и аргументом: аргумент — это то, что передается в функцию. Объявление функции включает список параметров, из которых один или несколько могут быть снабжены аргументами (значениями) по умолчанию. Не существует такого понятия, как параметр по умолчанию.

## **ДОРАБОТКА ВАШИХ АБСТРАКЦИЙ: ДОПОЛНИТЕЛЬНЫЕ АРГУМЕНТЫ ИЛИ ПЕРЕГРУЗКА?**

Рассмотрим для примера следующую функцию:

```
office make_office(float floor_space, int staff);
```

Эта функция возвращает экземпляр `office` — объект, представляющий собой офисное здание площадью `floor_space` квадратных метров и с кабинетами для `staff` сотрудников. Это одноэтажное здание с кухонными и туалетными помещениями и соответствующим количеством кофемашин, столов для настольного тенниса и массажных кабинетов. Однажды было решено расширить предметную область и добавить возможность моделирования двухэтажных офисных зданий. Это несколько усложнило ситуацию, так как двухэтажная модель предполагает определение путей эвакуации, более сложную схему кондиционирования воздуха, наличие лестниц в нужных местах и, конечно же, горки между этажами или, может быть, пожарного столба. К тому же нужно сообщить функции-конструктору, что она должна создать модель двухэтажного офисного здания. Это можно сделать с помощью третьего параметра:

```
office make_office(float floor_space, int staff, bool two_floors);
```

Проблема в том, что в этом случае придется просмотреть весь код и добавить `false` во все вызовы `make_office`. С другой стороны, можно определить

аргумент по умолчанию `false` для последнего параметра, и тогда ничего менять не придется. Вот как это выглядит:

```
office make_office(float floor_space, int staff, bool two_floors = false);
```

Одна короткая перекомпиляция — и все в порядке. К сожалению, демоны расширения предметной области еще не закончили: оказывается, одноэтажным офисным зданиям иногда требуется давать названия. Вы, как отзывчивый на вызовы и просьбы заказчика инженер, решаете расширить список параметров функции:

```
office make_office(float floor_space, int staff, bool two_floors = false,
    std::string const& building_name = {});
```

Вы переопределяете функцию и замечаете одну неприятность. Функция принимает четыре аргумента, причем последний необходим, только если третий аргумент `false` и из-за этого функция выглядит запутанной и сложной. Вы решаете добавить перегруженную версию функции:

```
office make_office(float floor_space, int staff, bool two_floors = false);
office make_office(float floor_space, int staff,
    std::string const& building_name);
```

Теперь у вас есть то, что известно как набор перегруженных функций. И каждый раз, встретив ссылку на имя функции, компилятор должен выбрать, какую реализацию выбрать, а для этого проверить типы переданных аргументов. Клиент вынужден вызывать правильную функцию, когда нужно идентифицировать здание. Наличие идентификации подразумевает одноэтажный офис.

Например, представьте, что в некотором клиентском коде предпринята попытка создать офис площадью 24 000 квадратных метров для 200 сотрудников. Офис расположен в одноэтажном здании с названием Eagle Heights. Вот как должен выглядеть соответствующий вызов:

```
auto eh_office = make_office(24000.f, 200, "Eagle Heights");
```

Конечно, вы должны гарантировать соблюдение определенной семантики в каждой функции и обеспечить, чтобы все функции действовали одинаково. Это тяжелое бремя сопровождения. Возможно, уместнее реализовать единственную функцию и потребовать от вызывающей стороны явно обозначать свой выбор.

Мы уже слышим, как вы говорите: «Постойте! А если написать приватную реализацию функции? В таком случае можно гарантировать единообразие

моделирования, просто вызывая приватную реализацию из перегруженных версий, и все будет в порядке».

Вы были бы правы, если бы не одно но. Клиенты могут с подозрением отнестись к двум функциям. Их может насторожить возможность несогласованности реализаций. Излишняя осторожность с их стороны может вселить в них страх и опасения. Одна функция с двумя аргументами по умолчанию для переключения между алгоритмами выглядит более надежно.

И снова мы слышим ваше возражение: «Это смешно! Я пишу качественный код, и мои клиенты доверяют мне. У меня весь код охвачен модульными тестами, и все в порядке. Вот уж спасибо так спасибо!»

К сожалению, даже если вы действительно пишете код высочайшего качества, это не обязательно относится к вашим клиентам. Взгляните еще раз на инициализацию `eh_office` и проверьте себя, сможете ли вы заметить ошибку. А другой человек сможет? Подумайте, а пока мы поговорим о разрешении перегрузки кода (как выбираются перегруженные версии).

## ТОНКОСТИ РАЗРЕШЕНИЯ ПЕРЕГРУЗКИ

Разрешение перегрузки — сложная задача для освоения. Почти 2 % стандарта C++20 посвящены определению работы механизма разрешения перегрузок. Вот краткий обзор.

Когда компилятор встречает вызов функции, он должен определить, на какую из функций этот вызов ссылается. Перед этим компилятор составляет список всех идентификаторов. Возможно, что в программе имеется несколько функций с одинаковыми именами, но с разными параметрами — набор перегруженных версий. Как компилятор определяет, какую из них вызывать?

Сначала он отбирает функции с тем же количеством параметров, с меньшим количеством и с параметром-многоточием или с большим количеством параметров, среди которых избыточные параметры имеют аргументы по умолчанию. Если какой-либо из кандидатов имеет предложение `requires` (`requires clause`, нововведение, появившееся в C++20), то оно должно быть удовлетворено. Ни один правосторонний (`rvalue`) аргумент не должен соответствовать неконстантному левостороннему (`lvalue`) параметру, и любой левосторонний (`lvalue`) аргумент не должен соответствовать ссылочному правостороннему (`rvalue`) параметру. Каждый аргумент должен иметь возможность быть преобразованным в соответствующий параметр посредством неявной последовательности преобразований.

В нашем примере компилятору передаются две версии `make_office`, отличающиеся третьим параметром. Одна принимает логическое значение, которое по умолчанию равно `false`, а вторая — `std::string const&`. По количеству параметров обе версии соответствуют операции инициализации `eh_office`.

Ни в одной из этих функций нет предложения `requires`, поэтому можно пропустить этот шаг. Точно так же нет ничего экзотического в привязках ссылок.

Наконец, каждый аргумент должен быть преобразован в соответствующий параметр. Первые два аргумента не требуют преобразования. Третий аргумент — это `char const*`, который, очевидно, преобразуется в `std::string` через неявный конструктор, являющийся частью интерфейса `std::string`. Но, к сожалению, это еще не все.

Когда имеется несколько перегруженных версий функции, они ранжируются по параметрам, чтобы упростить поиск наиболее подходящей. Версия F1 считается предпочтительнее версии F2, если неявные преобразования для всех аргументов F1 не хуже, чем у F2. Кроме того, в F1 должен быть хотя бы один параметр, неявное преобразование которого лучше соответствующего неявного преобразования в F2.

Слово «лучше» настораживает. Как ранжируются последовательности неявных преобразований?

Существует три типа последовательностей неявных преобразований: стандартная, определяемая пользователем и последовательность преобразований с многоточием.

Стандартная последовательность имеет три ранга: точное соответствие, продвижение и преобразование. Точное соответствие означает отсутствие необходимости преобразования и является предпочтительным рангом. Это также может означать преобразование левостороннего (`lvalue`) аргумента в правосторонний (`rvalue`).

Продвижение означает расширение представления типа. Например, объект типа `short` может быть продвинут до объекта типа `int` (такое преобразование называется целочисленным продвижением), а объект типа `float` может быть продвинут до объекта типа `double`, что известно как продвижение с плавающей точкой.

Преобразования отличаются от продвижения возможностью изменения значения, что может отрицательно сказаться на точности. Например,

значение с плавающей точкой можно преобразовать в целое число, округлив до ближайшего целого. Кроме того, целочисленные значения и значения с плавающей точкой, перечисления без указания области видимости, указатели и типы указателей на члены могут быть преобразованы в логическое значение. Эти три ранга являются концепциями, унаследованными от языка C, и от них невозможно отказаться из-за необходимости поддерживать совместимость с C.

Это частично охватывает стандартные последовательности преобразований. Преобразования, определяемые пользователем, выполняются двумя способами: либо с помощью неявного конструктора, либо с помощью неявного оператора преобразования. Именно этот тип преобразований мы ожидаем в нашем примере: наш аргумент `char const*` преобразуется в `std::string` через неявный конструктор, который принимает `char const*`. Это так же очевидно, как нос на вашем лице. Но с какой целью мы втянули вас в это обсуждение особенностей разрешения перегрузок?

## ВЕРНЕМСЯ К ПРИМЕРУ

В примере выше клиент ожидает, что к аргументу `char const*` будет применено определяемое пользователем преобразование в `std::string`, и этот временный правосторонний (rvalue) аргумент будет передан в виде ссылки на константу в третьем параметре второй функции.

Однако, как отмечалось выше, стандартные преобразования имеют приоритет перед определяемыми пользователем. В предыдущем разделе, описывая преобразования, мы выяснили, что имеется стандартное преобразование из указателя в логическое значение. Если вы когда-либо рассматривали старый код, передающий простые указатели между функциями, то наверняка видели такие конструкции:

```
if (ptr) {
    ptr->do_thing();
}
```

Условное выражение в операторе `if` является указателем, а не логическим значением, но указатель может быть преобразован в `false`, если он равен нулю. Это более краткий идиоматический способ записи:

```
if (ptr != 0) {
    ptr->do_thing();
}
```

В современном C++ мы все реже видим простые указатели, тем не менее следует помнить, что это совершенно нормальное и разумное преобразование. Именно это стандартное преобразование компилятор сочтет более предпочтительным и выберет его вместо, казалось бы, очевидного определяемого пользователем преобразования из `char const*` в `std::string const&`. К удивлению клиента, компилятор вызовет перегруженную версию, которая принимает логическое значение в третьем аргументе.

Чья это ошибка? Ваша или клиента? Если бы клиент записал вызов так:

```
auto eh_office = make_office(24000.f, 200, "Eagle Heights"s);
```

ошибка не возникла бы. Литеральный суффикс сигнализирует о том, что этот объект на самом деле является объектом `std::string`, а не `char const*`. Так что в этом случае явно виноват клиент. Он должен знать о правилах преобразования.

Однако это не оправдывает выбранное вами решение. Вы должны реализовать интерфейс так, чтобы его проще было использовать правильно, чем неправильно. Пропустить литеральный суффикс и тем самым допустить ошибку очень легко. Также подумайте, что случится, если перегруженная версия функции, принимающая логическое значение, будет добавлена *после* определения конструктора, принимающего `std::string const&`. До этого момента клиентский код будет действовать в соответствии с ожиданиями и с литеральным суффиксом, и без него. Но после добавления перегруженной версии компилятор начнет выбирать лучшее преобразование и клиентский код может неожиданно начать действовать не так, как ожидалось.

Кто-то может посчитать этот пример неубедительным и попробовать заменить `bool` на более подходящий тип, например определить перечисление для использования вместо логического значения:

```
enum class floors {one, two};
office make_office(float floor_space, int staff,
    floors floor_count = floors::one);
office make_office(float floor_space, int staff,
    std::string const& building_name);
```

К сожалению, и этот подход не является выходом из спорной ситуации. Был введен новый тип, только чтобы способствовать правильному использованию набора перегруженных функций. Спросите себя, действительно ли такое решение выглядит яснее этого:

```
office make_office(float floor_space, int staff, bool two_floors = false,
    std::string const& building_name = {});
```

Если и этот довод показался вам неубедительным, то спросите себя, что вы будете делать, когда наступит следующий этап расширения предметной области и прозвучит просьба-замечание: «Вообще-то, мы хотели бы иметь возможность давать названия и двухэтажным зданиям».

Вы должны реализовать интерфейс так, чтобы его проще было использовать правильно, чем неправильно.

## ОДНОЗНАЧНАЯ ПРИРОДА АРГУМЕНТОВ ПО УМОЛЧАНИЮ

Преимущество аргумента по умолчанию в том, что любое преобразование сразу становится очевидным. Вы можете видеть, что `char const*` преобразуется в `std::string const&`. Нет никакой двусмысленности в выборе преобразования, потому что оно может произойти только в одном месте.

Кроме того, как упоминалось выше, наличие единственной функции дает больше уверенности, чем перегруженный набор. Если вы выбрали хорошее имя для своей функции и хорошо спроектировали ее, то вашему клиенту не придется задумываться о том, какую версию вызвать. Но, как показывает пример, это проще сказать, чем сделать. Аргумент по умолчанию сообщает клиенту, что функция обладает гибкостью, предоставляет альтернативный интерфейс к своей реализации и гарантирует единство семантики.

Единственная функция также позволяет избежать дублирования кода. Создавая перегруженную версию, вы исходите из самых лучших побуждений. Конечно, это так. Но перегруженные версии действуют немного по-разному, и вы решаете инкапсулировать оставшееся сходство кодов в одной функции, которую вызывают обе перегруженные версии. Однако со временем перегруженные версии начинают во многом перекрываться, потому что становится все труднее отделить фактические различия их применения. В конечном итоге вы столкнетесь с проблемой усложнения сопровождения базы кода по мере разрастания функционала.

Есть одно ограничение. Аргументы по умолчанию должны определяться в обратном порядке по списку параметров. Например, такое объявление будет допустимым:

```
office make_office(float floor_space, int staff, bool two_floors,
                  std::string const& building_name = {});
```



А это — недопустимое:

```
office make_office(float floor_space, int staff, bool two_floors = false,
                  std::string const& building_name);
```

Если последнюю функцию вызвать только с тремя аргументами, то становится невозможно однозначно сказать что-либо о последнем аргументе, с каким параметром он должен быть связан: с `two_floors` или `building_name`?

Надеемся, мы смогли убедить вас, что перегрузку функций, несмотря на ее неоспоримые достоинства, не следует воспринимать поверхностно. Мы лишь слегка коснулись проблем разрешения перегрузки. Есть еще множество тонкостей, которые нужно изучить, если вы хотите по-настоящему понять, какая из перегруженных версий будет выбрана. Обратите внимание, что мы не рассматривали последовательности преобразований с многообразием и не обсуждали, что произойдет, если добавить в описанную схему шаблонную функцию. Однако если вы абсолютно уверены в необходимости использовать перегруженные версии, то мы вас убедительно просим: пожалуйста, не смешивайте аргументы по умолчанию с перегруженными функциями. Такая смесь трудно поддается анализу и расставляет ловушки для неосторожных. Это не тот стиль определения интерфейса, который проще использовать правильно, чем неправильно.

## АЛЬТЕРНАТИВЫ ПЕРЕГРУЗКЕ

Перегрузка функций сигнализирует клиенту, что доступ к части функциональности, абстракции, можно обеспечить несколькими способами. Функции с одним и тем же идентификатором можно вызвать с разными наборами аргументов. На самом деле, вопреки ожиданиям, фундаментальный строительный блок API был описан как набор перегруженных функций, а не как функция.

Однако в несколько надуманном примере для этой главы можно обнаружить, что набор перегруженных функций:

```
office make_office(float floor_space, int staff, floors floor_count);
office make_office(float floor_space, int staff,
                  std::string const& building_name);
```

не так очевиден, как набор отдельных функций:

```
office make_office_by_floor_count(float floor_space, int staff,
                                 floors floor_count);
office make_office_by_building_name(float floor_space, int staff,
                                   std::string const& building_name);
```

Перегрузка функций — хороший инструмент, но его следует использовать с осторожностью. Это классный молоток, но им нельзя почистить апельсин. Идентификаторы, определяемые вами, должны указываться как можно точнее.

О перегрузке можно рассказывать очень долго — например, для выбора наилучшей из перегруженных версий процесс ранжирования имеет довольно длинный список тай-брейков; если бы это был учебник, мы бы подробно описали их все. Но в данном случае достаточно предупредить, что к перегрузке нельзя относиться легкомысленно.

## ИНОГДА БЕЗ ПЕРЕГРУЗКИ НЕ ОБОЙТИСЬ

Описываемая рекомендация начинается словами: «Если есть выбор». Иногда может не быть возможности определить функцию с другим именем.

Например, может быть только один идентификатор конструктора. Поэтому если потребуется дать возможность создавать экземпляры класса несколькими способами, то вам действительно придется реализовать перегруженные версии конструктора.

Точно так же и операторы могут иметь единственное значение, очень ценное для ваших клиентов. Если по какой-то причине вы написали свой класс строк, то ваши клиенты предпочтут объединять строки таким способом:

```
new_string = string1 + string2;
```

а не:

```
new_string = concatenate(string1, string2);
```

То же верно в отношении операторов сравнения. Однако маловероятно, что при перегрузке операторов вам понадобится аргумент по умолчанию.

Стандарт предоставляет точку настройки `std::swap` и ожидает, что вы напишете перегруженную версию этой функции, оптимальную для вашего класса. В *Core Guidelines* имеется рекомендация «C.83. Для типов-значений желательно определить функцию `swap` со спецификатором `noexcept`», а она прямо предлагает создать перегруженную функцию. Однако и в этом случае крайне маловероятно, что при перегрузке функции понадобится аргумент по умолчанию.

Конечно, иногда просто нет доступных аргументов по умолчанию.

Итак, если вы *должны* выполнить перегрузку, делайте это сознательно и, повторим еще раз, *не* смешивайте аргументы по умолчанию с перегрузкой. В проектировании API это сравнимо с жонглированием заведенной бензопилой.

## ПОДВЕДЕМ ИТОГ

Мы рассмотрели, как влияет рост базы кода на архитектуру API, исследовали простой пример перегрузки и увидели, что именно при этом может пойти не так. В главе, посвященной тонкостям перегрузки, мы кратко пробежались по правилам выбора перегруженной версии компилятором. С учетом работы по этим правилам мы показали, что может состояться вызов совсем не той из перегруженных версий, которая ожидалась. В частности, ошибка в нашем примере была вызвана предоставлением логического параметра с аргументом по умолчанию в перегруженной версии, что открывает широкие возможности для преобразования других нелогических аргументов в этот параметр. Нашей целью было показать, что аргументы по умолчанию предпочтительнее перегрузки функций и смешивание перегрузки с аргументами по умолчанию — весьма рискованное предприятие.

Пример, конечно же, был так себе, но факт остается фактом: для неосторожного инженера перегрузка таит серьезную опасность. Избежать опасности или минимизировать ее можно, разумно используя аргументы по умолчанию и отказавшись от перегрузки. Желающие могут изучить все последствия перегрузки на своем любимом онлайн-ресурсе. Советуем сделать это, если когда-нибудь вы решите проигнорировать нашу вполне конкретную рекомендацию.