Веревка достаточной длины, чтобы… выстрелить себе в ногу

Правила программирования на Си и Си++

Ален И. Голуб

Москва 2001

Программисты, инженеры, научные работники, студенты и все, кто работает с Си или Си++! Если вы хотите писать лучший код без блужданий по лабиринтам технической документации, то это краткое, но содержательное руководство является именно тем, что вам нужно. ***"Веревка достаточной длины, чтобы… выстрелить себе в ногу"*** предлагает более 100 практических правил, которые вы сможете использовать для создания элегантного, простого в сопровождении кода. А так как книга написана признанным знатоком в этой области, то и вы в ней не заблудитесь.

Ален Голуб предлагает необходимый набор пояснений, советов и технических приемов с целью помочь вам полностью использовать возможности этих чрезвычайно мощных языков. Но не бойтесь встретить очередное скучное руководство по программированию. Автору удается сделать изложение столь серьезной темы живым и интересным за счет рассыпанного по тексту юмора и глубокого знания предмета.

Голуб рассматривает универсальные правила, форматирование и правильную организацию программ перед тем, как углубиться в такие основополагающие вопросы, как:

- Практические способы организации и написания сопровождаемого кода.
- Объектно-ориентированное программирование и методы абстракции данных.
- Как избежать проблем, специфических для Си и Си++.

Для закрепления правил, намеченных в этой книге в общих чертах, предлагается множество примеров. Вы также найдете здесь полезные проектные стратегии, освещение вопросов административного управления и многое другое.

Посвящается Аманде

Содержание

Благодарности

Работа над этой книгой затянулась, и я весьма обязан трем редакторам издательства McGraw-Hill, которые по очереди мирились с постоянными задержками с моей стороны: Нэйлу Ливайну, Дэну Гонно и Дженифер Холт-Диджованна. Я особенно признателен Бобу Дюшарму, который защитил меня от самого себя, сделав очень тщательный просмотр первоначального наброска. Его советы значительно улучшили книгу в ее нынешнем виде.

Введение

Название этой книги отражает то, что я считаю основной трудностью при работе как с Си++, так и с Си: эти языки дают вам столько гибкости, что если у вас нет желания и способности призвать себя к порядку, то в итоге вы можете получить гигантский модуль не поддающейся сопровождению тарабарщины, притворяющейся к тому же компьютерной программой. Вы можете поистине делать все при помощи этих языков, даже если вы этого не хотите. В этой книге делается попытка дать средство для преодоления этой трудности в виде собрания практических правил программирования на Си++ и Си — правил, которые, надеюсь, уберегут вас от неприятностей, если вы будете их использовать с самого начала. Хотя большинство из приводимых здесь правил применимы равно при программировании как на Си, так и на Си++, я включил много материала, относящегося лишь к миру Си++ и сконцентрированного по мере возможности в заключительном разделе. Если вы программируете лишь на Си, то просто игнорируйте материал по Си++, встречающийся вам в более ранних разделах.

Я профессионально занимаюсь программированием примерно с 1979 года и ежедневно пользуюсь правилами из этой книги. Я не утверждаю, что эти правила безусловны, или даже "верны". Однако я могу сказать, что они отлично мне служили все это время. Хотя эта книга не относится к категории путеводителей по "ловушкам и рытвинам", многие из этих правил предохранят вас от неприятностей того сорта, который обсуждается в путеводителях по "ловушкам и рытвинам".

Практические правила по своей сути гибки. Они постепенно меняются с ростом опыта, и ни одно правило не действует постоянно. Тем не менее я предупреждаю вас с самого начала, что мое мнение относительно этого материала самое наилучшее и что я не очень симпатизирую неряшливым мыслям или небрежному программированию. Я не извиняюсь за усиленное подчеркивание тех вещей, в которые я сильно верю. Мои мнения всегда могут измениться, если, конечно, вы сможете убедить меня

в том, что я не прав, но имейте в виду, что эта книга основана на опыте, а не на теории. Я сознаю, что большая часть этой книги подходит опасно близко к чьему-то культу и многие вещи, произносимые мной, дискуссионны, но думаю, что всегда имеется возможность разумного разговора двух людей, объединенных целью совершенствования своего мастерства.

Я часто читаю курсы по Си++ и объектно-ориентированному проектированию как по приглашению частных фирм, так и в Калифорнийском университете в Беркли. Эта книга появилась в ответ на просьбы моих студентов, большинство из которых увлеченные профессионалы с настоящим желанием изучить этот материал. Я вижу множество программ в процессе проверки домашних заданий, и эти программы достаточно репрезентативны в качестве произведений сообщества профессиональных программистов из района залива Сан-Франциско. К несчастью, каждый семестр я также вижу, что одни и те же проблемы повторяются снова и снова. Поэтому эта книга является некоторым образом и списком распространенных проблем, найденных мной в созданных настоящими программистами реальных программах, сопровождаемым моими советами по их решению.

Обсуждаемые здесь проблемы программирования и проектирования не ограничиваются, к несчастью, лишь ученическими программами. Многие из примеров того, что не следует делать, взяты из коммерческого продукта: библиотеки классов Microsoft Foundation Classes (MFC) корпорации Microsoft. Я могу сказать, что эта библиотека была спроектирована без заботы о удобстве сопровождения людьми, не подозревающими о существовании даже элементарных принципов объектно-ориентированного проектирования. Я не выделял явно большинство примеров этого в тексте, так как это не книга с названием "Что неправильно в MFC"; пользователи библиотеки MFC узнают ее код, когда натолкнутся на него. Я выбрал примеры из MFC просто потому, что мне пришлось много с ней работать и очень близко познакомиться с ее недостатками. Во многих других коммерческих библиотеках классов имеются сходные проблемы.

Наконец, эта книга не является введением в Си++. Обсуждение, сопровождающее относящиеся к Си++ правила, предполагает, что вы знаете этот язык. Я не расходую место на описание того, как работает Си++. Имеется множество хороших книг, которые учат вас языку Си++, включая мою собственную *"C+C++"* (New York: McGraw-Hill,1993). Вы должны также ознакомиться с принципами объектно-ориентированного проектирования. Я рекомендую второе издание книги Гради Буча *"Object-Oriented Analysis and Design with Applications"* (Redwood City: Benjamin

Cummings,1994)[♣].

О нумерации правил: иногда я группировал некоторые правила вместе, потому что удобно описывать их все одновременно. В этом случае все эти правила (имеющие различные номера) располагаются в начале раздела. Я использовал запись номера правила вида "1.2" в случаях, когда оно является особым случаем другого правила.

[♣] Буч Г. Объектно–ориентированный анализ и проектирование с примерами приложений на C++, 2–е изд./Пер. с англ.—М.; СПб.: "Издательство БИНОМ" — "Невский диалект", 1998.—560 с.—*Прим. перев*.

Часть

1

Процесс проектирования

Эта часть вместе с последующей, посвященной разработке, являются наиболее туманными в этой книге. Правила здесь довольно общего характера по своей природе, они совсем не затрагивают техники программирования на Си или Си++, а скорее рассматривают более общий процесс проектирования и разработки программы.

Правила из данной части относятся к процессу общего проектирования. После прочтения этой части в законченном виде я стал беспокоиться, что многие из этих правил будут казаться банальными. Несмотря на это, некоторые из приводимых здесь правил являются самыми важными в этой книге, потому что нарушение их может вызвать много бед в процессе разработки. В известном смысле, большинство правил этой части предназначены для управленцев; программисты их часто знают, но у них нет свободы, необходимой, чтобы воспользоваться своими знаниями.

1. Сущность программирования: без сюрпризов, минимум сцепления и максимум согласованности

Многие (если не все) правила в этой книге могут быть объединены в три метаправила (при желании), выраженные в заголовке этого раздела.

Правило "без сюрпризов" не требует пояснений само по себе. Пользовательский интерфейс должен действовать так, как кажется он должен действовать. Функция или переменная должны делать то, что означают их имена.

Сцепление — это связь между двумя программами или объектами пользовательского интерфейса. Когда один объект меняется, то все, с чем он соединен, может также измениться. Сцепление вызывает сюрпризы. (Я меняю эту штучку здесь, и внезапно та штуковина вон там перестает работать). Пример из Си++: если объект одного класса посылает сообщение объекту второго класса, то посылающий класс сцеплен с принимающим классом. Если вы меняете интерфейс для принимающего класса, то вы также должны исследовать код в посылающем классе, чтобы убедиться в том, что он еще работает. Этот вид слабого сцепления безвреден. Вам нужно знать об отношениях сцепления для сопровождения программы, но без некоторого количества сцеплений программа не могла бы работать. Несмотря на это, для вас желательно по мере возможности минимизировать число отношений сцепления.

Эта минимизация обычно выполняется в Си посредством модулей, а в Си++ посредством классов. Функции в модуле (функции-члены в классе) сцеплены друг с другом, но за исключением нескольких интерфейсных функций (или объектов) они вовсе не сообщаются с внешним миром. В Си вы должны использовать *статический* класс памяти, чтобы ограничить использование функции одним модулем. В Си++ вы используете *закрытые* функции-члены.

Согласованность является противоположностью сцепления; сущности, которые группируются вместе (пункты диалогового и простого меню, функции в модуле, или члены класса), должны быть связаны по назначению. Отсутствие связности также является "сюрпризом". У текстового редактора, которым я пользуюсь, имеется в меню пункт "Настройка" и, кроме того, дополнительные опции настройки рассыпаны по четырем другим всплывающим меню. Я ожидал согласованной конфигурации и, когда не смог найти нужную мне опцию в пункте "Настройка", то решил, что этой опции просто нет. Эта плохо спроектированная система до сих пор доставляет беспокойство; после года пользования я по-прежнему не помню, где расположена каждая опция, и часто вынужден тратить раздражающие пять минут на поиск в пяти разных

местах того, что хотел изменить. По отношению к исходному коду отсутствие согласованности заставляет вас делать то же самое — тратить свою жизнь на поиск объявлений функций в 15 различных файлах, что является очевидной проблемой при сопровождении.

2. Подавляйте демонов сложности (часть 1)

Ричард Рашид (разработчик Mach — варианта ОС UNIX) выступил несколько лет назад с основным докладом на конференции разработчиков Microsoft. Его главный смысл состоял в том, что слишком большая сложность как в пользовательском интерфейсе, так и в программе является единственной большой проблемой, стоящей перед проектировщиками и пользователями программного обеспечения. По иронии, его речь была произнесена спустя два дня после провалившейся попытки показать нескольким тысячам очень толковых программистов, как программировать разработанный Microsoft интерфейс OLE 2.0 — один из самых сложных интерфейсов прикладного программирования, когда-либо мной виденных. (OLE означает "связь и внедрение объекта". Стандарт OLE 2.0 определяет интерфейс, который может использоваться двумя программами для взаимодействия между собой определенным образом. Это действительно объектная ориентация на уровне операционной системы).

Предыдущий оратор, который убеждал нас пользоваться библиотекой Microsoft Foundation Classes (MFC), сказал нам, что поддержка OLE в MFC "включает 20000 строк кода, *необходимых* для каждого базового приложения OLE 2.0". Аудитория была ошеломлена не полезностью MFC, а тем фактом, что для написания базового приложения OLE 2.0 требуется 20000 строк кода. Любой интерфейс такой сложности таит в себе изъян. Следующие несколько правил используют OLE для показа характерных проблем, но не думайте, что проблема запутанности характерна лишь для Microsoft — она свойственна всей отрасли.

2.1. Не решайте проблем, которых не существует

2.2. Решайте конкретную проблему, а не общий случай

Поучительно использовать OLE 2.0 как пример того, что случается со многими слишком сложными проектами. Имеется две главные причины сложности интерфейса OLE. Во-первых, он безуспешно пытается быть независимым от языка программирования. Идея таблицы виртуальных функций Си++ является центральной для OLE 2.0. Спецификация OLE даже пользуется нотацией классов Си++ для документирования того, как

должны работать различные интерфейсы OLE. Для реализации OLE на другом языке программирования (не Си++) вы должны имитировать на этом языке таблицу виртуальных функций Си++, что фактически ограничивает ваш выбор Си++, Си или языком ассемблера (если вы не разработчик компиляторов, который может добавить к выбранному вами языку нужные свойства). Если честно, то вы должны быть сумасшедшим, чтобы программировать OLE не на Си++; потребуется гораздо меньше времени на изучение Си++, чем на написание имитатора Си++. То есть, эта идея независимости от языка программирования является неудачной. Интерфейс мог бы быть существенно упрощен за счет отказа от нее.

Возвращаясь к истории из предыдущего раздела, нужно заметить, что библиотека MFC в действительности решает проблему сложности, связанную с OLE, при помощи простого и легко понятного интерфейса, реализующего все возможности, нужные для большинства приложений OLE 2.0. Тот факт, что никто не хотел программировать с использованием OLE, пока для этого не появилась оболочка на основе MFC, впечатляет. Разработка хорошей оболочки вокруг плохого интерфейса не может быть решением лежащей в основе проблемы.

Если оболочка с использованием MFC столь проста, то почему лежащий в основе пласт так сложен? Ответ на этот вопрос является основным предметом проектирования. Создатели интерфейса OLE никогда не задавали себе два основных вопроса:

- Какие основные возможности должно поддерживать настоящее приложение?
- Как реализовать эти возможности простейшим способом?

Другими словами, они имели в виду не реальное приложение, когда они проектировали этот интерфейс, а какой-то теоретический худший случай. Они реализовали самый общий интерфейс из возможных, не думая о том, что *на самом деле* предполагается делать при помощи этого интерфейса, и получив в результате систему, которая может делать все, но при этом слишком сложная, чтобы быть пригодной для использования. (Вероятно, они совсем не пробовали реализовать этот интерфейс в каком-либо приложении, иначе они бы обнаружили эти проблемы).

Процесс объектно-ориентированного проектирования является в какой-то мере попыткой решения этой проблемы. Относительно просто добавить новую возможность в объектно-ориентированную систему или посредством наследования, или добавив новых обработчиков сообщений к существующим классам. Скрывая определения данных от пользователя класса, вы оставляете за собой право полностью менять внутреннюю организацию класса, включая определения данных, не беспокоя

пользователей этого класса, при условии, что вы сохраняете его существующий интерфейс.

В структурном проектировании вам не нужна такая роскошь. Вы обычно проектируете сперва структуры данных, и модификация структуры данных является серьезным делом, потому что нужно проверить каждую подпрограмму, использующую эту структуру данных, чтобы убедиться в том, что она еще работает. Как следствие, "структурные" программы склонны иметь много ничего не делающего кода. Это потому, что кто-нибудь может захотеть воспользоваться некой возможностью в будущем. На деле многие проектировщики структурных программ горды своей способностью предсказывать направление, в котором может развиваться программа. Все это приводит к большому объему ненужной работы и программам, имеющим больший размер, чем необходимо.

Вместо того, чтобы учитывать в проекте все возможные случаи, проектируйте свой код так, чтобы он мог быть легко расширен при необходимости добавления новых возможностей. Объектно-ориентированные проекты, как правило, тут работают лучше.

3. Интерфейс пользователя не должен быть похожим на компьютерную программу (принцип прозрачности)

Я однажды слышал, как кто-то сказал, что лучшим пользовательским интерфейсом из когда-либо разработанных является карандаш. Его назначение тотчас же понятно, для него не нужно руководство пользователя, он готовится к работе без особой суеты. Однако наиболее важным свойством является прозрачность. Когда вы пользуетесь карандашом, то думаете о том, что вы пишите, а не о самом карандаше.

Подобно карандашу, лучшими компьютерными интерфейсами являются те, которые скрывают сам факт того, что вы обращаетесь к компьютеру: замечательный пример — интерфейс с системой зажигания вашего автомобиля. Вы поворачиваете зажигание, включаете скорость и жмете на газ, как если бы все эти объекты интерфейса (ключ, рычаг скоростей, педаль) были прицеплены прямо на двигатель. Тем не менее, это не так: они теперь обычно просто устройства ввода в компьютер, который управляет двигателем.

К сожалению, подобный уровень ясности часто отсутствует в пользовательских интерфейсах. Представьте графический интерфейс пользователя Windows на автомобиле. Вы трогаетесь, выбрав в главном меню пункт "Движение автомобиля". Щелчок по нему откроет меню "Переключение скорости", которое предложит вам выбор из опций "Вперед", "Назад" и "Нейтральная". Щелкните по одной из них, чтобы

передвинуть флажок на нужное вам направление. Затем вернитесь в меню "Движение автомобиля" и выберите команду "Поехали". Это вызовет появление диалогового окна "Скорость", где вы должны использовать ползунок для ввода желаемой скорости. Однако установить скорость правильно трудно вследствие высокого разрешения ползунка (пол-миллиметра движения мыши соответствует примерно 1 км/ч), поэтому вы скорее установите 59,7 км/ч вместо 60. Затем вы нажимаете кнопку "Поехали" в диалоговом окне, вслед за чем появляется сообщение "Стояночный тормоз не убран — нажмите F1 для справки" (динамик издает громкий звук). Вы покорно щелкаете по кнопке "ОК", чтобы убрать окно сообщений, затем снова пытаетесь открыть главное меню, но машина просто посылает вам звуковой сигнал. Наконец, поняв, что дело в том, что диалоговое окно "Скорость" еще отображается, вы щелкаете по кнопке "Отмена", чтобы убрать его. Вы открываете меню "Стояночный тормоз" и убираете флажок "Включен". Затем вы снова открываете окно "Поехали". И вновь получаете сообщение (и громкий звук) о том, что вы должны сначала выбрать направление в меню "Переключение скорости". В этот момент вы решаете, что вам, может быть, лучше пройтись на работу пешком.

Вот другой пример: занимаясь недавно подготовкой обзора, я просмотрел несколько программ авиационных бортовых журналов. ("Бортовой журнал" — это очень простой табличный документ. Каждая строка соответствует отдельному вылету, а столбцы разбивают общую продолжительность вылета на различные категории: итоговая продолжительность, продолжительность полета в облаках и т.п.. В других столбцах полет помечается как деловой и так далее).

Самый лучший интерфейс из всех был тот, который выглядел совершенно одинаково с привычным бумажным журналом, но автоматизировал нудную работу. Вы вводили время в "итоговый" столбец — и то же самое время появлялось в других подходящих по смыслу столбцах. Значения по столбцам складывались автоматически для получения итогов по категориям. Вы могли легко генерировать необходимые отчеты и экспортировать данные в формат ASCII с разделителями из символов табуляции, который читается любой электронной таблицей или текстовым редактором. Для непривычного взгляда весь интерфейс казался, мягко говоря, разочаровывающим, но он был функциональным и интуитивно понятным, а программа — маленькой и быстрой. Однако самым важным было то, что этот интерфейс выглядел как бортовой журнал, а не как программа для Windows.

Другой крайностью был ошеломляющий графический интерфейс пользователя Windows: у него были диалоговые окна; у него была трехмерная графика; вы могли генерировать круговые диаграммы,

показывающие процент продолжительности полета в облаках по отношению к вашему общему налету на "Цесне–172" за последние 17 лет; вы могли помещать внутрь отсканированную фотографию самолета…— вы представили эту картину? Программа выглядела превосходно, но ее было почти невозможно использовать. Не было практической причины для создания большинства диаграмм и отчетов, которые она могла генерировать. Ввод данных был неудобный и медленный — вы должны были вызвать диалоговое окно с полями, разбросанными по всей его поверхности. Фактически вы должны были прочитать все, чтобы обнаружить ту категорию, которая вас интересовала, а некоторые из категорий были скрыты за кнопками, неизбежно влеча за собой сложный поиск. Чтобы добавить обиду к оскорблению, эта программа была надстроена над сервером реляционной базы данных (помните, что это для поддержки простой таблицы без реляционных связей). Она заняла 30 Мбайт на моем диске. Мне требовалось почти 5 минут, чтобы сделать запись, которая занимала примерно 10 секунд в бумажном бортовом журнале или упомянутом ранее простом графическом интерфейсе пользователя. Программа была бесполезна, но, конечно, потрясающа.

Одна из главных проблем заключалась в том, что инструменты, использованные для создания второй программы, перегружают проектирование интерфейса. Все эти программы были разработаны на языке очень высокого уровня Visual Basic (который мне на самом деле не очень нравится, между прочим). Приложения, созданные при помощи таких построителей приложений, как Visual Basic (или Power Builder, или Delphi, или …) обычно имеют специфический внешний вид, который немедленно говорит вам, что за инструмент был использован для построения этого приложения. Проектировщику интерфейса некуда обратиться за помощью, если этот специфический вид не подходит для конкретного проекта. Пользователи генераторов приложений должны иметь их несколько на выбор, чтобы затем использовать тот, который лучше всего соответствует потребностям данного интерфейса. Несмотря на это, мой опыт показывает, что наиболее практические программы со временем (в конце концов) должны перенести, по меньшей мере, часть кода интерфейса на язык низкого уровня типа Си или Си++, поэтому важно, чтобы ваш генератор приложений был способен также использовать низкоуровневый код.

4. Не путайте легкость в изучении с легкостью в использовании

Эта проблема когда-то касалась почти исключительно машин Macintosh, но Windows и здесь в последнее время выходит вперед. Компьютер Mac был спроектирован так, чтобы прежде всего быть простым в освоении. Положим, что тетушка Матильда Мак-Гиликатти часто заходила в компьютерный магазин, чтобы пользоваться их услугой по моментальной печати кулинарных рецептов. В итоге Матильда забирает компьютер домой и успешно вводит рецепты в течение нескольких месяцев. Теперь она хочет взять эти рецепты, проанализировать их химический состав и написать статью в научный журнал о коллоидных свойствах продуктов питания на основе альбумина. Доктор Мак-Гиликатти — хорошая машинистка, печатающая обычно около 100 слов в минуту, но эта ужасная мышь ее постоянно тормозит. Каждый раз, когда ее руки отрываются от клавиатуры, она теряет несколько секунд. Она пытается найти слово в своем документе и обнаруживает, что для этого должна открыть меню, ввести текст в диалоговое окно и щелкнуть по нескольким экранным кнопкам. В конце файла она должна явно указать утилите поиска возвратиться к его началу. (Ее версия редактора vi 15-летней давности позволяет выполнить все это при помощи двух нажатий клавиш — без необходимости отрываться от клавиатуры). Наконец, она обнаруживает, что на выполнение обычной работы — подготовки статьи в журнал — уходит в два раза больше времени, чем раньше, в основном из-за проблем с пользовательским интерфейсом. Ей не понадобилось руководство, чтобы пользоваться этой программой, — ну и что?

Вернемся к примеру с карандашом из предыдущего параграфа. Очень трудно научиться пользоваться карандашом. У большинства детей это занимает несколько лет. (Вы могли бы возразить, что, судя по каракулям на рецептах, многие врачи этому так и не научились). С другой стороны, после того, как вы научились, карандашом пользоваться очень легко.

Главная проблема здесь состоит в том, что для опытного пользователя часто требуется совершенно другой интерфейс, чем для начинающего. Дополнительная помощь типа "горячих" клавиш не решает эту проблему; старый неуклюжий интерфейс пользователя все еще мешает продуктивности, и нет особой разницы: откроете ли вы меню при помощи "горячей" клавиши, или мышью. Здесь проблема в самом меню.

5. Производительность может измеряться числом нажатий клавиш

Интерфейс, требующий меньше нажатий клавиш (или других действий пользователя типа щелчков мышью), лучше того, который требует много нажатий для выполнения одной и той же операции, даже если такие виды интерфейсов обычно сложнее в освоении.

Подобным образом пользовательскими интерфейсами, скрывающими информацию в меню или за экранными кнопками, обыкновенно труднее пользоваться, потому что для выполнения одной задачи необходимо выполнить несколько операций (вызвав подряд несколько спускающихся меню). Хороший пример — настройка программы. Во многих из используемых мной ежедневно программ опции настройки рассыпаны по нескольким меню. То есть для вызова диалогового окна, настраивающего один из аспектов того, что делает программа (например, выбор шрифта), я должен выбрать одно меню. Затем я должен вызвать другое меню, чтобы сделать что-то в том же духе (например, выбрать цвет). Лучше поместить все опции настройки на одном экране и использовать форматирование экрана для объединения опций по назначению.

6. Если вы не можете сказать это по-английски, то вы не сможете выполнить это и на Си/Си++

Это правило с последующим также относятся к правилам пользовательского интерфейса, но здесь под "пользователем" уже понимается программист, использующий написанный вами код — часто это вы сами.

Акт записи на английском языке описания того, что делает программа, и что делает каждая функция в программе, является критическим шагом в мыслительном процессе. Хорошо построенное, грамматически правильное предложение — признак ясного мышления. Если вы не можете это записать, то велика вероятность того, что вы не полностью продумали проблему или решение. Плохая грамматика и построение предложения являются также показателем небрежного мышления. Поэтому первый шаг в написании любой программы — записать то, что делает программа, и как она это делает.

Есть разные мнения о возможности мышления вне языка, но я убежден, что аналитическое мышление того типа, который нужен в компьютерном программировании, тесно связано с языковыми навыками. Я не думаю, что является случайностью то, что многие из знакомых мне лучших программистов имеют дипломы по истории, филологии и схожим наукам.

Также не является случайностью то, что некоторые из виденных мной худших программ были написаны инженерами, физиками и математиками, затратившими в университете массу энергии на то, чтобы держаться как можно дальше от занятий по языку и литературе.

Сущность заключается в том, что математическая подготовка почти не нужна в компьютерном программировании. Тот тип организационного мастерства и аналитических способностей, который нужен для программирования, связан полностью с гуманитарными науками. Логика, например, преподавалась на философском факультете, когда я был в университете. Процесс, используемый при проектировании и написании компьютерных программ, почти полностью идентичен тому, который используется, чтобы сочинять и писать книги. Процесс программирования совсем не связан с теми процессами, которые используются для решения математических уравнений.

Здесь я делаю различие между информатикой (*computer science*) — математическим анализом компьютерных программ — и программированием или разработкой программного обеспечения — дисциплиной, интересующейся написанием компьютерных программ. Программирование требует организационных способностей и языковой подготовки, а не абстрактного мышления, необходимого для занятий математическим анализом. (В университете меня заставили проходить год на лекции по математическому анализу, но я никогда из него ничего не использовал ни на занятиях по информатике, хотя для них матанализ был необходимым условием, ни в реальной жизни).

Я как-то получил открытую рецензию на книгу, посвященную мной предмету проектирования компиляторов, в которой рецензент (который преподавал в одном из ведущих университетов) заявил, что он "считает абсолютно неуместным включение исходного кода компилятора в книгу о проектировании компиляторов". По его мнению, необходимо учить "фундаментальным принципам" — лежащей в основе математике и теории языка, а детали реализации — "тривиальны". Первое замечание имеет смысл, если у вас создалось впечатление, что книга написана ученым-специалистом по информатике, а не программистом. Рецензент интересовался лишь анализом компилятора, а не тем как его написать. Второе замечание просто показывает вам, насколько изолировала себя научная элита от реального труда программирования. Интересно, что основополагающая работа по теории языка, сделавшая возможным написание компиляторов, была выполнена в Массачусетском технологическом институте лингвистом Наумом Хомским, а не математиком.

Обратной стороной этой медали является то, что если вы зашли в тупик при решении проблемы, один из лучших способов выйти из него — это объяснить проблему приятелю. Почти всегда решение возникает в вашей голове посредине объяснения.

6.1. Начинайте с комментариев

Если вы последовали совету в предыдущем правиле, то комментарии для вашей программы уже готовы. Для того, чтобы получить документированное описание реализации, вы просто писали и добавляли вслед за каждым абзацем блоки кода, реализующие качества, описанные в этом абзаце. Оправдание "у меня не было времени, чтобы добавить комментарии" на самом деле означает "я писал этот код без проекта системы и у меня нет времени воспроизвести его". Если создатель программы не может воспроизвести проект, то кто же сможет?

7. Читайте код

Все писатели — это читатели. Вы учитесь, когда смотрите, что делают другие писатели. Удивительно, но программисты — писатели на Си++ и Си — часто не читают код. Тем хуже. Я настоятельно рекомендую, чтобы, как минимум, члены группы программирования читали код друг друга. Читатель может найти ошибки, которые вы не увидели, и подать мысль, как улучшить код.

Идея здесь — не формальная "критика кода", имеющая довольно сомнительный характер: никто не хочет наступать на ногу коллеге, поэтому шансы получить полезную обратную связь в формальной ситуации малы. Для вас лучше присесть с коллегой и просто разобрать код строка за строкой, объясняя что как делается и получая какую-то обратную связь и совет. Для того, чтобы подобное упражнение принесло пользу, автор кода не должен делать никаких предварительных пояснений. Читатель должен быть способен понимать код, читая его. (Нам всем приходилось иметь дело с учебниками, столь трудными для понимания, что ничего нельзя было понять без объяснения преподавателя. Хотя это и гарантирует, что преподаватель не останется без работы, но никак не отражается на авторе учебника). Если вам пришлось объяснять что-то вашему читателю, то это значит, что ваше объяснение должно было быть в коде в виде комментария. Добавьте этот комментарий, как только вы его произнесли; не откладывайте этого до окончания просмотра.

7.1. В цехе современных программистов нет места примадоннам

Это следствие из правила чтения. Программисты, которые думают, что их код совершенен, которые отвергают критику, вместо того, чтобы считать ее полезной, и которые настаивают на том, что они должны работать втихомолку, вероятно, пишут тарабарщину, не поддающуюся сопровождению — даже если кажется, что она работает. (Смысловое ударение здесь на слове *кажется*).

8. Разбивайте сложные проблемы на задачи меньшего размера

На самом деле это также правило и литературного стиля. Если концепцию слишком сложно объяснить за один раз, то разбейте ее на меньшие части и объясняйте каждую по очереди. То же назначение у глав в книге и параграфов в главе.

Как пример, связанный с программированием, возьмем прошитое бинарное дерево, отличающееся от нормального дерева тем, что указатель на узел-потомок в концевом узле указывает на само дерево. Действительным преимуществом прошитого дерева является то, что его легко пересечь нерекурсивно при помощи этих дополнительных указателей. Проблема заключается в том, что сложно выйти из алгоритмов пересечения (в особенности обратного пересечения). С другой стороны, имея указатель на узел, легко написать алгоритм поиска последующего элемента в обратном порядке. Путем изменения формулировки с "выполнить пересечение в обратном порядке" на "начав с самого отдаленного узла, искать последующие элементы в обратном порядке, пока они не закончатся" получаем разрешимую задачу:

```
tree t;                                 // дерево
node = postorder_first( t );            // исходный узел
while( node )                           // есть еще узлы?
   node = postorder_successor( t ); // следующий узел-родитель
```

9. Используйте весь язык

9.1. Используйте для работы соответствующий инструмент

Данное правило является спутником правила "Не путайте привычность с читаемостью", представленного ниже, но скорее больше касается проблем руководства. Мне часто говорят, что студентам не разрешается использовать некоторые части Си или Си++ (обычно это указатели),

потому что они "нечитаемы". Обычно это правило навязывается руководителями, знающими ФОРТРАН, БЕЙСИК или какой-то другой язык, не поддерживающий указатели, ибо их не очень-то заставишь изучать Си. Вместо того, чтобы допустить, что их знания недостаточны, такие руководители будут лучше калечить своих программистов. Указатели отлично читаемы для программистов на Си.

И наоборот, я видел ситуации, где руководство требовало, чтобы программисты перешли с языка программирования типа КОБОЛ на Си, но не желало оплачивать переподготовку, необходимую для перехода. Или хуже, руководство платило за переподготовку, но не предоставляло времени, необходимого для действительного изучения материала. Переподготовка является занятием, требующим всего рабочего дня. Вы не можете одновременно выполнять "полезную" работу, а если попытаетесь, то ваши деньги будут выброшены на ветер. Так или иначе, после того, как руководители видят, что их работники не были превращены в гуру программирования на Си++ после 3-дневного краткого курса, они реагируют, накладывая ограничения на использование некоторых частей языка. Фактически говоря "вы не можете использовать ту часть Си++, которая не похожа на язык, который мы использовали до перехода на Си++". Естественно, что будет нельзя эксплуатировать ни одну из прогрессивных особенностей языка — которые прежде всего и являются главной причиной его использования — если вы ограничите себя "простейшим" подмножеством особенностей.

Глядя на эти ограничения, мне в первую очередь интересно знать, зачем было менять КОБОЛ на Си. Принуждение программистов на языке КОБОЛ использовать Си всегда поражало меня своей большой глупостью. КОБОЛ — великолепный язык для работы с базами данных. У него есть встроенные примитивы, упрощающие выполнение задач, которые довольно трудны для Си. Си, в конце концов, был разработан для создания операционных систем, а не систем управления базами данных. Довольно просто дополнить КОБОЛ, чтобы он поддерживал модный графический интерфейс пользователя, если это единственная причина перехода на Си.

10. Проблема должна быть хорошо продумана перед тем, как она сможет быть решена

Это правило с двумя последующими первоначально располагалось в начале этой главы. Подумав, я переместил их сюда, так как побоялся, что, прочитав их, вы пропустите оставшуюся часть главы. Однако в мои намерения не входит чтение проповедей. Эти правила посвящены весьма реальным проблемам и во многих отношениях являются самыми важными

правилами в этой книге.

Настоящее правило является настолько очевидным утверждением в повседневной жизни, что кажется странным его восприятие как едва ли не ереси применительно к программированию. Мне часто говорят, что "невозможно потратить пять месяцев на проектирование, не написав ни одной строки кода — ведь наша производительность измеряется числом строк кода, написанных за день". Люди, говорящее это, обычно знают, как делается хороший проект; просто у них нет этой "роскоши".

Мой опыт говорит, что хорошо спроектированная программа не только работает лучше (или просто работает), но и может быть написана быстрее и быть проще в сопровождении, чем плохо спроектированная. Лишние четыре месяца при проектировании могут сэкономить вам более четырех месяцев на этапе реализации и буквально годы в период сопровождения. Вам не добиться высокой производительности, если приходится выбрасывать прошлогоднюю работу из-за существенных изъянов проекта.

Кроме того, скверно спроектированные программы труднее реализовать. Тот аргумент, что у вас нет времени на проектирование, потому что вы "должны захватить рынок программ как можно скорее", просто не выдерживает никакой критики, потому что реализация плохого (или никакого) проекта требует гораздо больше времени.

11. Компьютерное программирование является индустрией обслуживания

Меня иногда шокирует неуважение, проявляемое некоторыми программистами по отношению к пользователям своих программ, как если бы "пользователь" (произносится с презрительной усмешкой) был низшей формой жизни, неспособной к познавательной деятельности. Но факт состоит в том, что весь компьютер существует лишь с одной целью: служить конечному пользователю наших продуктов. Если никто бы не пользовался компьютерными программами, то не было бы программистов.

Печальным фактом является то, что существенно больше половины разрабатываемого ежегодно кода выбрасывается за ненадобностью. Такие программы или никогда не поступают в эксплуатацию, или используются лишь очень короткое время, после чего выбрасываются. Это означает невероятную потерю производительности, сокращая для большинства управляющих реальные среднесуточные цифры выработки. Подумайте о всех начинающих фирмах, выпускающих программы, которые никогда не будут проданы, о всех внутрифирменных группах разработчиков, пишущих бухгалтерские пакеты, которыми нельзя пользоваться.

Легко увидеть, как возникает эта печальная ситуация: программисты создают программы, которые никому не нужны. Исправить ее тоже легко, хотя это и сталкивается с неожиданными трудностями в некоторых условиях: спросите людей, что им нужно, и затем сделайте то, что они вам сказали.

К сожалению, многие программисты производят впечатление полагающих, что конечные пользователи не знают, чего хотят. Вздор! Почти всегда пользователи оказываются так запуганы сыплющим специальными терминами "экспертом", что замолкают. Мне часто говорили: "Я знаю, что мне нужно, но не могу это выразить". Лучший ответ на это: "Отлично, скажите это на нормальном языке — я сделаю перевод на компьютерный".

12. Вовлекайте пользователей в процесс проектирования

13. Заказчик всегда прав

Ни одной программе не добиться успеха, если ее проектировщики не общаются непосредственно с ее конечными пользователями. Несмотря на это, часто ситуация больше напоминает игру ("испорченный телефон"), в которую многие из нас играли в детском саду и при которой 20 ребятишек садятся в кружок. Кто-нибудь шепчет фразу своему соседу (соседке), который передает ее своему, и так далее по кругу. Забава заключается в том, чтобы послушать, как сообщение звучит после того, как пройдет весь круг — обычно ничего похожего на исходную фразу. Тот же самый процесс часто встречается при разработке программ. Пользователь говорит с управляющим, докладывающим другому управляющему, который нанимает консультационную фирму. Президент консультационной фирмы разговаривает с руководителем разработчиков, который в свою очередь говорит со старшим группы, обращающимся, наконец, к программистам. Шансы на то, что даже простой документ с требованиями останется после этого процесса невредимым, равны нулю. Единственным решением этой проблемы является тесное вовлечение пользователей в процесс разработки, лучше всего путем включения, по крайней мере, одного конечного пользователя в команду разработчиков.

Родственная ситуация складывается в случае простой самонадеянности части программистов, которые говорят: "Я знаю, что пользователи сказали, что им нужно сделать это таким способом, но у них нет достаточных знаний о компьютерах, чтобы принять сознательное решение; мой способ лучше". Такое отношение фактически гарантирует, что программой никогда не будут пользоваться. Исправить ситуацию здесь можно, официально назначив конечного пользователя лицом, оценивающим

качество проекта. Никто не может начать писать код до тех пор, пока пользователь-член команды не даст на это добро. Сотрудники, игнорирующие проект в пользу своих идей, должны быть уволены. В реальной жизни для подобного типа детского упрямства на самом деле нет места.

При этом нужно сказать, что опытный проектировщик зачастую предлагает лучшее решение проблемы, чем то, что придумано конечным пользователем, в особенности, если учесть, что конечные пользователи часто предлагают интерфейсы, созданные по образцу программ, которыми они постоянно пользуются. Несмотря на это, вы должны убедить пользователя, что ваш способ лучше, перед тем, как его реализовать. "Лучший" интерфейс не является лучшим, если никто, кроме вас, не сможет (или не захочет) им пользоваться.

14. Малое — это прекрасно (большое == медленное)

Распухание программ является огромной проблемой. Жесткий диск вместимостью 350 Мбайт на моем лэптопе может вместить операционную систему, усеченные версии моих компилятора и редактора и больше ничего. В стародавние времена я мог разместить версии для CP/M тех же программ на единственной дискете вместимостью 1,2 Мбайта. UNIX в то время спокойно работал на 16-разрядном PDP-11 с 64 Кбайтами ядра (внутренней памяти). В наше время большинство операционных систем требуют 32-разрядных машин с минимум 16 Мбайтами оперативной памяти, чтобы работать с приемлемой скоростью[♣]. Я убежден, что б*о*льшая часть этого распухания памяти является результатом небрежного программирования.

В добавок к проблеме размера у вас также есть проблема со временем выполнения. Виртуальная память не является настоящей памятью. Если ваша программа слишком велика, чтобы поместиться в оперативной памяти, или если она выполняется одновременно с другими программами, то она должна периодически подкачиваться с диска. На эти подкачки, мягко выражаясь, расходуется время. Чем меньше программа, тем менее вероятно, что произойдет подкачка, и тем быстрее она будет выполняться.

Третьей проблемой является модульность. Одна из основ философии UNIX гласит "меньше — лучше". Большие задачи лучше выполняются взаимодействующей системой маленьких модульных программ, каждая из которых делает хорошо лишь одно задание, но каждая из них может сообщаться с другими компонентами. (Стандарт связи и внедрения

[♣] Уже не редкость емкость дисковой памяти, превышающая спустя 5 лет указанные автором значения на два порядка, а оперативной — на порядок. — *Прим.перев*.

объектов Microsoft (OLE) добавляет это свойство в Windows, а OpenDoc — в Macintosh). Если ваше приложение представляет собой модульную конструкцию из маленьких программ, работающих вместе, то становится очень просто настраивать вашу программу по заказу путем смены модулей. Если вам не нравится этот редактор, то поменяйте его на новый.

Наконец, программы обычно уменьшаются в процессе усовершенствования. Большие программы, вероятно, никогда не подвергались усовершенствованиям.

Разыскивая решение этой проблемы, я заметил, что коллективы программистов с плохим руководством часто создают излишне большие программы. То есть группа ковбоев от программирования, каждый из которых работает в одиночку в своем офисе и не разговаривает с другими, напишет массу лишнего кода. Вместо одной версии простой служебной функции, используемой по всей системе, каждый программист создаст свою версию одной и той же функции.

Часть

2

Общие проблемы разработки программ

Эта часть книги содержит общие правила для процесса разработки программ и не касается деталей собственно языков Си и Си++. Я сделаю это в последующих частях.

15. Прежде всего, не навреди

Это правило касается сопровождения программ. Будучи ребенком, я читал научно-фантастический рассказ, в котором незадачливый путешественник во времени случайно наступает на доисторическую бабочку и, вернувшись в свое время, находит окружающий мир изменившимся ужасным образом. Это похоже на большие компьютерные программы, где тронь здесь что-то кажущееся незначительным — и где-то там вся программа перестает работать. Методы объектно-ориентированного проектирования существуют, прежде всего, для решения (или по крайней мере для облегчения решения) этой проблемы в будущем, но уже существуют миллионы строк старого кода, который сегодня нуждается в сопровождении.

Мне приходилось видеть людей, которые изменяют программу просто потому, что им не нравится, как она выглядит. Это не очень хорошая идея. Если вы не знаете всех частей программы, затрагиваемых изменением (а это почти невозможно), то не трогайте код. Вы можете вполне резонно возразить, что на самом деле ни одно из правил в этой книге не относится к сопровождению программ. Вы просто не можете менять существующий код в соответствии с каким-то методическом руководством (как бы вам этого ни хотелось), не вызывая риска непоправимого вреда. Представленные здесь правила полезны лишь в том случае, когда вы начинаете программу с нуля.

16. Редактируйте свою программу

17. Программа должна быть переписана не менее двух раз

18. Нельзя измерять свою производительность числом строк

Прежде, когда вы изучали английский в школе, то вам никогда не приходило в голову сдавать черновик письменного задания, если вы, конечно, рассчитывали на оценку выше тройки. Тем не менее, многие компьютерные программы являются просто черновиками и содержат столько же ошибок, сколько и черновики ваших сочинений. Все хорошие программы вначале написаны, а затем отредактированы с целью улучшения. (Конечно, я имею в виду "редактировать" в смысле "исправлять".)

Имейте в виду, что редактирование должно быть сделано по окончании, потому что неотредактированный текст программы, по сути,

невозможно сопровождать (точно также, как и ваше неотредактированное сочинение было невозможно читать). Создатели программы знакомы с ее текстом и могут выполнить редактирование более эффективно, чем программист, занимающийся сопровождением, который сначала должен ее расшифровать перед тем, как выполнить какую-либо реальную работу.

К сожалению, это очень похоже на театральное шоу, когда кто-то пишет программу быстро, но не думая о сопровождении или об элегантности. "Ого, он выдает в два раза больше кода вдвое быстрее". Учтите, что тот же бедный сопровождающий программист будет затем вынужден затратить в восемь раз больше времени, сокращая первоначальный размер программы наполовину и делая ее пригодной для использования. Число строк кода в день, как мера объема, не является мерилом производительности.

Если вам нужен другой, чем сопровождение, мотив, то имейте в виду, что редактирование может рассматриваться как процесс уменьшения чего-либо. Маленькие программы выполняются быстрее.

19. Вы не можете программировать в изоляции

Классическая книга Джеральда Уэйнберга *"The Psychology of Computer Programming"* (New York: Van Nostrand Reinhold, 1971) содержит великолепную историю об автоматах с газированной водой. Администрация одного вычислительного центра решила, что слишком много времени растрачивается сотрудниками у автоматов с газированной водой. Люди создают много шума и ничего при этом не делают, поэтому автоматы убрали. Через несколько дней консультанты на местах были настолько перегружены работой, что к ним стало невозможно обратиться. Мораль состоит в том, что люди совсем не зря растрачивали время: оказывается, издавая весь этот шум, они помогали друг другу в решении проблем.

Изоляция может стать настоящей проблемой в группе объектно-ориентированного проектирования, которая по необходимости должна состоять из пользователей, проектировщиков, программистов, специалистов по документации и т.д., работающих совместно. Так как число программистов в этой группе часто меньше, чем в более традиционных проектных коллективах, то становится трудно найти кого-то, с кем можно обсудить проблемы; страдает производительность. Подумайте о еженедельных вечеринках в вашей фирме, как средстве повышения производительности.

20. Пустые потери времени

Если вы не можете решить неподатливую проблему, то займитесь на некоторое время чем-либо другим. Программисты часто наиболее продуктивны, когда смотрят в окно, слоняются по коридорам с пустым выражением на своих лицах, сидят в кафе и пьют кофе с молоком, или иным способом "теряют время".

Я был студентом в те древние времена, когда персональным компьютером был Apple I, а серьезные студенты-программисты владели коробками S-100, которые вы программировали, вводя двоичные команды переключателями лицевой панели по одному байту за раз. (Если вы были счастливчик, то имели интерпретатор языка БЕЙСИК и терминал, сделанный из старого телевизора). Студенты делили PDP 11/70, работавшую под UNIX (которая отлично работала на 16-битовой машине с 64 Кбайт памяти — Боже мой! Как все с тех пор усовершенствовалось). О возможности использования персонального компьютера для выполнения домашних заданий не было и речи.

На среднем занятии по программированию присутствовало от 40 до 80 человек, и одновременно проводилось шесть и более занятий. Поэтому, когда на занятии раздавалось задание, то вы хватали бумагу с ним и буквально опрометью бросались вниз в терминальный зал, где приковывали себя к компьютеру и начинали яростно программировать до тех пор, пока не выполняли свое задание. Это могло продолжаться несколько дней. Если вы отрывались поесть или поспать, то ваш терминал занимал другой, и у вас появлялась весьма реальная перспектива не уложиться в срок, отведенный на задание. Некоторые люди все еще продолжают программировать подобным способом.

Такая обстановка, конечно, не способствовала хорошо продуманному проектированию программ, поэтому большинство из этих программ были в четыре раза больше, чем необходимо, и требовали в два раза больше времени на отладку, чем требовалось. К тому же, количество строк кода, написанное за час, сокращается пропорционально количеству часов, которые вы просидели, глядя на экран. (Это иллюзия — думать, что вы можете достичь большей производительности, работая 12 часов в день вместо 8).

Однажды, будучи на последнем курсе, я был так расстроен, пытаясь решить одну проблему, о которую бился головой в течение примерно четырех часов, что с чувством отвращения завершил сеанс и выскочил на улицу. Примерно три минуты спустя, когда я спускался по холму за порцией сосисок, искомое решение неожиданно всплыло в моей голове. Это было настоящим откровением: вы должны расслабиться, чтобы дать

своему мозгу возможность работать. К сожалению, я не смог пробиться назад к компьютеру, поэтому я так никогда и не исправил свою ошибку, но, по крайней мере, понял, как должен работать тот процесс.

21. Пишите программу с учетом сопровождения — вы специалист по сопровождению

Сопровождение начинается немедленно после завершения программы, а сопровождением на этой стадии обычно занимаетесь вы сами. Это хорошая мысль — осчастливить сопровождающего программиста. Поэтому ваша первая забота о том, чтобы программа легко читалась. Структура и назначение каждой строки должны быть избыточно ясны, и если это не так, то вам нужно добавить поясняющие комментарии.

Одной из причин того, что поиски математических доказательств корректности программ остаются донкихотством, заключается в том, что нет программ без ошибок. Каждая программа не только содержит ошибки, но и требования к ней меняются, как только программа начинает эксплуатироваться, и у пользователя появляются потребность в каких-то новых свойствах, что вызывает появление новых и усовершенствованных ошибок. Так как ошибки всегда с нами, то мы должны писать нашу программу так, чтобы их можно было легче искать.

Вы можете переформулировать настоящее правило таким образом: Не умничайте. Искусный код почти невозможно сопровождать.

21.1. Эффективность — часто просто пугало

Я потратил несколько часов, делая одну подпрограмму более "эффективной", и не останавливался, чтобы подумать о том, как часто эта подпрограмма будет вызываться, что является пустой потерей времени в том случае, когда программа вызывается лишь один или два раза. Ваша программа должна быть непременно настолько эффективной, насколько это возможно, но вашей первоочередной заботой является сопровождение, и вы не должны приносить читаемость в жертву на алтарь эффективности. Напишите программу сперва с учетом сопровождения, затем запустите свою программу под профайлером и определите, где на самом деле есть узкие места. Будучи вооружены реальной информацией, вы теперь знаете, где стоит обменять часть читаемости на скорость, и можете вернуться и сделать это изменение. Тем не менее, вы можете включить первоначальный текст в комментарий, чтобы не потерять его. Всегда имейте в виду, что любое количество подчисток на уровне текста программы не повысит эффективность так, как это сделает лучший алгоритм. Пузырьковая сортировка будет

выполняться медленно вне зависимости от того, насколько хорошо она запрограммирована.

Часть

3

Форматирование и документация

Форматирование важно; тексты на Си и Си++ и так достаточно тяжело читаются, чтобы ухудшать ситуацию еще и плохим форматированием. Представьте, что вы пытаетесь прочесть книгу, текст которой не отформатирован: нет абзацных отступов, разделения абзацев пустой строкой, пробелов после знаков препинания и так далее. Сопровождение плохо отформатированной программы невозможно.

Я объединил правила форматирования и документации в одной части, потому что форматирование является одним из лучших инструментов документирования, имеющихся в вашем распоряжении. Под "документацией", рассматривающейся в этой части книги, понимается документирование программ (т.е. комментарии, а не документация уровня пользователя). Его зря не относят к программированию, ведь хороший комментарий для вас — в буквальном смысле нить Ариадны.

Я сознаю, что дискуссии по вопросам форматирования и документации часто достигают религиозного накала. Имейте в виду, что приводимые здесь мной правила — только те, которыми я пользуюсь сам. Имеются и другие совершенно разумные методы работы. С другой стороны, кто-то не особо умный мне однажды сказал, что "не важно, какой стиль форматирования вы применяете, если вы применяете его постоянно". Программа, которая постоянно форматируется плохо, хуже, чем понятная временами. Случайный просвет лучше, чем никакого.

22. Программа без комментариев ничего не стоит

Программа, на написание которой затрачен год, может использоваться в течение 10 лет. Вам придется затратить на сопровождение гораздо больше денег, чем вы выделили на первоначальную разработку, а программа без комментариев несопровождаема. "Блестящий" программист, который втрое быстрее, чем другие, пишет короткий, элегантный, но некомментированный текст программы, вам обойдется дорого. Какому-то менее талантливому программисту придется затратить в 10 раз больше времени, чем нужно, устраняя неизбежные ошибки.

Программисты, которые не могут писать по-английски (или на том языке, на котором говорят в стране, где предполагается осуществлять сопровождение), изготовляют часовые бомбы, а не компьютерные программы. Так как хорошая документация столь необходима для процесса сопровождения, то важно, чтобы программисты были способны ее написать. По этой причине начинающие программисты, имеющие диплом по филологии, истории или другой гуманитарной дисциплине, часто являются более предпочтительными, чем люди с дипломами по естественным наукам (математике, физике и т.п.). Специалисты по естественным наукам редко знают как писать, а большинство из них также не знают как программировать; они обучены тому, как запрограммировать алгоритм, а не как написать пригодную для сопровождения компьютерную программу.

К счастью, писать можно легко научиться. Конечно, если вы придерживаетесь правила "Сделай сначала комментарии", то вам придется писать все свои комментарии до начала программирования.

23. Располагайте программу и документацию вместе

Если документация отделена от текста программы, то ее очень трудно обновлять. Следовательно, основная часть вашей документации должна располагаться в комментариях, а не в отдельном документе.

Если вам на самом деле нужна отпечатанная документация высшего качества, то вы можете воспользоваться чем-нибудь похожим на систему Web (для языка Паскаль) или CWeb (для языков Си и Си++) в комбинации с TEX[1]. Я пользуюсь подобной системой под названием

[1] Web описана в книге Дональда Кнута ""*The WEB System of Structured Documentation*" (Palo Alto: Stanford University Dept. of Computer Science, Report No.STAN-CS-83-980, 1983). Система CWeb описана в книге Дональда Е. Кнута и Сильвио Ливая "*The CWeb System of Structured Documentation*" (Reading: Addison Wesley, 1994). Обе публикации не только описывают как работают эти системы, но

arachne, которая была разработана мной для того, чтобы писать свою книгу *"Compiler Design in C"*. (Arachne документирует тексты на Си и Си++, используя в качестве редактора troff). Все эти программы позволяют вам размещать исходный текст программы и документацию в одном файле. Вы можете выделить исходный текст для компиляции, или загрузить этот файл в текстовый процессор, чтобы напечатать единое руководство с исходным текстом и документацией. Эти системы позволяют осуществлять перекрестный поиск идентификаторов в программе и документации, позволяя вам прослеживать связи одной части программы с другой ("этот код используется вон там"), и так далее. Так как для получения печатной версии используется обычный текстовый процессор, то вы можете делать то, чего непросто добиться в комментариях — вставлять рисунки, например.

24. Комментарии должны быть предложениями

Они должны быть хорошо составлены и иметь правильную пунктуацию, по возможности без сокращений. Не превращайте свои комментарии в секретный код, применяя странные сокращения и изобретая свой собственный грамматический строй. Вам также не должна требоваться нить Ариадны для расшифровки комментариев.

25. Пропустите свой исходный тест через систему проверки орфографии

Ваши комментарии не только станут более читаемыми, этот метод побудит вас использовать в качестве имен переменных те, что легко читаются, т. к. являются обычными словами.

26. Комментарий не должен подтверждать очевидное

Начинающие программировать на Си склонны попадать в эту ловушку. Избегайте явно нелепых случаев типа:

```
++x;          // увеличить x
```

но мне также не нравятся комментарии типа:

хорошо демонстрируют это. В этих книгах документируются реальные тексты программ, реализующих указанные системы.

TEX является редакционно-издательской системой Кнута. Она имеется в нескольких коммерческих версиях.

```
/*-----------------------------------------
 * Определения глобальных переменных:
 *-----------------------------------------
 */
```

Любой средний программист знает, как выглядит определение.

27. Комментарий должен предоставлять только нужную для сопровождения информацию

Особенно неприятным и бесполезным комментарием является декларативный заголовочный блок. Заголовок сам по себе не является злом, а совсем наоборот. Блок комментариев в начале файла, описывающий, что делается в файле, может быть довольно полезен. Хороший блок говорит вам, какое свойство реализуется файлом, показывает список открытых (не статических) функций, сообщает вам, что эти функции делают и т.д.

Заголовки, которые мне не нравятся, имеют содержание, которое определяется указанием, обычно типа какого-то руководства по фирменному стилю. Такие заголовки обычно выглядят подобно листингу 1, увеличивая беспорядок посредством обильных количеств бесполезной информации за счет читаемости. Часто случается так, как на листинге 1, когда заголовок существенно больше самой программы. Также обычно, как в нашем случае, что текст программы или совершенно самодокументирован, или нужно добавить одну или две строки комментариев, чтобы он им стал. Хотя требование вышеуказанной бессмыслицы и может согревать душу деспота-руководителя, но для облегчения сопровождения оно мало что дает.

Листинг 1. Бесполезный заголовочный комментарий

```
1  /*----------------------------------------------------------**
2  **                                                          **
3  ** ДАТА: 29 февраля 2000 г.                                 **
4  ** ФУНКЦИЯ:                                                 **
5  **  equal                                                   **
6  **                                                          **
7  ** АВТОР:                                                   **
8  **  Джозеф Эндрюс                                           **
9  **                                                          **
10 ** ОПИСАНИЕ:                                                **
11 **  Эта функция предназначена для сравнения двух строк     **
12 **  на лексикографическое равенство.                        **
13 **                                                          **
14 ** ИСКЛЮЧЕНИЯ:                                              **
15 **  Функция не работает для строк Unicode.                  **
16 **                                                          **
17 ** СПЕЦИАЛЬНЫЕ ТРЕБОВАНИЯ:                                  **
```

```
 18 **  нет.                                                    **
 19 **                                                         **
 20 ** АРГУМЕНТЫ:                                              **
 21 **  char *s1; Указатель на первую сравниваемую строку     **
 22 **  char *s2; Указатель на вторую сравниваемую строку     **
 23 **                                                         **
 24 ** РЕЗУЛЬТАТЫ:                                             **
 25 **  Функция возвращает true, если строки-аргументы       **
 26 **  лексикографически идентичны.                          **
 27 **                                                         **
 28 ** КОММЕНТАРИИ:                                           **
 29 **  нет.                                                  **
 30 **                                                         **
 31 ** ПРИМЕЧАНИЯ ПО РЕАЛИЗАЦИИ:                              **
 32 **  нет.                                                  **
 33 **                                                         **
 34 ** ИСТОРИЯ ИЗМЕНЕНИЙ:                                     **
 35 **                                                         **
 36 ** АВТОР: Эндрюс, Джозеф                                 **
 37 ** ДАТА: 12, июль, 1743                                  **
 38 ** ИЗМЕНЕНИЕ: Начальное состояние                         **
 39 **                                                         **
 40 ** АВТОР: Джонс, Том                                     **
 41 ** ДАТА: 13, июль, 1743                                  **
 42 ** ИЗМЕНЕНИЕ: Изменены имена аргументов со str1, str2.   **
 43 **                                                         **
 44 ** Вест текст программы в этом файле охраняется          **
 45 ** авторским правом.                                     **
 46 ** Copyright (c) Вымышленная корпорация                  **
 47 ** Все права сохраняются.                                **
 48 **                                                         **
 49 ** Никакая часть этой подпрограммы не может быть         **
 50 ** воспроизведена в любой форме без явного разрешения    **
 51 ** в трех экземплярах со стороны отдела Министерства     **
 52 ** сокращения персонала. Нарушители будут лишены своего  **
 53 ** старшего сына.                                        **
 54 **-----------------------------------------------------------**
 55 */
 56 inline equal ( char *s1, char *s2 )
 57 {
 58     return !strcmp( s1, s2 ); // Возвращает истину, если
 59 }                             // строки равны.
```

Действительная проблема заключается в том, что этот тип заголовков нарушает ряд других правил, таких как: "не комментируй очевидное", "исключай неразбериху" и так далее. Тот минимум реальной информации, который содержится в этом заголовке, относится к системе регистрации изменений, а не к исходному тексту программы. Комментарии в программе должны сообщать вам сведения, полезные при сопровождении.

28. Комментарии должны быть в блоках

Комментарии в общем воспринимаются лучше, когда помещаются в многострочных блоках, которые чередуются с блоками текста программы. Для этого комментарий должен описывать на высоком уровне, что делают несколько последующих строк кода. Если комментарии попадаются через строчку, то это похоже на чтение двух книг одновременно, причем по строке из каждой по очереди. И если программа, комментируемая вами, сложная, то вы можете воспользоваться сносками:

```
// Вот блочный комментарий, описывающий последующий блок
// программы. После общего резюме я описываю некоторые
// особенности:
//
// 1. Этот комментарий описывает, что происходит в строке
// с меткой 1
//
// 2. Этот комментарий описывает, что происходит в строке
// с меткой 2
//
// В точке 1 алгоритм устанавливается на ...
//

here_is_the_code();
while( some_condition )
{
  this_code_is_rather_obscure();      /* 1 */
}
more_stuff_here();
while( some_condition )
{
  this_code_is_also_obscure();        /* 2 */
}
```

29. Комментарии должны быть выровнены вертикально

Выравнивайте начало и конец комментария вертикально в многострочных комментариях.

```
/* Первая строка,
 * вторая строка,
 * третья строка.
 */
```

Если ваш компилятор их поддерживает, то здесь помогут комментарии в стиле Си++:

```
// Первая строка,
// вторая строка,
// третья строка.
```

Есть две причины появления этого правила, они обе демонстрируются в последующей программе:

```
/*********************************************************
void the_function( void )

  Это многострочный комментарий, выполняющий все, что должен
  делать комментарий.

  К сожалению, отсутствие слева вертикального столбца из
  звездочек затрудняет зрительное разделение комментария и
  программы

*********************************************************

void the_function( void )
{
// далее настоящая функция.

   code_goes_here();
}

/*********************************************************/
```

Во-первых, заметили ли вы, что я забыл поместить / в конце второй строки звездочек? Таким путем можно запросто терять целые функции. Во-вторых, трудно заметить, где оканчивается комментарий и начинается текст программы. Исправьте обе ошибки следующим образом:

```
/*********************************************************
 * void the_function( void )
 *
 * Это многострочный комментарий, выполняющий все, что должен
 * делать комментарий.
 *
 * Вертикальный столбец из звездочек слева облегчает
 * зрительное разделение комментария и программы
 *********************************************************
 */

void the_function( void )
{
// далее настоящая функция.

   code_goes_here();
}
```

30. Используйте аккуратные столбцы везде, где можно

Так как форматирование по сути является видом комментирования, то это правило применяйте также и к тексту программы. Два следующих блока функционально одинаковы, но заметьте, насколько легче искать имена переменных во втором блоке, причем не из-за выравнивания комментариев, а потому что имена образовали аккуратный столбец:

```
int x; // Опишите, что делает x.
unsigned long int (*pfi)(); // Опишите, что делает pfi.
const char *the_variable; // Опишите, что делает the_variable.
int z; // Опишите, что делает z.
x = 10; // Здесь идет комментарий.
the_variable = x; // Здесь второй комментарий.
z = x; // А здесь третий.
```

по сравнению с:

```
int                x;              // Опишите, что делает x.
unsigned long int ( *pfi )();      // Опишите, что делает pfi.
int                z;              // Опишите, что делает z.

const char         *the_variable; // Опишите, что делает
                                  //   the_variable.
x            = 10;        // Здесь идет комментарий.
the_variable = x;         // Здесь второй комментарий.
z            = x;         // А здесь третий.
```

Вы можете рассматривать на этот вид форматирования как по сути "табличный", как если бы я создал таблицу со столбцами "тип", "имя" и "описание".

Другим хорошим местом для использования столбцов является список инициализации элементов в Си++, который я форматирую следующим образом:

```
class derived : public base
{
   string        str;
   const int     x;
public:
   derived( char *init_str, int init_x ) {}
}

derived::derived( char *init_str, int init_x )
                                    :base( str, x   )
                                    ,str ( init_str )
                                    ,x   ( init_x   )
{}
```

31. Не располагайте комментариев между именем функции и открывающей скобкой

Основная сложность в следующем примере:

```
foo( int x )
/* Не помещайте
 * комментарий
 * здесь. */
{
    //...
}
```

заключается в том, что тело функции может оканчиваться на следующей странице или не помещаться на одном экране. То есть читающий не может сказать, видит ли он прототип или действительное определение. Поместите этот комментарий или до имени функции, или вставьте его в тело функции ниже открывающей скобки:

```
/* Или помещайте
** его здесь.
*/

foo( int x )
{
    /* или здесь,
    ** с таким же отступом, что и у кода.
    */
}
```

32. Помечайте конец длинного составного оператора чем-нибудь, имеющим смысл

Прежде всего, подобные комментарии в конце блока:

```
while( a < b )
{
    for( i = 10; --1 >= 0; )
    {
       f( i );
    } // for
} // while
```

не дают ничего, кроме неразберихи, если блоки короткие. Я использую их только тогда, когда составной оператор слишком велик, чтобы поместиться на экран (в моем текстовом редакторе около 40 строк) или в нем столько уровней вложений, что я не могу понять суть происходящего. Комментарии в конце блока обычно целесообразны в больших составных операторах, но мне приходилось часто видеть подобный код:

На первой странице:

```
while( a < b )
{
    while( something_else() )
    {
        for( i = 10; --1 >= 0; )
        {
            for( j = 10; --j >= 0; )
            {
// далее идет масса кода
```

На какой-то из последующих страниц:

```
            } // for
        } // for
    } // while
} // while
```

Эти комментарии слишком кратки, чтобы быть полезными. Завершающие блок комментарии должны полностью описывать управляющий оператор. Завершающие блок комментарии из предыдущего примера должны выглядеть таким образом:

```
            } //       for( j = 10; --j >= 0; )
        }    //     for( i = 10; --1 >= 0; )
    }        //   while( something_else() )
}           // while( a < b )
```

Так как **#ifdef** почти всегда расположен на некотором расстоянии от **#endif**, то я всегда ставлю метку у **#endif**:

```
#ifndef __SOMEFILE_H_
#define __SOMEFILE_H_

// здесь следует 1000 строк программы

#endif // __SOMEFILE_H_
```

То же самое я делаю с **#else**.

33. Располагайте в строке только один оператор

Нет абсолютно никакой причины упаковывать в одну строку столько операторов, сколько сможете, если только у вас нет намерения сделать программу нечитаемой. Если в строке больше одной точки с запятой, то что-то неверно. Не используйте запятую-оператор (даже если вы знаете, что это такое) по той же причине. Очевидным исключением является оператор **for**, все три части которого должны быть на одной строке.

34. Указывайте имена аргументов в прототипах функций

Это особенно важно в определениях классов. Страницы руководств (и встроенных систем помощи) для программы, над которой вы старательно работаете, вряд ли существуют в тот момент, когда они вам нужны больше всего — когда вы начинаете разработку. Вы обычно получаете документацию из двух мест: комментариев, добавленных к настоящим функциям и заголовочных файлов. Из них обычно лучше заголовочный файл, как более компактный. В любом случае из документации, которая на самом деле нужна по функции, вам необходимо знать порядок и число аргументов, а легче всего найти эту информацию в прототипе функции. Если ваш прототип похож на этот:

```
some_function( int, int, int, int, int );
```

то вам не удастся получить много помощи.

35. Используйте "предикатную" форму при разбиении длинных выражений

"Предикатом" в английском языке называется вторая половина предложения — глагол и дополнение, над которым глагол выполняет действие. Порядок слов в английском предложении, конечно, фиксированный: глагол всегда идет первым.

Многие компьютерные языки имитируют структуру английского языка (языки Си и Си++ попадают в эту категорию). Паскаль, например, даже повторяет пунктуацию английского предложения: с точкой с запятой, отделяющей независимые операторы, и точкой на конце. Вызов функции является хорошим примером предикатной формы: глагол является именем функции, а прямые дополнения (вещи, на которые действует глагол) являются аргументами.

Вы также можете посмотреть на операторы типа глагола, потому что они выполняют некоторое действие над операндами ("объектами"). Это разумно, поэтому используйте ту же самую схему, что и в английском предложении, если вам нужно разбить длинное выражение на несколько строк. Помещайте сначала глагол (оператор):

```
if( its_thursday_and_the_moon_is_in_scorpio()
  ||its_friday_afternoon_and_close_to_quitting_time()
  ||i_just_cant_bear_to_look_at_this_computer_for_another_minute()
  )
{
   go_home();
}
```

Конечно, вы можете добиться большего, просто сократив имена подпрограмм до чего-то более приемлемого, так чтобы все они поместились на одной строке.

Заметьте, как я расположил круглые и фигурные скобки в предыдущем примере с целью придания его структуре большей ясности. И при этом я использовал фигурные скобки, хотя язык этого не требовал, для облегчения поиска оператора, связанного с **if**. Следующий текст программы читается с большим трудом, потому что хуже видно, где заканчивается **if** и начинается оператор:

```
if( its_thursday_and_the_moon_is_in_scorpio()
  ||its_friday_afternoon_and_close_to_quitting_time()
  ||i_just_cant_bear_to_look_at_this_computer_for_another_minute())
  go_home();
```

36. Подпрограмма должна помещаться на экране

Затраты на вызов подпрограмм в Си/Си++ невелики; если для функции указано ключевое слово **inline**, то затрат фактически нет. Поэтому хорошая мысль состоит в том, чтобы создавать подпрограммы удобного в обращении размера и использовать их в больших количествах. Имя подпрограммы позволяет добиваться значительной абстракции. Если имена выбраны правильно, то у вас зачастую можете исчезнуть нужда в комментариях.

Вообще, мне нравится, когда рабочая часть подпрограммы (ее текст, в меньшей степени комментарии заголовка и так далее) видна полностью при редактировании; она должна целиком помещаться на экране или в окне.

37. Нужно обеспечивать возможность распечатки всего текста программы

Часто проще найти ошибку на распечатанной странице, чем на экране, даже на большом. Текст на бумаге легче читается вследствие более высокого разрешения и лучшего фокусного расстояния, чем на экране. При этом вы можете разложить бумажные распечатки по всей поверхности стола. Вы вряд ли чем-нибудь сможете заменить бумагу в этом плане, если только у вас нет монитора с диагональю полтора метра и разрешением 300 точек на дюйм.

Итак, ваша программа должна легко печататься. Подпрограммы должны помещаться по возможности на одной странице (не более 60 строк), и ни одна строка программы не должна быть такой длины, чтобы не поместиться в строку распечатки (около 79 символов, в зависимости от

шрифта), даже если ваш текстовый редактор поддерживает скроллинг по горизонтали. Если вы выйдете за эти пределы, то будете вынуждены использовать столь мелкий шрифт для вывода, что совсем не сможете прочитать свою распечатку.

38. Используйте штриховую линию для зрительного разделения подпрограмм

Я всегда ставлю такой комментарий:

```
//---------------------------------------------------------
```

над каждым определением функции. (И к тому же, я не использую штриховую линию нигде более). Хотя пустые строки замечательно подходят для зрительного разделения блоков программы, исключительно их использование не дает эффекта. Штриховая линия между функциями облегчает их поиск. Так же как пустая строка указывает на границу абзаца, штриховые линии подобны заголовкам разделов. Если мне нужно еще более четкое разделение, то я использую:

```
//=========================================================
//ОПИСЫВАЮЩИЙ ТЕКСТ
//=========================================================
```

Подумайте об этом, как о заголовке для главы. Я стараюсь не помещать никаких комментариев, за исключением штриховых линий, вне функции, потому что такие комментарии затрудняют обнаружение определений этой функции. При этом я форматирую функции следующим образом:

```
//---------------------------------------------------------
void descriptive_name( type descriptive_name )
{
// Если имена функции и аргументов недостаточно содержа-
// тельны, то я помещаю здесь комментарий, описывающий,
// что она делает. Я опускаю этот комментарий, если имена
// достаточно понятны. (Соответствующее правило гласит:
// "Не объясняй очевидного").
//
// Затем я описываю возвращаемое значение и аргумент.
// И вновь вы можете не использовать комментарий, если
// имена достаточно удачные.
//
// Наконец, я помещаю здесь комментарий, описывающий, как
// функция делает то, что она делает. И снова я пропускаю
// этот комментарий, если программа сама по себе достаточно
// содержательна.

  code_goes_here();
}
```

39. Пробел — один из наиболее эффективных комментариев

Это кажется мелочью, но это может чрезвычайно улучшить читаемость вашей программы. Обратите внимание, как используются пробелы в этой книге в качестве организующего средства, и вы поймете, как использовать их в своей собственной программе. Пустые строки (или отступ в первой строке) зрительно разделяют абзацы. Пробел следует за точкой, но не должен ей предшествовать, потому что точка завершает что-либо. Идея вам ясна. А вот правила:

- Разбивайте текст программы на логические куски (т.е. абзацы), где каждый кусок выполняет одну операцию. Окружите эти куски или пустыми строками, или строками с фигурными скобками.
- За знаком препинания всегда должен идти пробел.
- Операторы являются сокращениями слов. Когда вы видите "+", то говорите "плюс". Подобно любому сокращению, вы должны окружать идентификатор пробелами. (Например: `a + b` читается "a плюс b", `a+b` читается "аплюсb").
- Исключение составляют унарные операторы, которые рассматриваются как словарные префиксы или суффиксы (`*p`, `a--`, `f(arg,arg)` и т.д.).
- `.` или `->` в Си/Си++ являются эквивалентом символа подчеркивания. До и после них пробелов быть не должно: `p->msg()`, `obj.msg()`.

Вот пример того, что может произойти, когда вы что-нибудь упаковываете слишком плотно. Рассмотрим:

```
int *p;
y=(x/*p++);
f(int /* вставка */);
```

Если вы удалите комментарии, то получите:

```
int *p;
y=(x
     );
```

Сочетание `/*` в выражении `y=(x/*p++)` расценивается как символ начала комментария, который заканчивается сочетанием `*/` в вызове функции `f()`. (Такой случай действительно со мной произошел, и мне потребовался целый день, чтобы в нем разобраться. Естественно, компилятор не давал сообщений об ошибках, потому что здесь все синтаксически правильно).

Еще замечание по данному поводу. Мне часто приходилось позднее видеть объявления подобные следующему:

```
int*        x;
```

Проблема состоит в том, что:

```
int*        x,  y;
```

не объявляет два указателя, как подсказывает распределение пробелов. Здесь мы имеем на самом деле еще одну проблему из рода "я могу на любом языке программирования писать как на ФОРТРАНЕ". Было бы прекрасно, если бы Си работал так же, как подсказывает предыдущее форматирование, но это не так. После правильного форматирования

```
int         *x,  y;
```

становится совершенно ясно, что `x` — указатель, а `y` — нет.

40. Используйте отступы в четыре пробела

Никлас Вирт, который изобрел языки Паскаль и Модула-2, однажды выпустил книгу, где всюду использовались отступы в один символ. Чтение в ней листингов стало одним из самых тяжелых случаев в моей практике. Используйте достаточно большие отступы, чтобы ваш читатель мог сказать, что в тексте видно абзацы; четыре пробела кажутся идеальными.

Вы должны делать отступы последовательно. Даже во внешнем блоке подпрограммы должны быть отступы. Такой вариант неприемлем:

```
void f( void )
{
if( x )
    yyy();
more_code();
even_more_code();
}
```

потому что слишком трудно найти начало подпрограммы. Сравните предыдущий вариант со следующим:

```
void f( void )
{
    if( x )
       yyy();
    more_code();
    even_more_code();
}
```

41. Условные операторы выделяются абзацными отступами

Я делаю это даже в операторах из одной строки:

```
if( by_land )
    one();
else
    two();
```

а не так:

```
if( by_land ) one() else two();
```

Очевидным исключением является:

```
if( by_land )
{
    one();
}
else if( by_sea )
{
    two();
}
else if( by_air )
{
    three();
}
```

Я использовал здесь скобки по двум причинам. Во-первых, я как-то попал внутрь условного оператора при отладке и забыл вставить скобки, как в следующем тексте:

```
if( by_land )
    one();                  if(debug) printf("Ox...");
else if ( by_sea )
```

что привело привело фактически к:

```
if( by_land )
    one();

if (debug)
    printf("Ox...");
else
    if( by_sea )
```

Со скобками программа к тому же лучше читается. Я часто нарушаю правило абзацных отступов, когда использую форматирование для того, чтобы показать с кристальной ясностью, что происходит. Аккуратные столбцы делают это осуществимым:

```
if      ( by_land   ) one();
else if ( by_sea    ) two();
else if ( by_tunnel ) three();
```

но это — нечитаемо:

```
if (by_land)one();
else if(by_sea)two();
else if(by_tunnel)three();
```

Подобный код никуда не годится:

```
for ( a ; b ; c );
while ( ++i < 10 );
```

Слишком просто случайно сделать следующее:

```
while ( i < 10 );
    ++i;
```

(Другими словами, "вечно сравнивать `i` с `10`, затем увеличить `i`"). Если точка с запятой никогда не ставится в конце строки, начинающейся с **`for`** или **`while`**, то вы можете использовать утилиту поиска строки типа grep для обнаружения таких проблем.

41.1. Комментарии должны иметь тот же отступ, что и окружающий текст программы

Абзацные отступы предназначены для того, чтобы сделать структуру вашей программы легко понятной. Если вы организуете отступы в комментариях беспорядочным образом, то этим вы лишите их смысла. Комментарий в следующей программе должен быть снабжен отступами:

```
f()
{
/* Здесь идет
 ** длинный комментарий
 */
    code();
}
```

Здесь имеется сходная проблема:

```
f()
{
int local_var;
int another_local_var;
    code();
}
```

Отсутствие отступа при определении локальных переменных заставляет предположить, что они находятся в той же области видимости, что имя

функции (которая является глобальной). Так как это не так, то вы должны сделать отступ, чтобы указать на область видимости:

```
f()
{
    int local_var;
    int another_local_var;
    code();
}
```

42. Выравнивайте скобки вертикально по левой границе

Иногда поиск отсутствующей фигурной скобки превращается в крупную проблему. Если вы вынесете скобки туда, где их хорошо видно, то их отсутствие будет сразу же заметно:

```
while ( some_condition )
{
   // внутренний блок
}
```

Я в самом деле не люблю так называемый стиль Кэрнигана и Ричи:

```
if( condition ){
   code();
}else{
   more_code();
}
```

Здесь не только трудно проверить скобки на парность, но и отсутствие зрительного разделения за счет строк, содержащих лишь открытые скобки, ведет к ухудшению читаемости.

43. Используйте скобки, если в условном операторе имеется более, чем одна строка

Это правило применяется, если даже дополнительными строками является комментарий. Проблема заключается в том, что слишком легко случайно добавить другой оператор и забыть добавить скобки. Текст, подобный нижеследующему, более надежен:

```
if( something() )
{
    */ Quo usque tandem abutere Gatesalina, patientia nostra♣.
```

♣ "До каких же пор ты, Катилина, будешь испытывать наше терпение..." — начало известной речи Цицерона. — *Прим. перев.*

```
    */
    somethig_else();
}
```

Часть

4

Имена и идентификаторы

Имена играют важную роль. Правильно выбранные имена могут сделать программу поистине самодокументированной, не требуя совсем или требуя мало дополнительного набора в виде явных комментариев. Плохо выбранные имена (например, `state` — состояние или штат ?) могут добавить ненужную сложность в вашу программу. Эта часть книги содержит правила выбора имен.

44. Имена должны быть обычными словами английского языка, описывающими то, что делает функция, аргумент или переменная

Избегайте аббревиатур; они ухудшают читабельность программ. Некоторые по привычке продолжают использовать аббревиатуры, что приводит к абсурдной практике, типа отбрасывания последней буквы слова или удаления из него всех гласных. Возьмем странно названную функцию UNIX `creat()`; очевидно, что `create()` лучше. Я также видел странности типа `lnghth` вместо `length` и `mt` вместо `empty`.

При этом общепринятые аббревиатуры являются очевидными исключениями. Вот тот минимум из них, которым я пользуюсь сам:

`col` — индекс столбца;
`cur` — текущий;
`i j` — обобщенные счетчики циклов;
`max` — максимум (обычно в качестве префикса или суффикса);
`min` — минимум (обычно в качестве префикса или суффикса);
`obj` — обобщенный объект (имеется указатель на базовый класс, но производный класс не известен);
`p ptr` — обобщенный указатель;
`s str` — строка (в языке Си обычно **`char`***),

но не употребляйте их, если называемый объект используется не как обобщенный. Например, `i` имеет смысл использовать в качестве счетчика цикла в операторе **`for`**, если этот счетчик используется просто для подсчета числа итераций:

```
for( i = 10; --i >= 0; )     // нарисовать 10 тире
  putchar('-');
```

Используйте настоящее имя, если вы применяете счетчик для чего-нибудь отличного от счета. Сравните этот код:

```
for( i = 0; i < imax; ++i )
  for( j = 0; j < jmax; ++j )
    move_cursor( i,j );
```

со следующим:

```
for( row = 0; row < max_row; ++row )
  for( col = 0; col < max_col; ++col )
    move_cursor( row, col );
```

Я также избегаю использовать `x` и `y` вместо `row` и `column`. Одно из ранее указанных правил рекомендует пропускать программу через систему проверки орфографии. Действительное достоинство этого метода состоит

в том, что он побуждает вас использовать в качестве имен обычные слова.

44.1. Не используйте в качестве имен тарабарщину

Отличный образец такого подхода можно наблюдать в любом предлагаемом Microsoft примере программы, хотя эта проблема ни в коем случае не ограничивается корпорацией Microsoft. Все демонстрационные программы Microsoft Windows включают тип переменной в ее имя. Например, объявление типа:

```
const char *str;
```

будет сделано следующим образом:

```
LPCSTR   lpszstr;
```

Переведите `lpszstr` как "указатель типа `long` с именем `str` на строку, оканчивающуюся `0`". На самом деле здесь несколько проблем, не последней из которых является тот факт, что `LPCSTR` скрывает наше объявление указателя. Тем не менее, обсуждаемое правило посвящается проблеме самого имени.

Этот стиль выбора имен называется "венгерской" записью по названию родины руководителя отдела программирования Microsoft Чарльза Саймони, который его изобрел. (а не потому, что его использование придает программам Microsoft такой вид, как будто они написаны на венгерском языке.)

Венгерская запись целесообразна для языка ассемблера, в котором все, что вы знаете о переменной — это ее размер. Включение информации о типе в имя переменной позволяет вам контролировать правильность ее использования.[2] Языки более высокого уровня типа Си и Си++ используют для этой цели объявление переменных.

Доктор Саймони несколько раз в печати защищал такой метод записи, но я бы не стал его рекомендовать для программ на Си или Си++. По моему мнению, венгерская запись не дает ничего, кроме ухудшения читаемости программ. Простые `str` или `string` значительно легче читаются и содержат ту же информацию. Если вам на самом деле нужно узнать тип, то для этого достаточно вернуться к определению.[3]

---

[2] Я подозреваю, что венгерская запись так интенсивно используется вследствие того, что большая часть Microsoft Windows написана на языке ассемблера.

[3] По крайней мере, оно должно быть. Я подозреваю, что некоторые энтузиасты венгерской записи так плохо организуют свои программы, что просто не могут найти нужные объявления. Включая тип в имя, они избавляются от многих часов поисков в неудачно спроектированных листингах. Программы на языке ассемблера, которые по необходимости включают в себя множество глобальных переменных, являются очевидным исключением.

Существует и более распространенный, хотя и менее радикальный прием, при котором имена указателей начинают символом p. Эта практика тоже загромождает программу. Вы ведь не начинаете имена целочисленных переменных типа **int** символом i, переменных типа **double** — d, а функций — f? Очевидным исключением является случай, когда у вас есть объект и указатель на этот объект в одной и той же области видимости:

```
char      str[128],     *pstr  =  str;
```

с другой стороны, для указателя, вероятно, лучше содержательное имя. Сравните:

```
char    str[128],    *first_nonwhite  =  str;
while ( isspace(*first_nonwhite) )
    ++first_nonwhite;
// В этой ситуации имя *first_nonwhite говорит вам гораздо
// больше о том, что делает переменная, чем предыдущее "*pstr".
```

45. Имена макросов должны записываться ЗАГЛАВНЫМИ_БУКВАМИ

Как показывается в последующих разделах, макросы часто вызывают побочные эффекты. Поэтому полезно иметь возможность определить с первого взгляда, что у вас является макросом. Конечно, вы не должны использовать только заглавные буквы для чего-нибудь помимо макросов, иначе вы не достигнете цели данного правила.

45.1. Не используйте заглавных букв для констант перечисления

Должна быть обеспечена возможность замены констант, определенных в перечислении, на переменную типа **const**. Если ее имя записано заглавными буквами, то вам придется его менять. Кроме того, имеются еще и проблемы с макросами (вскоре будут рассмотрены), которых нет у перечислений. Поэтому будет полезно иметь возможность различать их с первого взгляда.

45.2. Не используйте заглавных букв в именах типов, созданных при помощи `typedef`

Так как макрос также может использоваться в манере, подобной **typedef**, то полезно знать может или нет что-то быть использовано в качестве синтаксически правильного типа. Например, имея:

```
typedef void (*ptr_to_funct)(int );
```

вы можете написать следующее:

```
(ptr_to_funct)( p );   // преобразует p в указатель на функцию
 ptr_to_funct f(long); // f возвращает указатель на функцию
```

Макрос типа:

```
#define PTR_TO_FUNCTION void (*) (int )
```

позволяет вам сделать преобразование:

```
(PTR_TO_FUNCTION) ( p );
```

но не позволяет объявить функцию:

```
PTR_TO_FUNCTION f(long);
```

Указанный макрос при подстановке дает:

```
void (*) (int ) f(long);
```

но компилятору нужно:

```
void (*f(long))(int );
```

Имя типа из строчных букв не вызовет никаких проблем при чтении, потому что по смыслу вы всегда можете сказать, используется ли оно для типа или нет.

46. Не пользуйтесь именами из стандарта ANSI Си

Идентификаторы, начинающиеся с символа подчеркивания, и имена типов, оканчивающиеся на `_t`, были зарезервированы стандартом ANSI Си для использования разработчиками компиляторов. Не используйте эти символы. Также избегайте имен функций, вошедших в стандарт ANSI Си и в проект стандарта ISO/ANSI для Си++[♣].

47. Не пользуйтесь именами Microsoft

Это может показаться правилом, специфичным только для Microsoft, но на самом деле это не так (учитывая имеющуюся склонность Microsoft к мировому господству). Любой, кто заботится о переносимости, должен рассчитывать на то, что его или ее программа со временем может или работать под управлением операционной системы Microsoft, или взаимодействовать с библиотекой классов Microsoft. Библиотека MFC, например, перенесена на Macintosh и во многие операционные среды

[♣] В августе 1998 г. стандарт ратифицирован в виде *"ISO/IEC 14882, Standard for the C++ Programming Language"*. Популярно изложен в книге: Страуструп Б. Язык программирования С++, 3–е изд. /Пер. с англ.—СПб.; М.: "Невский диалект" — "Издательство БИНОМ", 1999.–991 с. — *Прим. перев.*

UNIX/Motif на момент написания этой книги, и, вероятно, появится на других операционных системах в ближайшем будущем.

На момент написания этой книги интерфейс прикладного программирования Windows (API) включает в себя около 1200 функций. Библиотека MFC, быстро вытесняющая первоначальный интерфейс на языке Си, добавляет около 80 определений классов. К сожалению, метод Microsoft состоит в добавлении каждый раз дополнительных функций и классов в новую версию компилятора. Если Microsoft случайно выберет в качестве имени для функции или класса то, которое вы используете для каких-то других целей, угадайте, кому из вас придется его сменить?

Так как ни один из идентификаторов Microsoft не соответствует стандарту ANSI Си, требующему, чтобы имена поставляемых разработчиком объектов начинались с символа подчеркивания, то вы должны предохраняться, избегая использования соглашений по выбору имен в стиле Microsoft:

- Все имена функций Microsoft используют соглашения в стиле Паскаля о `СмесиЗаглавныхИСтрочныхБукв()`, и они всегда начинаются с заглавной буквы. Я предпочитаю имена только из строчных букв с символами подчеркивания, но что бы вы ни выбрали, `НеИспользуйтеСтильMicrosoft()`. Функции-члены в классах MFC используют то же самое соглашение.
- Все имена классов Microsoft начинаются с заглавной "C" с последующей заглавной буквой (например, `CString`, `CWnd`, `CDialog` и т.д.). Начальная "C" мало что дает, кроме беспорядка, и ее пропуск удаляет нас от области имен Microsoft.
- Одна из наиболее фундаментальных заповедей объектно-ориентированного проектирования запрещает оставлять незащищенными данные-члены в определении класса. Тем не менее, многие классы MFC имеют открытые поля данных. Все эти поля начинаются с `m_`, не имеющих другого назначения, кроме как увеличить беспорядок. Тем не менее, мы можем использовать эту бессмыслицу для того, чтобы не начинать имена своих собственных полей с `m_` и таким образом легко отличать свои члены от унаследованных из базовых классов MFC.

48. Избегайте ненужных идентификаторов

Имена для констант часто вообще не нужны. Например, не определяйте значения, возвращаемые при ошибке; если возвращается всего одна ошибка, возвратите просто `FALSE`. Не делайте так:

```
enum { INSERT_ERROR, DELETE_ERROR };

insert()
{
   //...
   return INSERT_ERROR;
}

delete()
{
   //...
   return DELETE_ERROR;
}
```

а просто возвратите `0` в случае ошибки и в случае успеха любое правильное значение типа `1`.

49. Именованные константы для булевых величин редко необходимы

Выбор неверного имени может добавить значительную ненужную сложность в вашу программу. Рассмотрим следующую простейшую функцию, которая подсчитывает количество слов в строке:

```
int nwords(const char *str)
{
    typedef enum { IN_WORD, BETWEEN_WORDS } wstate;

    int     word_count  = 0;
    wstate  state       = BETWEEN_WORDS;
    for(; *str ; ++str )
    {
       if( isspace(*str) )
           state = BETWEEN_WORDS;
       else
           if( state != IN_WORD )
           {
               ++word_count;
               state = IN_WORD;
           }
    }
    return word_count;
}
```

Неправильно выбранное имя `state` заставило нас ввести два ненужных

идентификатора: `IN_WORD` и `BETWEEN_WORDS`. Теперь взгляните на этот вариант:

```
int nwords2(const char *str)
{
    int  word_count     = 0;
    int  in_word        = 0;

    for(; *str ; ++str )
    {
        if( isspace(*str) )
            in_word = 0;
    else
        if( !in_word )
        {
            ++word_count;
            in_word = 1;
        }
    }
    return word_count;
}
```

Переименование нечетко названной переменной `state` во что-нибудь, что действительно описывает назначение переменной, позволило мне исключить булевые именованные константы `IN_WORD` и `BETWEEN_WORDS`. Получившаяся подпрограмма меньше и легче читается.

Вот другой пример. Следующая программа:

```
enum child_type { I_AM_A_LEFT_CHILD, I_AM_A_RIGHT_CHILD };
struct tnode
{
    child_type    position;
    struct tnode  *left,
                  *right;
} t;
//...
t.position = I_AM_LEFT_CHILD;
if( t.position == I_AM_LEFT_CHILD )
    //...
```

может быть упрощена подобным образом[♣]:

```
struct tnode
{
    unsigned      is_left_child ;
    struct tnode  *left,
                  *right;
} t;
t.is_left_child = 1;
if( t.is_left_child )
```

[♣] В стандарте ISO/IEC 14882 существует тип ‘bool’. Имеет смысл заменить тип переменной `is_left_child` на **`bool`**. — *Ред.*

```
    //...
```

тем самым исключая два ненужных идентификатора. И вот последний пример:

```
enum { SOME_BEHAVIOR, SOME_OTHER_BEHAVIOR, SOME_THIRD_BEHAVIOR
};

f( SOME_BEHAVIOR,       x);
f( SOME_OTHER_BEHAVIOR, x);
f( SOME_THIRD_BEHAVIOR, x);
```

требующий четырех идентификаторов (три именованные константы и имя функции). Лучше, хотя это не всегда возможно, исключить селекторную константу в пользу дополнительных функций:

```
some_behavior(x);
some_other_behavior(x);
some_third_behavior(x);
```

Обратной стороной этой монеты является вызов функции. Рассмотрим следующий прототип:

```
create_window( int has_border, int is_scrollable,
               int is_maximized );
```

Я снова выбрал рациональные имена для исключения необходимости в именованных константах. К сожалению, вызов этой функции плохо читаем:

```
create_window( TRUE, FALSE, TRUE );
```

Просто взглянув на такой вызов, я не получу никакого представления о том, как будет выглядеть это окно. Несколько именованных констант проясняют обстоятельства в этом вызове:

```
enum { UNBORDERED  =0; BORDERED   =1}; // Нужно показать
значения,
enum { UNSCROLLABLE=0; SCROLLABLE =1}; // или create_window()
enum { NORMAL_SIZE =0; MAXIMIZED  =1}; // не будет работать.
 //...
create_window( BORDERED, UNSCROLLABLE, MAXIMIZED );
```

но теперь у меня другая проблема. Я не хочу использовать именованные константы внутри самой `create_window()`. Они здесь только для того, чтобы сделать ее вызов более читаемым, и я не хочу загромождать эту функцию таким кодом, как:

```
if( has_border == BORDERED )
 //...
```

сравнивая его с более простым:

```
if( has_border )
```

```
//...
```

Первый вариант уродлив и многословен. К сожалению, если кто-то изменит значение именованной константы `BORDERED`, второй оператор **`if`** не будет работать. Я обычно соглашаюсь с мнением, что программист, занимающийся сопровождением, не должен менять значения идентификаторов, как я это проделал в предыдущем примере.

Часть

5

Правила обычного программирования

Эта часть содержит правила, относящиеся к написанию собственно исходного текста программы, в отличие от предыдущей части, в которой рассматривалась разработка программы в целом. Эти правила не слишком зависят от выбора языка программирования.

50. Не путайте привычность с читаемостью

(Или синдром "настоящего программиста, который может программировать на любом языке как на ФОРТРАНе"). Многие люди пытаются злоупотреблять препроцессором для того, чтобы придать Си большее сходство с каким-нибудь другим языком программирования. Например:

```
#define begin  {
#define end    }

while ( ... )
begin
// ...
end
```

Эта практика ничего не дает, кроме того, что ваш код становится нечитаемым для кого-нибудь, кто не знает того языка, который вы стараетесь имитировать. Для программиста на Си код станет менее читаемым, не более того.

Родственная проблема связана с использованием макросов препроцессора для скрытия синтаксиса объявлений Си. Например, не делайте следующего:

```
typedef const char *LPCSTR;

LPCSTR str;
```

Подобные вещи вызывают проблемы с сопровождением, потому что кто-то, не знакомый с вашими соглашениями, будет должен просматривать **typedef**, чтобы разобраться, что происходит на самом деле. Дополнительная путаница возникает в Си++, потому что читатель может интерпретировать происходящее, как определение объекта Си++ из класса `LPCSTR`. Большинству программистов на Си++ не придет в голову, что `LPCSTR` является указателем. Объявления Си очень легко читаются программистами на Си. (Кстати, не путайте вышеупомянутое с разумной практикой определения типа **word** в виде 16-битового числа со знаком для преодоления проблемы переносимости, присущей типу **int**, размер которого не определен ни в ANSI Си, ни в ANSI Си++).

К тому же, многие программисты избегают условной операции (**?:**) просто потому, что она им кажется непонятной. Тем не менее, это условное выражение может существенно упростить ваш код и, следственно, сделать его лучше читаемым. Я думаю, что:

```
printf("%s", str ? str : "<пусто>");
```

гораздо элегантнее, чем:

```
if ( str == NULL )
    printf( "<пусто>" );
else
    printf( "%s", str );
```

Вы к тому же экономите на двух вызовах `printf()`. Мне также часто приходится видеть неправильное использование операций `++` и `--`. Весь смысл автоинкремента или автодекремента заключается в соединении этих операций с другими. Вместо:

```
while ( *p )
{
    putchar ( *p );
    ++p;
}
```

или:

```
for( ; *p ; ++p )
    putchar ( *p );
```

используйте:

```
while( *p )
    putchar ( *p++ );
```

Этот код вполне читаем для компетентного программиста на языке Си, даже если ему нет эквивалентной операции в ФОРТРАНе или Паскале.

Вы также никогда не должны прятать операторы в макросах из-за того, что вам просто не нравится, как они выглядят. Я однажды видел следующее в реальной программе:

```
struct tree_node
{
    struct tree_node *lftchld;
};

#define left_child(x) ((x)->lftchld)
//...
traverse( tree_node *root )
{
    if( left_child(root) )
        traverse( left_child( root ) );
    // ...
}
```

Программист намеренно сделал определение структуры труднее читаемым для того, чтобы избежать конфликта имен между полем и совершенно необходимым макросом, и все из-за того, что ему не

нравился внешний вид оператора ->. Для него было бы гораздо лучшим выходом просто назвать поле `left_child` и совсем избавиться от макроса.

Если вы действительно думаете, что программа должна внешне выглядеть как на Паскале, чтобы быть читаемой, то вы должны программировать на Паскале, а не на Си или Си++.

51. Функция должна делать только одно дело

Это обычно не очень удачная мысль — записывать то, что должна делать функция, через ее аргументы. Это должно делать имя функции. Например:

```
UpdateAllViews( CView *sender, long lhint, CObject *phint )
{
// sender lhint phint
// NULL   xx    xx     Начальное обновление, вызываемое из
//                     обрамляющего окна
// Cview* 0     Crect* Вызывается, когда встроенный объект
//                     становится действительным. phint
//                     указывает на прямоугольник документа,
//                     сохраняя положение недействительного
//                     объекта
// Cview* 1     Crect* Сообщение, посылаемое объектом CView*
//                     ("sender" - передатчик). phint сохраняет
//                     для CView* обрамляющее окно его клиента.
}
```

Вам нужны вместо этого три функции: `initial_update()`, `update_embedded_object()` и `update_view()`. Верным ключом для понимания того, что здесь что-то не так, является туманная природа имен аргументов. Функциям не должны передаваться "намеки". Им должны даваться указания.

52. Иметь слишком много уровней абстракции или инкапсуляции так же плохо, как и слишком мало

Основной смысл использования таких абстракций, как функции или символьные константы (или инкапсуляций, подобных определениям `struct` или `class`), заключается в улучшении читаемости программ. Не пользуйтесь ими просто потому, что вы можете делать это. Например, вложенные структуры в данном фрагменте не служат какой-либо полезной цели:

```
struct tree_node;

struct child_ptr
{
```

```
    unsigned          is_thread;
    struct tree_node *child;
};

struct tree_node
{
    struct child_ptr left,
                     right;
};

tree_node *p;
if( !p->left.am_a_thread )
    p = p->left.child;
```

Следующий код лучше читается, потому что в нем меньше точек, и легче сопровождается, так как в нем нужно отлаживать на одно определение меньше:

```
struct tree_node
{
    struct tree_node *left_child;
    unsigned          left_is_thread : 1;
    struct tree_node *right_child;
    unsigned          right_is_thread : 1;
};

if( !p->left_is_thread )
    p = p->left_child;
```

53. Функция должна вызываться более одного раза, но…

Кроме того, если функция должным образом связана (т.е. если она выполняет единственную операцию и весь код функции работает на ее результат), то нет причины извлекать кусок кода в другие функции, если только вы не желаете использовать эту часть кода где-то еще. Мой опыт говорит, что когда функция становится слишком большой, то часто возможно выделить куски, которые обладают достаточной общностью, чтобы быть использованными где-либо еще в программе, так что это правило на самом деле не противоречит правилу "маленькое — прекрасно". Если вы не выделяете этот код, то блочный комментарий, описывающий назначение этого блока программы, который вы могли бы выделить, служит той же самой цели, что и имя функции — документированию.

Тем не менее, иногда выделение кода в функции меньшего размера существенно улучшает читаемость этого кода по причине устранения беспорядка. Однако эта практика — создание абстракции для части кода в виде имени функции — добавляет идентификаторы в область глобальных

имен, и эта возросшая общая сложность является определенным минусом. Если я использую функцию этим способом — как абстракцию —, то обычно объявляю ее одновременно статической, чтобы к ней нельзя было получить доступ снаружи текущего файла, и встроенной, чтобы ее вызов не приводил к накладным расходам. Не доводите процесс функционального абстрагирования до крайности. Мне доводилось видеть отличные программы, доведенные абстрагированием до полностью нечитаемого состояния такой степени, что нет ни одной функции длиннее, чем 5 или 6 строк. Получающаяся программа работает также значительно медленнее, чем необходимо, и ее исходный текст в 5 раз длиннее, чем нужно.

53.1. Код, используемый более одного раза, должен быть помещен в функцию

Это правило является обратной стороной предыдущего. Если вы обнаруживаете почти идентичный код появляющимся более чем в одном месте своей программы, то этот код должен быть выделен в подпрограмму, которая вызывается из нескольких мест. Выгода состоит в меньшем размере программы и лучшей сопровождаемости вследствие упрощения программы и того, что вы должны теперь сопровождать лишь одну функцию; если вы находите ошибку, то вам нужно исправить ее только в одном месте. Как было упомянуто ранее, имя функции также дает хорошую абстракцию. Вызовы функции с хорошо выбранным именем являются обычно самодокументирующимися, устраняя необходимость в комментариях.

54. Функция должна иметь лишь одну точку выхода

Это правило применимо лишь к программам на Си. Вообще, множество переходов **goto** к одной точке выхода лучше, чем много операторов **return**. Этим способом вы можете поместить точку прерывания у единственной точки выхода, вместо того, чтобы возиться с несколькими прерываниями. Например[♣]:

```
f()
{
    int возвращаемое_значение = ОШИБКА;
    if( некое_условие )
    {
        // ...
```

[♣] Решение о переводе некоторых из идентификаторов, по меньшей мере, спорное. Однако, если вы не знаете английского, то будете лишены возможности оценить юмор автора, которым он оживил большую часть своих примеров. — *Ред.*

```
        возвращаемое_значение = НЕЧТО;
        goto выход;
}
else
{
    // ...
    возвращаемое_значение = НЕЧТО_ЕЩЕ;
    goto выход;
}

выход:
    return возвращаемое_значение;
}
```

Этот метод не срабатывает в Си++, потому что функции конструктора вызываются неявно в качестве части объявления; объявление часто скрывает вызов функции. Если вы пропускаете объявление, то вы пропускаете и вызов конструктора. Например, в следующей программе деструктор для `x` *вызовется*, а конструктор — нет:

```
foo()
{
    if( некое_условие )
        goto  выход;

    некий_класс x;  // Конструктор не вызывается. (Оператор
                    // goto перескакивает через него.)
                    // ...
выход:
                // Здесь вызывается деструктор для x
                // при выходе x из области видимости.
}
```

Вследствие этой проблемы лучше всего совсем избегать переходов **`goto`** в программах на Си++.

54.1. Всегда предусматривайте возврат значения из блока внешнего уровня

Иногда, когда подпрограммы короткие, не стоит стараться обеспечить единственную точку выхода. (По моему мнению, правило "избегай запутанности" перекрывает любое другое правило, с которыми оно входит в конфликт). В этой ситуации всегда старайтесь убедиться, что из подпрограммы нет таких путей, которые не проходят через оператор **`return`**. Не так:

```
if( a )
{
    // ...
    return делай_что_нужно();
}
```

```
else
{
    // ...
    return ОШИБКА;
}
```

а так:

```
if( a )
{
    // ...
    return делай_что_нужно();
}
// ...
return ОШИБКА;
```

В идеале, выход по ошибке организуется из внешнего уровня блока так, чтобы вы правильно обработали неожиданный аварийный выход на внешний уровень.

55. Избегайте дублирования усилий

Следующий фрагмент демонстрирует эту проблему:

```
if( strcmp(a, b) < 0  )
{
}
else if( strcmp(a, b) > 0 )
{
}
else if( strcmp(a, b) == 0 )
{
}
```

Вызов `strcmp()` в Си связан с немалыми накладными расходами (как в Паскале и других языках программирования), значительно лучше сделать так:

```
int cmp = strcmp(a, b);
if( cmp < 0 )
{
}
else if( cmp > 0 )
{
}
else  // остается случай cmp == 0
{
}
```

56. Не захламляйте область глобальных имен

Беспорядок в области глобальных имен является характерной проблемой для среды групповой разработки. Вам не очень понравится спрашивать разрешение от каждого участника группы каждый раз, когда вы вводите новый идентификатор. Поэтому:

- Локальная переменная всегда более предпочтительна, чем член класса.
- Член класса всегда более предпочтителен, чем статическая глобальная переменная.
- Статическая глобальная переменная всегда более предпочтительна, чем настоящая глобальная переменная.

Статический глобальный идентификатор не экспортируется из файла .c, поэтому он невидим из других модулей. Применяйте модификатор **`static`** к как можно большему числу глобальных идентификаторов (переменным и функциям). Ключ доступа **`private`** в определении класса еще лучше. Идентификатор, определенный локально внутри подпрограммы, лучше всех, потому что он изолирован от всех других функций в программе.

Вывод: избегайте препроцессора. Так как у макроса такая большая область видимости, то он, по сути, то же самое, что и глобальная переменная.

56.1. Избегайте глобальных идентификаторов

Раскрывая немного предыдущее правило, положим, что две функции связаны посредством глобальной переменной, если одна из них устанавливает эту переменную, а вторая ее использует. (Если бы глобальная переменная не использовалась совместно, то не было бы причины иметь ее глобальной; она могла бы быть статической локальной). Отношения связи с участием глобальных переменных вызывают особенно неприятные проблемы при сопровождении, потому что эти отношения тяжело отслеживать. Когда глобальная переменная меняется во время выполнения программы, то очень трудно разобраться, что ее изменило. Аналогично, если вы должны изменить поведение глобального объекта, то очень трудно разобраться, где он используется. По этой причине лучше всего вообще избегать глобальных переменных. Конечно, большинству программ реального мира необходимо незначительное количество глобальных переменных, но, как правило, я начинаю сильно нервничать, если их становится больше 10.

Вы зачастую можете ограничить область видимости глобальной переменной одним файлом, объявив ее статической, что, по меньшей

мере, ограничит ущерб одним файлом. По крайней мере, вы знаете, что все отношения связи сосредоточены в текущем файле. Также имейте в виду, что все, что я говорил о глобальных переменных, относится и к макросам, функциям и так далее. Ограничивайте доступ к функциям, делая их любой ценой статическими.

56.2. Никогда не требуйте инициализации глобальной переменной при вызове функции

Вот одна ситуация, где оправданы статические глобальные переменные: если у вас применяется система рекурсивных функций. (Вы можете использовать статические глобальные переменные для сокращения потребного объема стековой памяти, применяя их для передачи значений между подпрограммами. Вам никогда не нужно использовать статическую глобальную переменную для передачи информации из одной подпрограммы в другую, являющуюся рекурсивным экземпляром той же самой подпрограммы. Верным выбором в этой ситуации будет использование локальной статической переменной. Используйте статические глобальные переменные в ситуациях, где вызывается более одной подпрограммы: `A()` вызывает `B()`, которая вызывает `A()`, вызывающую в свою очередь `B()`, и так далее).

Так как глобальная переменная, используемая нашей рекурсивной функцией, сделана статической для минимизации связывания, то как вам ее инициализировать? Далее показано, как не нужно этого делать. Вот файл 1:

```
static  int  glob;

get_glob( x )
{
    return glob;
}

set_glob( x )
{
    glob = x;
}

void recursive_function( void )
{
    int  y = glob;
    // ...
    recursive_function();
}
```

а вот файл 2:

```
set_glob( 10 );
```

```
recursive_function();
x = get_glob();
```

Вы при этом немногого достигли с точки зрения связывания; на самом деле, с простой глобальной переменной было бы проще управляться. Кроме того, вы подготовили себе потенциальную проблему: возможность забыть вызвать `set_glob()`. Вот как сделать это правильно:

```
static int glob;

static void recursive_function( void )
{
    int y = glob;
    // ...
    recursive_function();
}

int do_recursive( int init_val )
{
    glob = init_val;
    recursive_function();
    return glob;
}
```

Ни к глобальной переменной, ни к рекурсивной функции нельзя получить доступ прямо снаружи модуля из-за статического выделения памяти. Вы должны получить доступ к рекурсивной функции посредством *функции доступа* `do_recursive()`, которая гарантирует, что все инициализировано правильно перед тем, как выполнить вызов рекурсивной функции.

56.2.1. Делайте локальные переменные статическими в рекурсивных функциях, если их значения не участвуют в рекурсивном вызове

Так как мы занялись темой рекурсии, то вот правило, которое используется для того, чтобы еще сильнее сократить использование стека. Локальная переменная может быть объявлена статической (тем самым она минует стек), если ее значение не должно сохраняться после рекурсивного вызова. Вот один пример:

```
f()
{
    static int i;
    // ...

    for( i = 10; --i >= 0; )
    // ...

    f();
```

```
    for( i = 10; --i >= 0; ) // переменная i вновь инициализиру-
                             // ется после рекурсивного вызова,
                             // поэтому она может быть статичес-
}                            // кой.
```

Вот другой:

```
int f()
{
    static int depth     = 0;
    static int depth_max = 0;

    ++depth; depth_max = max( depth, depth_max );

    if( depth > 10 )
        return -1;      // уровень рекурсии слишком глубок.

    f();
    --depth;
    return depth_max;
}
```

В этом последнем случае переменная `depth` используется для передачи информации — глубины рекурсии — от одного экземпляра подпрограммы другому, рекурсивному экземпляру этой же самой подпрограммы. Переменная `depth_max` хранит след достигнутой максимальной глубины рекурсии. `depth` вовсе не будет работать, если она должна будет сохранять свое значение после вызовов — весь смысл в том, что каждый рекурсивный вызов модифицирует эту переменную.

56.3. Используйте счетчик экземпляров объектов вместо инициализирующих функций

Инициализирующие функции, с очевидным исключением в виде конструкторов Си++, не должны использоваться просто потому, что слишком просто забыть их вызвать. Многие системы с оконным интерфейсом, например, требуют, чтобы вы вызывали функцию инициализации окна перед его созданием (и другую — закрытия — после удаления последнего окна). Это плохая идея. Уладьте эту проблему при помощи счетчика экземпляров, который обычно в Си должен быть глобальной переменной (объявленной статической для ограничения области ее видимости). Сделайте это так:

```
static int num_windows = 0; // ограничьте доступ к текущему
                            // модулю
create_window()
{
if( ++num_windows == 1 ) // только что создано первое окно
    initialize_video_system();
```

```
// ...
}
destroy_window()
{
    // ...
    if( --num_windows == 0 )       // только что уничтожено
        shutdown_video_system();   // последнее окно
}
```

В Си++ вы можете для этой цели использовать статический член класса.

56.4. Если оператор `if` завершается оператором `return`, то не используйте `else`

Вместо:

```
if( условие )
    return xxx;
else
{
    делать_массу_вещей();
}
```

обычно лучше записать:

```
if ( условие )
    return xxx;
делать_массу_вещей();
```

Лучше сделать так, чтобы последним оператором **return** был аварийный возврат по ошибке, так чтобы вы получили сообщение об ошибке, если нечаянно заблудились.

Условный оператор также может решать эту проблему в простых ситуациях и делать код более читаемым для новичка. Вместо:

```
f()
{
// ...
if( x )
    return 123;
else if ( y )
    return 456;
else
    return ERROR;
}
```

используйте

```
f()
{
    // ...
    return x ? 123   :
           y ? 456   :
               ERROR ;
```

```
}
```

Заметьте, насколько форматирование улучшает читаемость предыдущего кода.

Одна распространенная ситуация, в которой у вас имеется множество точек возврата, выглядит следующим образом:

```
if( A )
    return FAIL;

else if( B )
    return SUCCESS;
else
{
    // Масса кода
    return SUCCESS;  // Подозрительны два одинаковых
                     // возвращаемых значения.
}
```

Вы можете устранить это следующим образом. Во-первых, избавьтесь от повтора возвращаемых значений, переместив их во внешний уровень вот так:

```
if( A )
    return FAIL;

else if( B )
    ;
else
{
    // Масса кода
}
return SUCCESS;
```

Затем освободитесь от предложения **if**, связанного с пустым оператором:

```
if( A )
    return FAIL;

else if( B )
{
    // Масса кода
}
return SUCCESS;
```

57. Помещайте более короткий блок условного оператора `if/else` первым

Часто бывает, что у оператора **`if/else`** одно предложение (или внутренний блок) короткое (обычно оператор для обработки ошибки), а другое, выполняющее собственно работу, — большое:

```
if( некая_ошибка() )
    error( "AAAAxxxx!!!!" );
else
{
    // Здесь следуют 30 строк кода
}
```

Всегда помещайте короткое предложение в начале. То есть, не делайте так:

```
if( !некая_ошибка() )
{
    // Здесь следуют 30 строк кода
}
else
    error( "AAAAxxxx!!!!" );
```

Проблема заключается в том, что проверка в операторе **`if`** управляет **`else`** в той же степени, что и **`if`**. Если большой блок следует первым, то вероятность того, что вычисляться будет предложение **`else`**, расположенное на следующем экране или странице, довольно велика. И если я допустил в нем ошибку, то мне придется поломать голову над тем, как добраться до **`else`**. Если в это время **`if`** в поле зрения, то я знаю, как туда попасть.

58. Старайтесь сдвинуть ошибки с этапа выполнения на этап компиляции

Неинициализированные переменные — по сути ошибки, ждущие своего часа. Вы всегда должны инициализировать переменную при ее объявлении. В Си++ инициализация во время объявления возможна всегда, потому что объявление может проводиться везде, где можно поместить оператор; просто откладывайте объявление до тех пор, пока у вас не будет достаточно информации для *объявления в произвольном месте* с инициализацией переменной. Таким образом, если вы попытаетесь использовать эту переменную преждевременно, то получите ошибку на этапе компиляции ("переменная не найдена") вместо ошибки во время выполнения.

В Си вы можете объявлять переменную после любой открытой фигурной скобки, поэтому вы можете часто откладывать объявление на некоторое время, но при этом у вас в распоряжении нет гибкости Си++. В самом крайнем случае инициализируйте переменную таким значением, которое в ответ на него понятным образом вызовет в подпрограмме аварию; не присваивайте переменной произвольное значение — оно может быть принято в программе за имеющее смысл. Например, указатель, инициализированный значением `NULL`, более надежен, чем имеющий произвольное значение, которое может оказаться существующим адресом.

С другой стороны, хорошо иметь все объявления переменных в одном месте в начале блоке так, чтобы вы могли их легко найти. Если ваша подпрограмма достаточно мала, то вы обычно можете сделать и то, и другое. Например, вы можете разделить подпрограмму на части для того, чтобы переместить объявления в начало блока, чтобы вам их было легче найти. Подпрограмма, подобная следующей:

```
f()
{
    // код, который не использует переменную i

    int i = init_val;
    // код, который использует переменную i
}
```

может быть разделена следующим образом:

```
f()
{
    // код, который не использует переменную i
    g( init_val );
}

g( int init_val )
{
    int i = init_val;
    // код, который использует переменную i
}
```

Переменная-счетчик общего назначения, которая инициализируется в начале цикла **`for`**, является очевидным исключением из этого правила[♣]. Иногда использование слишком большого количества подпрограмм может вызвать больше проблем, чем решить их, и в этом случае лучше выбрать внутренние объявления. Используйте свою голову.

[♣] Переменная, объявленная в операторе **`for`**, не выживает после этого оператора. — *Ред*.

59. Применяйте указатели на функции Си в качестве селекторов

Это правило строго для программистов на Си. (Программирующие на Си++ должны использовать виртуальные функции). В Си заменяйте подобный код:

```
typedef enum  shape_type { CIRCLE, LINE, TEXT };
typedef struct
{
    shape_type  type;
    union shape_data
    {    // здесь данные для различных форм.
    } data;

} shape;

extern void print_circle( shape *p );
extern void print_line  ( shape *p );
extern void print_text  ( shape *p );

shape a_circle = { CIRCLE, ... };

print_shape( shape *p )
{
    switch( p->type )
    {
        case CIRCLE: print_circle( p );
        case LINE:   print_line  ( p );
        case TEXT:   print_text  ( p );
    }
}
```

на следующий:

```
typedef struct
{
    void (*print)( struct *shape );
    union   shape_data;
    {         // здесь данные для различных фигур.
    }
}
shape;

extern void print_circle( shape *p );
extern void print_line  ( shape *p );
extern void print_text  ( shape *p );

shape a_circle = { print_circle, ... };

print_shape( shape *p )
{
```

```
    ( p->type )( p );
}
```

Главные преимущества такого подхода заключаются в следующем:

- Вам больше не нужен перечислитель `shape_type`.
- Функцию `print_shape()` теперь написать гораздо проще.
- `print_shape()` будет продолжать работать без модификации, когда вы добавите новые фигуры в эту систему.

60. Избегайте циклов `do/while`

Цикл **`do`**/**`while`** опасен в принципе, так как вы обязательно выполняете его тело хотя бы один раз. Следовательно, вы должны проверить условия завершения до входа в этот цикл. Я часто вижу код, похожий на следующий:

```
if( !проверить_нечто )
    return ERROR;
do
{
    начинка();

} while( проверить_нечто );
```

Вам гораздо лучше сделать так:

```
while( проверить_нечто )
    начинка();
```

Похожий случай:

```
if( некое_условие() )
    do
        // масса материала
    while( некое_условие() && другой_материал() );
```

легче трактовать следующим образом:

```
while( некое_условие() )
{
    //   масса материала
    if( !другой_материал() )
        break;
}
```

Я профессионально занимаюсь программированием с 1979 года и за это время использовал цикл **`do`**/**`while`** всего два раза.

60.1. Никогда не используйте `do/while` для бесконечного цикла

Код, похожий на следующий:

```
do
{
    // здесь следует несколько страниц кода
} while( 1 );
```

просто заставляет сопровождающего программиста шарить по листингу взглядом, ища **`while`**, вместо того, чтобы найти его сразу в случае расположения оператор **`while`** `(1)` в начале цикла.

61. В цикле со счетчиком его значение должно по возможности уменьшаться

Циклы являются одним из тех мест, где малое повышение эффективности значительно улучшает выполнение программы, потому что их код выполняется многократно. Так как сравнение с нулем обычно более эффективно, чем сравнение с определенным числом, то цикл с уменьшающимся счетчиком, как правило, выполняется быстрее. Используйте

```
for( i = max; --i >= 0; )
    ;
```

вместо:

```
for( i = 0; i < max; ++i )
    ;
```

Заметьте, что в обоих случаях счетчики могут быть использованы в качестве действительного индекса массива, что может сделать этот код менее подверженным ошибкам вследствие того, что он ограничивает доступ к массиву его границами.

62. Не делайте одно и то же двумя способами одновременно

В качестве контрапункта к предыдущему правилу рассмотрим следующий фрагмент (содержащий в себе ошибку):

```
int array[ARRAY_SIZE];
int *p = array;
for( i = 1; i < ARRAY_SIZE ; ++i )
    *p++ = 0;
```

Проблема состоит в том, что счетчик не совпадает по фазе с указателем (`i`

имеет значение 1, когда указатель указывает на элемент array[0]), и последний элемент массива не будет инициализирован.

Я обычно предпочитаю для простых перемещений по массивам указатели (вместо индексов массива), потому что указатели, как правило, более эффективны, устраняя неявную операцию умножения в выражении a[i], интерпретируемом как:

```
( a + ( i* sizeof(a[0])))
```

Я бы переписал это код таким образом:

```
int         array[ARRAY_SIZE];
int         *current          = array;
int *const  end               = array + (SIZE-1);
while( current <= end )
    *current++ = 0;
```

Так же надежно (хотя и менее эффективно) сделать следующее:

```
int array[ARRAY_SIZE];
int i;
for( i = 0; i < ARRAY_SIZE ; ++i )
    array[i] = 0;
```

Кстати, если вы используете указатели, то вам придется извлекать индекс при помощи арифметики указателей, а не за счет сохранения второй переменной. У вас могут возникнуть проблемы, если вы передадите i функции в предыдущем примере с ошибкой. Воспользуйтесь подобным кодом:

```
for( current = array; current <= end; ++current )
{
    // ...
    f( current - array ); // передать функции f() текущий
                          // индекс массива
}
```

С другой стороны, обычно нужно избегать кода, подобного следующему, так как такой оператор цикла чрезвычайно неэффективен:

```
while( (current - array) < ARRAY_SIZE )
    // ...
```

63. Используйте оператор `for`, если имеются любые два из инициализурующего, условного или инкрементирующего выражений

Иначе используйте **`while`**. Такой код:

```
int  x = 10;

// далее следует 200 строк кода, в которых переменная x не
```

```
// используется

while( x > 0 )
{
    // снова следует 200 строк кода

    f( x-- );
}
```

не очень хорош, даже если вы сэкономили немного времени, соединив операцию декрементирования `--` с вызовом функции. Переместите инициализацию и `x--` в оператор **`for`**. Так как объявление в Си++ может располагаться везде, где можно поместить оператор, то вы даже можете объявить `x` непосредственно перед **`for`**:

```
int x = 10;
for ( ; x > 0 ; --x )
{
    // следует 200 строк кода

    f(x);
}
```

(И хотя вы можете сказать, что в Си++ есть возможность сделать **`for`** (**`int`**`=0;...`, такая практика вводит в заблуждение, потому что на самом деле область видимости у `x` внешняя, как если бы ее объявление было сделано в строке, предшествующей оператору **`for`**. Я не рекомендую этого)*.

Если три оператора внутри **`for`** слишком длинны, чтобы поместиться в одной строке, то вы можете их отформатировать их следующим образом:

```
for( некое_длинное_имя_переменной = f();
     некое_длинное_имя_переменной ;
     некое_длинное_имя_переменной = f() )
{
    // ...
}
```

но часто лучше выделить одно из предложений вот так:

```
int некое_длинное_имя_переменной = f();
for(; некое_длинное_имя_переменной;
      некое_длинное_имя_переменной = f() )
{
    // ...
}
```

или в чрезвычайном случае

---

* Кроме того, стандарт пересмотрел подход к жизни переменных, объявленных в операторе **`for`**. — *Ред.*

```
int некое_чрезвычайно_длинное_имя_переменной = f();
for(; ; некое_чрезвычайно_длинное_имя_переменной = f() )
{
    if( !некое_чрезвычайно_длинное_имя_переменной )
        break;
    // ...
}
```

Главное — это сосредоточить инициализацию, проверку и инкрементирование в одном месте. Я никогда не сделаю так:

```
int некое_чрезвычайно_длинное_имя_переменной = f();
while( некое_чрезвычайно_длинное_имя_переменной )
{
    // много строк кода

    некое_чрезвычайно_длинное_имя_переменной = f();
}
```

потому что это нарушает контроль над циклом.

64. То, чего нет в условном выражении, не должно появляться и в других частях оператора `for`

Так как оператор **`for`** предназначен для того, чтобы собрать инициализирующую, условную и инкрементирующие части цикла в одном месте, так чтобы вы могли, взглянув, понять, что происходит, то вы не должны загромождать оператор **`for`** материалом, который не имеет отношения к циклу, иначе вы лишите смысла всю эту конструкцию. Избегайте подобного кода:

```
int *ptr;
// ...
for( ptr = array, i = array_size; --i >= 0; f(ptr++) )
    ;
```

который лучше сформулировать так:

```
int *ptr = array;
for( i = array_size; --i >= 0 ; )
    f( ptr++ );
```

65. Допускайте, что ситуация может измениться в худшую сторону

Одним из лучших примеров этой проблемы связан со "знаковым расширением". Большинство компьютеров используют так называемую арифметику "двоичного дополнения". Крайний левый бит отрицательного числа в этой арифметике всегда содержит `1`. Например, восьмибитовый тип **`char`** со знаком, содержащий число `-10`, будет представлен в машине

с двоичным дополнением как 11110110 (или 0xf6). То же самое число в 16-битовом типе **int** представлено как 0xfff6. Если вы преобразуете 8-битовый **char** в **int** явно посредством оператора приведения типов или неявно, просто используя его в арифметическом выражении, где второй операнд имеет тип **int**, то компилятор преобразует **char** в **int**, добавляя второй байт и дублируя знаковый бит (крайний слева) **char** в каждом бите добавленного байта. Это и есть знаковое расширение.

Существуют два вида операций сдвига вправо на уровне языка ассемблера: "арифметический сдвиг" дает вам знаковое расширение (то значение, что было в крайнем левом бите до сдвига, будет в нем и после сдвига); "логический сдвиг" заполняет левый бит нулем. Данное правило, независимо от того, арифметический или логический сдвиг у вас, когда вы используете оператор сдвига Си/Си++, звучит очень просто: если вам требуется знаковое расширение, то допустите, что у вас получится заполнение нулями. А если вам нужно заполнение нулями, то представьте, что у вас получилось знаковое расширение.

Другим хорошим примером являются возвращаемые коды ошибок. Удивительное количество программистов не заботится о проверке того, не вернула ли функция malloc() значение NULL при недостатке свободной памяти. Быть может, они полагают, что имеется бесконечный объем виртуальной памяти, но известно, что ошибка может с легкостью вызвать резервирование всей имеющейся памяти, и вы никогда не обнаружите этого, если не будете проверять возвращаемые коды ошибок. Если функция может индицировать состояние ошибки, то вы должны допустить, что ошибка произойдет, по меньшей мере, однажды за время жизни программы.

66. Компьютеры не знают математики

Компьютеры — это арифметические инструменты, славные счетные машины. Они не знают математики. Поэтому даже такие простые выражения, как следующее, могут добавить вам хлопот:

```
int x = 32767;
x = (x * 2)/ 2;
```

(На 16-разрядной машине x получится равным -1.32767 — это 0x7fff. Умножение на 2 — на самом деле сдвиг влево на один бит, дает в результате 0xfffe — отрицательное число. Деление на два является арифметическим сдвигом вправо с гарантированным знаковым расширением, и так вы получаете теперь 0xffff или -1). Поэтому важно каждый раз при выполнении арифметических вычислений учитывать ограничения компьютерной системы. Если вы производите умножение

перед делением, то при этом рискуете выйти за пределы разрядности при сохранении результата; если вы сначала делите, то рискуете случайно округлить результат до нуля; и так далее. Численным методам для компьютеров посвящены целые книги, и вам нужно прочитать хотя бы одну из них, если в ваших программах много математики.

Вы также должны знать мелкие грехи своего языка программирования. В Си, например, преобразование типов выполняется по принципу "оператор за оператором". Я однажды потратил утро, пытаясь разобраться, почему следующий код ничего не делает:

```
long x;
x &= 0xffff; // очистить все, кроме младших 16-ти бит
             // 32-битного типа long.
```

Компьютер имел 16-битовый тип **int** и 32-битовый тип **long**. Константа `0xffff` типа **int** с арифметическим значением `-1`. Компилятор Си при трансляции `&=` обнаруживал разные типы операндов и поэтому преобразовывал **int** в **long**. `-1` в типе **long** представляется как `0xffffffff`, поэтому логическая операция `И` не имела эффекта. Это как раз тот способ, которым и должен работать данный язык программирования. Я просто об этом не подумал.

Заметьте, что вы не можете исправить эту ситуацию приведением типов. Все, что делает следующий код, это заменяет неявное преобразование типа явным. Но, тем не менее, происходит то же самое:

```
x &= (long)0xffff;
```

Единственным методом решения этой проблемы является:

```
x &= 0xffffUL;
```

или равноценный ему.

66.1. Рассчитывайте на невозможное

Оператор **switch** всегда должен иметь предложение с ключевым словом **default** для ситуации по умолчанию, особенно если эта ситуация не должна возникать:

```
f( int i )    // переменная i должна иметь значение 1 или 2.
{
    switch( i )
    {
    case 1: сделать_нечто();          break;
    case 2: сделать_нечто_другое();   break;
    default:
        fprintf(stderr, "Внутренняя ошибка в f(): неверное
                         значение i (%d)", i );
    exit( -1 );
```

```
    }
}
```

То же самое относится к блокам **if**/**else**, работающим в манере, схожей с оператором **switch**.

В цикле также нужна проверка на невероятное. Следующий фрагмент работает, даже если `i` первоначально равно `0` — чего по идее быть не должно:

```
f( int i )    // переменная i должна быть положительной
{
    while ( --i >= 0 )
        сделать_нечто();
}
```

Конструкция **while**`(--i)` менее надежна, так как она дает ужасный сбой в случае, если `i` сначала равно `0`.

66.2. Всегда проверяйте коды возврата ошибки

Это должно быть очевидно, но комитет ISO/ANSI по Си++ потребовал, чтобы оператор **new** вызывал исключение, если он не смог выделить память, потому что было установлено, что удивительное множество ошибок во время выполнения в реальных программах вызвано тем, что люди не потрудились проверить, не возвратил ли **new** значение `NULL`.

Мне также довелось видеть множество программ, в которых люди не позаботились посмотреть, сработала ли функция `fopen`(), перед тем как начать пользоваться указателем `FILE`.

67. Избегайте явно временных переменных

Большинство переменных, используемых лишь один раз, попадают в эту категорию. Например, вместо:

```
int x = *p++;
f( x );
```

должно быть:

```
f( *p++ );
```

Редко бывает, что полезна явная временная переменная, если вам нужно гарантировать порядок вычислений, или если выражение просто такое длинное, что его невозможно прочитать. В последнем случае имя переменной даст полезную информацию и, будучи выбрано правильно, может устранить необходимость в комментарии. Например, вы можете заменить:

```
f( Coefficient_of_lift * (0.5 * RHO * square(v)) );
```

```
// передать функции f() образующуюся подъемную силу
```

на:

```
double lift = Coefficient_of_lift * (0.5 * RHO * square(v));
f( lift );
```

Это правило не запрещает ни одно из подобных применений, а является, скорее, вырожденным случаем того, что упомянуто мной вначале.

68. Не нужно магических чисел

В основном тексте вашей программы не должно быть чисел в явном виде. Используйте перечислитель или константу для того, чтобы дать числу символическое имя. (Я уже объяснял, почему для этого не очень хорошо применять **#define**). Тут есть два преимущества:

- Символическое имя делает величину самодокументируемой, устраняя необходимость в комментарии.
- Если число используется более чем в одном месте, то менять нужно лишь одно место — определение константы.

Я иногда делаю исключение из этого правила для локальных переменных. Например, в следующем фрагменте используется магическое число (128):

```
f()
{
    char buf[128]
    ...
    fgets( buf, sizeof (buf) / sizeof(*buf), stdin );
}
```

Я использовал **sizeof**() в вызове fgets(), поэтому изменения размера массива автоматически отражаются в программе. Добавление дополнительного идентификатора для хранения размера добавит излишнюю сложность.

69. Не делайте предположений о размерах

Классической проблемой является код, исходящий из того, что тип **int** имеет размер 32 бита. Следующий фрагмент не работает, если у вас 32-битный указатель и 16-битный тип **int** (что может быть при архитектуре Intel 80x86):

```
double  a[1000], *p = a;
// ...
dist_from_start_of_array_in_bytes = (int)p - (int)a;
```

Более трудно уловима такая проблема в Си (но не в Си++):

```
g()
{
```

```
    doesnt_work( 0 );
}

doesnt_work( char *p )
{
    if( !p )                    // вероятно не работает
    // ...
}
```

Компилятор соглашается с этим вызовом, потому что в Си разрешены ссылки вперед (и не разрешены в Си++, так что там это не проблема). `0` — это тип **`int`**, поэтому в стек помещается 16-битовый объект. Но функция ожидает 32-битный указатель, поэтому она использует 16 бит из стека и добавляет к ним еще 16 бит всякого мусора для создания 32-битного указателя. Вероятнее всего, что **`if`**`(!p)` даст ложный результат, так как только 16 бит из 32 будут равны `0`.

Традиционное решение состоит в использовании `typedef` :

```
typedef int  word;          // всегда 16 бит
typedef long dword;         // всегда 32 бита.
```

После чего вы можете поменять операторы **`typedef`** в новой операционной среде, чтобы гарантировать, что `word` по-прежнему имеет размер 16 бит, а `dword` — 32 бита. Для 32-разрядной системы предыдущее может быть переопределено как:

```
typedef short word;         // всегда 16 бит
typedef int   dword;        // всегда 32 бита.
```

Другая связанная с размерностью часовая бомба спрятана в том способе, которым в ANSI Си обеспечивается работа с иностранными языками. ANSI Си определяет тип `wchar_t` для работы с расширенными наборами символов типа Unicode — нового 16-битного многонационального набора символов. Стандарт ANSI Си также утверждает, что перед строкой с расширенными символами должен стоять символ `L`. Microsoft и другие поставщики компиляторов стараются помочь вам писать переносимые программы, предусматривая макросы типа:

```
#ifdef _UNICODE
    typedef wchar_t _TCHAR
#   define _T(x)  L##x
#else
    typedef char _TCHAR
#   define _T(x) x
#endif
```

Если константа `_UNICODE` не определена, то оператор:

```
_TCHAR *p = _T("делай_что_нужно");
```

имеет значение:

```
char *p = "делай_что_нужно";
```

Если константа _UNICODE определена, тот же самый оператор получает значение:

```
wchar_t *p = L"делай_что_нужно";
```

Пока все хорошо. Вы теперь можете попробовать перенести вашу старую программу в среду Unicode, просто используя свой редактор для замены всех экземпляров **char** на _TCHAR и помещения всех строковых констант в скобки _T(). Проблема состоит в том, что такой код, как ниже (в котором все _TCHAR первоначально были типа **char**), более не работает:

```
_TCHAR str[4];
// ...
int max_chars = sizeof(str); // предполагает, что тип char
                             // имеет размер 1 байт
```

Тип _TCHAR будет иметь размер 2 байта при определенной константе _UNICODE, поэтому число символов у вас будет определено в два раза большим, чем есть на самом деле. Для исправления ситуации вы должны воспользоваться следующим вариантом:

```
int max_chars = sizeof(str) / sizeof(*str);
```

70. Опасайтесь приведения типов (спорные вопросы Си)

Оператор приведения типов часто понимается неправильно. Приведение типов *не указывает* компилятору "считать эту переменную принадлежащей к этому типу". Оно должно рассматриваться как операция *времени выполнения*, которая создает временную переменную типа, определенного для приведения, затем инициализирует эту временную переменную от операнда. В Си++, конечно, эта инициализация может обернуться очень большими накладными расходами, так как возможен вызов конструктора.

Первое место, где неверное понимание приведения типов может навлечь на вас неприятности, находится в Си, где не требуются прототипы функций. Когда компилятор находит вызов функции без предшествующего прототипа, то полагает, что эта функция возвращает тип **int**. В следующем фрагменте *не говорится* "malloc() на самом деле возвращает указатель, а не тип **int**":

```
int *p = (int *) malloc( sizeof(int) );
```

а, скорее, код говорит "я полагаю, что malloc() возвращает тип **int**, так

как тут нет предшествующего прототипа, и преобразую этот **int** в указатель для присваивания его значения p". Если тип **int** имеет размер 16 бит, а указатель 32-битовый, то вы теперь в глубокой луже. Вызов malloc() может вернуть и 32-битовый указатель, но так как компилятор полагает, что malloc() возвращает 16-битовый **int**, то он игнорирует остальные 16 бит. Затем компилятор обрезает возвращенное значение до 16-бит и преобразует его в 32-битовый тип **int** принятым у него способом, обычно заполняя старшие 16 бит нулями. Если указатель содержал адрес больше, чем 0xffff, что вероятно для большинства компьютеров, то вы просто теряете старшие биты. Единственным способом урегулирования этой проблемы является указание для malloc() соответствующего прототипа, который подскажет, что malloc() возвращает указатель (обычно путем включения файла <stdlib.h>).

Следующий проблема состоит в том, что предыдущее приведение типа может быть вставлено в программу просто для того, чтобы заставить заткнуться компилятор, который наверняка будет выдавать предупреждение о несоответствии типов, если оператор приведения отсутствует. Приведением типов часто злоупотребляют подобным образом — чтобы заглушить компилятор, вместо того, чтобы в самом деле обратить внимание на предупреждение. Многие компиляторы, например, выдают предупреждение о возможном округлении, встретив следующий код:

```
f( int x );

// ...

unsigned y;
f( y );
```

и многие программисты заглушат такой компилятор при помощи f((**int**)y). Несмотря на это, приведение типа не изменит того факта, что тип **unsigned int** может содержать такое значение, которое не поместится в **int** со знаком, поэтому результирующий вызов может не сработать.

Вот сходная проблема, связанная с указателями на функции. Следующий код, случается, работает отлично:

```
some_object array[ size ];
int my_cmp( some_object *p1, some_object *p2 );

qsort( array, size, sizeof(some_object),(
       (*)(void*, void*)) my_cmp );
```

Следующий похожий код просто печально отказывается работать без предупреждающего сообщения:

```
some_object array[ size ];
void foo( int x );

qsort( array, size, sizeof(some_object),
       ((*)(void*, void*)) foo);
```

Функция `qsort()` передает аргументы-указатели в `foo()`, но `foo()` ждет в качестве аргумента **`int`**, поэтому будет использовать значение указателя в качестве **`int`**. Дальше еще хуже — `foo()` вернет мусор, который будет использован `qsort()`, так как она ожидает в качестве возвращаемого значения **`int`**.

Выравнивание также связано с затруднениями. Многие компьютеры требуют, чтобы объекты определенных типов располагались по особым адресам. Например, несмотря на то, что 1-байтый тип **`char`** может располагаться в памяти по любому адресу, 2-байтовый **`short`** должен будет иметь четный адрес, а 4-байтовый **`long`** — четный и кратный четырем. Следующий код вновь не выдаст предупреждений, но может вызвать зависание компьютера во время выполнения:

```
short  x;
long  *lp = (long*)( &x );
*lp = 0;
```

Эта ошибка особенно опасна, потому что `*lp = 0` не сработает лишь тогда, когда `x` окажется по нечетному или не кратному четырем адресу. Может оказаться, что этот код будет работать до тех пор, пока вы не добавите объявление второй переменной типа **`short`** сразу перед `x`, после чего эта программа зависнет.

Один из известных мне компиляторов пытается справиться с этой проблемой, фактически модифицируя содержимое указателя для того, чтобы гарантировать правильный адрес в качестве побочного эффекта от приведения типа. Другими словами, следующий код мог бы на самом деле модифицировать `p`:

```
p = (char *)(long *);
```

71. Немедленно обрабатывайте особые случаи

Пишите свои алгоритмы таким образом, чтобы они обрабатывали вырожденные случаи немедленно. Вот тривиальный пример того, как сделать это неправильно:

```
print( const char *str )
{
```

```
    if( !*str )         // ничего не делать, строка пуста
       return;
    while( *str )
       putchar( *str++ );
}
```

Оператор **if** тут не нужен, потому что этот случай хорошо обрабатывается циклом **while**.

Листинги 2 и 3 демонстрируют более реалистический сценарий. Листинг 2 определяет умышленно наивный заголовок связанного списка и функцию для удаления из него элемента.

Листинг 2. Связанный список: вариант 1

```
typedef struct node
{
   struct node *next, *prev;
   //...
} node;

node *head;

remove( node **headp, node *remove )
{
   // Удалить элемент, на который указывает remove, из
   // списка, на начало которого указывает *headp.

   if( *headp == remove )  // Этот элемент в начале списка.
   {
      if( remove->next )         // Если это не единственный
         remove->next->prev = NULL; // элемент в списке, то
                                    // поместите следующий
                                    // за ним элемент первым
                                    // в списке.
      *headp = remove->next;
   }
   else                // Элемент находится в середине списка
   {
      remove->prev->next = remove->next;
      if( remove->next )
         remove->next->prev = remove->prev;
   }
}
```

Листинг 3 делает то же самое, но я модифицировал указатель на предыдущий элемент в структуре `node`, поместив туда адрес поля `next` предыдущего элемента вместо указателя на всю структуру. Это простое изменение означает, что первый элемент больше не является особым случаем, поэтому функция `remove` становится заметно проще.

Смысл этого состоит в том, что незначительная переделка этой задачи позволяет мне использовать алгоритм, не имеющий особых случаев, этим

самым упрощая программу. Конечно, эта простота не дается бесплатно — теперь стало невозможным перемещение по списку в обратном направлении — но нам ведь это может быть и не нужно.

Листинг 3. Связанный список: вариант 2

```
 1  typedef struct node
 2  {
 3     struct node *next, **prev; // <== К prev добавлен символ *
 4     //
 5  } node;
 6
 7  node *head;
 8
 9  remove( node **headp, node *remove )
10  {
11     if( *(remove->prev) = remove->next ) // если не в конце
12       remove->next->prev = remove->prev; // списка, то уточнить
13                                          // следующий элемент
14  }
```

72. Не старайтесь порадовать lint

Lint является программой проверки синтаксиса для языка Си. (Также имеется версия для Си++ в среде MS-DOS/Windows. Она выпускается фирмой Gimple Software). Хотя эти программы неоценимы при использовании время от времени, они выводят такую кучу сообщений об ошибках и предупреждений, что текст вашей программы будет почти невозможно прочитать, если вы попробуете избавиться от всех них. Оказывается, нужно избегать кода, подобного следующему:

```
(void ) printf("...");
```

Тут вообще нет ничего неверного в присваивании в цикле:

```
while( p = f() )
    g(p);
```

даже если новичок в программировании может воспользоваться символом = вместо ==. (Проблемы новичков не должны приниматься во внимание, когда вы обсуждаете рекомендуемый стиль для опытных профессионалов. Это похоже на принятие закона, который требует, чтобы все велосипеды оснащались боковыми колесиками, потому что двухлетним малышам без них тяжело кататься).

Я читал предложение поручать компилятору поиск небрежных присваиваний (когда вы на самом деле подразумевали сравнение) просто размещая константы в левой части. Например, следующий фрагмент даст ошибку при компиляции, если вы используете = вместо ==:

```
#define MAX 100
```

```
// ...
if ( MAX == x )
// ...
```

В этой идее есть достоинства, но я нахожу, что такой код труднее читается.

73. Помещайте код, динамически распределяющий и освобождающий память, в одном и том же месте

Утечки памяти — выделенную память забыли освободить — являются большой проблемой в приложениях, продолжительность работы которых не ограничена: серверах баз данных, торговых автоматах, операционных системах и так далее. Имеется множество способов для того, чтобы отслеживать эту проблему. Многие программы, например, модифицируют функцию `malloc()` для создания списка выделенных областей памяти, который может просматриваться функцией `free()` для проверки задействованных указателей. Вы также можете доработать заголовочный файл, необходимый для сопровождения списка выделенных областей памяти, путем помещения туда информации о том, откуда было произведено выделение памяти. (Передайте `__LINE__` и `_FILE__` в свою отладочную версию `malloc()`). Список выделенных областей памяти должен быть пуст при завершении работы программы. Если он не пуст, то вы можете его просмотреть и, по крайней мере, разобраться, где была выделена эта память.

Отслеживание — это хорошо, но предупреждение лучше. Если есть возможность, то вы должны размещать `free()` в той же функции, где сделан соответствующий вызов `malloc()`. Например, используйте это:

```
void user( void )              // пользователь
{
    p = malloc( size );
    producer( p );              // изготовитель
    consumer( p );              // потребитель
    free( p );
}
```

вместо этого:

```
void *producer( )
{
    void *p = malloc( size );
    // ...
    return p;
}
void consumer( void *p )
{
    // ...
```

```
    free( p );
}
void user( void )
{
    void *p = producer();
    consumer( p );
}
```

Несмотря на это, вы не сможете всегда делать подобное на Си. Например, предыдущие правила о неинициализированных переменных относятся и к памяти, выделенной `malloc()`. Лучше сделать так:

```
some_object *p = allocate_and_init(); // Не возвращает значения,
                                      // если памяти
недостаточно.
```

чем так:

```
some_object *p = malloc( sizeof(some_object) );
if( !p )
    fatal_error("Недостаточно памяти!")
init();
```

В Си++ эта проблема решается при помощи конструкторов.

74. Динамическая память — дорогое удовольствие

Следующей основной проблемой при использовании `malloc()/free()` (или **`new`**/**`delete`**) является время, требуемое для управления памятью; оно может быть значительным. Я однажды сократил время выполнения на 50% путем замены многочисленных вызовов `malloc()` и `free()` на другую стратегию. Например, если у вас есть очень активно используемая структура из одинаковых объектов, то вы можете использовать нечто подобное коду из листинга 4 для управления членами структуры данных.

Листинг 4. Управление собственным списком высвобожденных элементов

```
 1  typedef struct some_class
 2  {
 3    struct some_class *next;
 4    //...
 5  }
 6  some_class;
 7
 8  static some_class *free_list = NULL;
 9  //-----------------------------------------------------------
10  free_object( some_class *object )
11  {
12    // Вместо того, чтобы передать память из-под объекта
13    // функции free(), свяжите ее с началом списка
14    // высвобожденных элементов.
15    object->next = free_list;
```

```
 16   free_list = object;
 17 }
 18 //-------------------------------------------------------
 19 free_all_objects( )
 20 {
 21   // Высвободить все объекты в список высвобожденных
 22   // элементов. Сортировка этих объектов по базовому адресу
 23   // перед началом цикла улучшит скорость работы по
 24   // освобождению, но я не делаю здесь этого. (Для
 25   // сортировки связанных списков превосходно подходит
 26   // алгоритм Quicksort).
 27
 28   some_object *current;
 29   while( free_list )
 30   {
 31     current = free_list;
 32     free_list = current->next;
 33     free( current );
 34   }
 35 }
 36 //-------------------------------------------------------
 37 some_class   *new_object( )
 38 {
 39   // Если в списке высвобожденных элементов имеется объект,
 40   // то используйте его. Размещайте объект посредством
 41   // malloc(),только если список высвобожденных объектов
 42   // пуст.
 43   some_class *object;
 44   if( free_list )
 45   {
 46     object = free_list;   // передайте память для объекта
 47     free_list = object->next;
 48   }
 49   else
 50   {
 51     // Вы можете улучшить производительность еще более,если
 52     // будете размещать объекты в памяти по 100, а не по 1
 53     // за раз, но это усложнит функцию free_all_objects().
 54
 55     object = malloc( sizeof(some_class) );
 56   }
 57
 58   if( object )
 59 // поместите здесь инициализирующий код
 60
 61 return object;
 62 }
```

75. Тестовые подпрограммы не должны быть интерактивными

Мне часто показывают тестовые программы, имеющие усовершенствованный интерактивный интерфейс пользователя. Это не

только является пустой тратой времени, но и не может быть использовано для тщательного тестирования. Люди, сидящие за клавиатурами и пробующие все, что им приходит в голову, недостаточно методичны. Поэтому для систематического выполнения функций, подлежащих проверке, лучше использовать неинтерактивную тестовую функцию.

И, кстати, не удаляйте эту тестовую процедуру; просто используйте предложение **`#ifdef`** `TEST` для включения или выключения этой подпрограммы из компиляции. Если вы похожи на меня, то вы утром удалите эту тестовую функцию, а уже к обеду она вам понадобиться снова, даже если вы не пользовались ей последние два года.

76. Сообщение об ошибке должно подсказывать пользователю, как ее исправить

Когда-то, во времена CP/M, отладчик DDT имел единственное сообщение об ошибке. Не имело значения, что вы натворили, он всегда говорил:

```
?
```

Несмотря на то, что подобная обработка ошибок, к счастью, больше не является нормой, я все еще вижу ее, хотя и с более прихотливыми украшениями. Несколько программ для Windows, мне принадлежащие, просто издают звуковой сигнал, когда я ввожу неверное значение в диалоговом окне, заставляя меня гадать, какое же значение верное. Я часто видел сообщения об ошибках, которые давали мне обильное количество информации о том, что я сделал неправильно, не давая подсказки, как это делается правильно. В самой печальной ситуации диалоговое меню предлагает вам ввести число, но когда вы вводите неверное число, оно выводит такое сообщение:

```
Неверное значение.
```

без намека на то, какое значение будет верным. Иногда диалоговое окно отказывается закрыться до тех пор, пока вы не введете верное значение. Иногда программа захватывает управление компьютером, и вы не можете переключиться в другое приложение, завершив то, в котором с вами случилась беда. (В Windows это "свойство" называется системно-модальным диалоговым окном. Пожалуйста, не пользуйтесь им). И вот вы перед выбором: завершить программу, нажав Ctrl-C или ее эквивалент (если можете), отключить питание или наобум набирать числа на клавиатуре до тех пор, пока вам не удастся наткнуться на то, которое удовлетворит программу ввода данных. Вопрос состоит в том, что займет у вас больше времени: повторить последние три часа работы, которую вы не подумали сохранить перед вызовом диалогового окна, или потратить

еще три часа, играя в "холодно-горячо" с диалоговым окном. Это как раз та ситуация, из-за которой у компьютеров дурная слава.

Сообщение об ошибке должно подсказывать вам, как исправить эту ситуацию, что-нибудь типа:

```
Числа должны быть в диапазоне от 17 до 63 включительно.
```

или:

```
Даты должны иметь формат дд-мм-гггг.
```

Должен быть какой-то способ (типа клавиши "help/справка") для получения дополнительной информации, если она вам нужна. Наконец, у вас должен быть способ безопасного прекращения процесса ввода данных (типа клавиши "Выход").

77. Не выводите сообщения об ошибке, если она исправима

Библиотечные подпрограммы (и большинство подпрограмм на языках низкого уровня) не должны выводить сообщений об ошибках. Они должны возвращать код ошибки, который может быть проверен вызывающей подпрограммой. Таким способом эта подпрограмма может предпринять какие-то корректирующие действия. Например, если рекурсивная функция-сортировщик сообщила об ошибке, выйдя за пределы стека, то вызвавшая ее подпрограмма не сможет попытаться заново выполнить сортировку с использованием другого, нерекурсивного алгоритма. Пользователь должен увидеть сообщение об ошибке, даже если вторая попытка сортировки будет удачной.

78. Не используйте системно-зависимых функций для сообщений об ошибках

Многие среды с оконным интерфейсом не поддерживают понятия стандартного устройства для вывода или для сообщений об ошибках. (В этих средах вызовы `printf()` или `fprintf(stderr, ...)` обычно игнорируются). Если вы будете основываться на догадках о своей среде, то обнаружите, что вам необходимы значительные доработки просто для того, чтобы перекомпилировать код для новой среды).

Минимум вашей защиты может быть таким:

```
#define error        printf
#define terminate(x) ExitProcess(x)
```

после чего используйте:

```
if ( some_error )
```

```
{
    error("Тут что-то не так ");
    terminate( -1 );
}
```

Вот более гибкое решение:

```
#include <stdio.h>
#include <stdarg.h>

#ifdef   WINDOWS
void error( const char* format, ... )
{
    char buf[255]; // надеемся, что такой размер будет
                   // достаточен
    va_list  args;
    va_start(  args,  format );
    if( vsprintf( buf, format, args ) < sizeof(buf) )
        ::MessageBox(NULL,buf, "*** ОШИБКА ***",
                     MB_OK | MB_ICONEXCLAMATION );
    else
    {
        ::MessageBox(NULL,
          "Переполнение буфера при печати сообщения об ошибке.",
          "Фатальная ошибка",
          MB_OK | MB_ICONEXCLAMATION );
        ExitProcess( -1 );
    }
    va_end( args );
}
#elif MOTIF
// Здесь следует функция обработки ошибки, используемая
// ОС Motif
#else
void error( const char* format, ... )
{
    va_list args;
    va_start( args, format );
    vfprintf(stderr, format, args );
    va_end ( args );
}
#endif
```

Часть

6

Препроцессор

Многие свойства языка Си++ делают препроцессор Си менее важным, чем он был по традиции. Тем не менее, препроцессор иногда нужен даже в программе на Си++ и, естественно, остается неотъемлемой частью программирования на Си. Правила в этой главе книги посвящены правильному использованию препроцессора.

Я должен сказать, что многие из тех макросов, которые мне пришлось видеть, имеют довольно сомнительные основания с точки зрения удобочитаемости и сопровождения. Я часто предпочитаю не пользоваться макросом, если эквивалентная функция сделает ту же самую работу в более удобочитаемой манере, и никогда не хочу использовать макрос, вызывающий побочный эффект (будет рассмотрено ниже). В Си++ я *никогда* не пользуюсь макросами с параметрами, используя вместо них встроенные (**`inline`**) функции или шаблоны, которые расширяются компилятором до встроенных функций. Макрос с параметрами даже в языке Си должен быть методом, используемым лишь в крайнем случае. В макросах трудно обнаруживать ошибки (так как они не выполняются в пошаговом режиме), их трудно читать и, в лучшем случае, сложно сопровождать. Используйте их лишь тогда, когда скорость выполнения действительно является критерием, подтверждаемым фактическим тестированием кода с их использованием. Таким образом, эта глава книги содержит правила для тех случаев, где препроцессор является единственным решением проблемы.

79. Все из одного *.h* файла должно быть использовано в, по меньшей мере, двух *.c* файлах

Это правило говорит само за себя — не загромождайте область глобальных имен идентификаторами, которые не используются глобально. Если идентификатор не используется вне текущего файла, то он должен объявляться лишь в области действия текущего файла. Если этот не используемый совместно идентификатор является глобальной переменной или функцией, то он должен быть объявлен статическим.

Заметьте, что статические функции находят применение даже в Си++. Имеется тенденция помещать все основные функции, используемые любым из обработчиков сообщений, в собственно определение класса. Иногда локальная статическая функция в файле *.cpp* делает эту работу так же хорошо, и нет нужды загромождать прототипом этой функции определение класса.

80. Используйте вложенные директивы `#include`

Хотя большинство из правил в этой главе говорят вам, как избежать использования препроцессора, механизм включения файлов директивой **`#include`** является обязательной функцией препроцессора как в Си, так и в Си++. Тем не менее, даже здесь существуют проблемы.

Это на самом деле плохая идея - требовать, чтобы кто-нибудь включал файл, способный включать в себя следующий. Я всегда располагаю директивы **`#include`** без определенного порядка или забываю вставить одну из них. Следовательно, заголовочный файл должен всегда включать те файлы, которые определяют то, что используется в текущем заголовочном файле. Вследствие того, что могут возникнуть проблемы, если компилятор прочитает какой-нибудь *.h* файл более одного раза, вы должны предпринять шаги для предотвращения многократной обработки одного и того же файла. Помещайте строки типа:

```
#ifndef FILENAME_H_
#define FILENAME_H_
```

в начале каждого заголовочного файла, и вставляйте соответственно:

```
#endif // FILENAME_H_
```

в конце. Так как константа `FILENAME_H_` будет уже определена к моменту второй попытки препроцессора обработать этот файл, то его содержание при втором проходе будет проигнорировано.

81. Вы должны быть всегда способны заменить макрос функцией

Это вариант для макросов правила "не нужно неожиданностей (без сюрпризов) ". Что-то, похожее внешне на функцию, должно действовать подобно функции, даже если это на самом деле макрос. (По этой причине я иногда предпочитаю записывать имя макроса заглавными буквами не полностью, если его поведение сильно напоминает функцию. Хотя я всегда использую все заглавные, если у макроса есть побочные эффекты). При этом возникает насколько вопросов.

Во-первых, макрос не должен использовать переменные, не передаваемые в качестве аргументов. Вот наихудший из возможных способов в таких обстоятельствах:

Следующий код находится в заголовочном файле:

```
#define end() while(*p) \
        ++p
```

а этот — в файле *.c*:

```
char *f( char *str )
{
    char *p = str;
    end();
    // ...
    return p;
}
```

Здесь для сопровождающего программиста имеется несколько неприятных сюрпризов. Во-первых, переменная `p` явно не используется, поэтому появляется искушение стереть ее, разрушая этим код. Аналогично, программа разрушится, если имя `p` будет заменено другим. Наконец, будет большой неожиданностью то, что вызов `end()`, который выглядит внешне как вызов обычной функции, будет модифицировать `p`.

Первая попытка урегулирования этой проблемы может выглядеть следующим образом. В заголовочном файле:

```
#define end(p) while(*p)  \
               ++p
```

и в файле *.c*:

```
char *f( char *str )
{
    end(str);
    // ...
    return str;
}
```

Но теперь макрос все еще необъяснимо модифицирует `str`, а нормальная функция Си не может работать таким образом. (Функция Си++ может, но не должна. Я объясню почему в той главе книги, которая посвящена Си++). Для модификации строки `str` в функции вы должны передать в нее ее адрес, поэтому то же самое должно быть применимо к макросу. Вот третий (наконец-то правильный) вариант, в котором макрос `end()` попросту заменен функцией с таким же именем. В заголовочном файле:

```
#define end(p) while(*(*p)) \
   ++(*p)
```

и в файле *.c*:

```
char *f( char *str )
{
    end(&str);
    // ...
    return str;
}
```

Вместо `end(&str)` будет подставлено:

```
while(*(*&p))
    ++(*&p)
```

и `*&p` — это то же самое, что и `p`, так как знаки `*` и `&` отменяют друг друга — поэтому макрос в результате делает следующее:

```
while(*(p))
    ++(p)
```

Вторая проблема с макросом в роли функции возникает, если вы желаете выполнить в макросе больше, чем одно действие. Рассмотрим такой макрос:

```
#define two_things()    a();b()

if( x )
    two_things();
else
    something_else();
```

который будет расширен следующим образом (тут я переформатировал, чтобы сделать происходящее неприятно очевидным):

```
if ( x )
    a();
b();
else
    something_else();
```

Вы получаете сообщение об ошибке "у **else** отсутствует предшествующий оператор **if**". Вы не можете решить эту проблему, используя лишь фигурные скобки. Переопределение макроса следующим образом:

```
#define two_things()  { a(); b(); }
```

вызовет такое расширение:

```
if( x )
{
    a();
    b();
}
;
else
    something_else();
```

Эта вызывающая беспокойство точка с запятой — та, что следует после `two_things()` в вызове макроса. Помните, что точка с запятой сама по себе является законным оператором в Си. Она ничего не делает, но она законна. Вследствие этого **else** пытается связаться с этой точкой с запятой, и вы получаете то же самое "у **else** отсутствует предшествующий оператор **if**".

Не нужно говорить, что, несмотря на то, что макрос выглядит подобно вызову функции, его вызов может не сопровождаться точкой с запятой. К счастью, для этой проблемы имеется два настоящих решения. Первое из них использует малоизвестный оператор "последовательного вычисления" (или запятую):

```
#define two_things() ( a(), b() )
```

Эта запятая — та, что разделяет подвыражения в инициализирующей или инкрементирующей частях оператора **for**. (Запятая, которая разделяет аргументы функции, не является оператором последовательного вычисления). Оператор последовательного вычисления выполняется слева направо и получает значение самого правого элемента в списке (в нашем случае значение, возвращаемое `b()`). Запись:

```
x = ( a(),b() );
```

означает просто:

```
a();
x = b();
```

Если вам все равно, какое значение имеет макрос, то вы можете сделать нечто подобное, используя знак плюс вместо запятой. (Выражение:

```
a()+b();
```

в отдельной строке совершенно законно для Си, где не требуется, чтобы результат сложения был где-нибудь сохранен). Тем не менее, при знаке плюс порядок выполнения не гарантируется; функция `b()` может быть вызвана первой. Не путайте приоритеты операций с порядком выполнения. Приоритет просто сообщает компилятору, где неявно размещаются круглые скобки. Порядок выполнения вступает в силу после того, как все круглые скобки будут расставлены по местам. Невозможно добавить дополнительные скобки к `((a())+(b()))`. Операций последовательного вычисления гарантированно выполняется слева направо, поэтому в нем такой проблемы нет.

Я должен также отметить, что операция последовательного вычисления слишком причудлива, чтобы появляться в нормальном коде. Я использую ее лишь в макросах, сопровождая все обширными комментариями, объясняющими, что происходит. Никогда не используйте запятую там, где должна быть точка к запятой. (Я видел людей, которые делали это, чтобы не использовать фигурные скобки, но это страшно даже пересказывать).

Второе решение использует фигурные скобки, но с одной уловкой:

```
#define two_things()    \
        do              \
        {               \
           a();         \
           b();         \
        } while( 0 )

if( x )
    two_things();
else
    something_else();
```

что расширяется до:

```
if( x )
    do
    {
        a();
        b();
    } while ( 0 ) ; // <== точка с запятой связывается с
                    // оператором while ( 0 )
else
    something_else();
```

Вы можете также попробовать так:

```
#define two_things() \
if( 1 )              \
```

```
{                       \
   a();                 \
   b();                 \
} else
```

но я думаю, что комбинация **do** с **while** (0) незначительно лучше.

Так как вы можете объявить переменную после любой открытой фигурной скобки, то у вас появляется возможность использования предшествующего метода для определения макроса, имеющего по существу свои собственные локальные переменные. Рассмотрим следующий макрос, осуществляющий обмен значениями между двумя целочисленными переменными:

```
#define swap_int(x,y)   \
        do              \
        {               \
           int x##y;    \
           x##y = x;    \
           x = y;       \
           y = x##y     \
        }               \
        while (0)
```

Сочетание ## является оператором конкатенации в стандарте ANSI Си. Я использую его здесь для обеспечения того, чтобы имя временной переменной не конфликтовало с любым из имен исходных переменных. При данном вызове:

```
swap(laurel, hardy);
```

препроцессор вначале подставляет аргументы обычным порядком (заменяя x на laurel, а y на hardy), давая в результате следующее имя временной переменной:

```
int laurel##hardy;
```

Затем препроцессор удаляет знаки решетки, давая

```
int laurelhardy;
```

Дополнительная польза от возможности замены макросов функциями заключается в отладке. Иногда вы хотите, чтобы что-то было подобно макросу по эффективности, но вам нужно при отладке установить в нем точку прерывания. Используйте для этого в Си++ встроенные функции, а в Си используйте следующие:

```
#define _AT_LEFT(this) ((this)->left_child_is_thread ? NULL\
                       :(this)->left)

#ifdef DEBUG
static tnode *at_left(tnode *this) { return _AT_LEFT(this); }
```

```
#else
#   define at_left(this) _AT_LEFT(this)
#endif
```

Я закончу это правило упоминанием о еще двух причудливых конструкциях, которые иногда полезны в макросе, прежде всего потому, что они помогают макросу расширяться в один оператор, чтобы избежать проблем с фигурными скобками, рассмотренных ранее. Положим, вы хотите, чтобы макрос по возможности расширялся в единственное выражение. Оператор последовательного вычисления достигает этого в ущерб читаемости, и наряду с ним я никогда не использую формы, показанные в таблице 1, по той же причине — их слишком трудно читать. (Коли на то пошло, я также не использую их в макросах, если я могу достичь желаемого каким-то другим способом).

Таблица 1. Макросы, эквивалентные условным операторам

Этот код:	Делает то же самое, что и:
`( a && f() )`	`if( a )` `  f();`
`( b || f() )`	`if( !b )` `  f();`

`( z ? f() : g())`	`if( z )`
	`  f();`
	`else`
	`  g();`

Первые два выражения опираются на тот факт, что вычисления в выражении с использованием операций `&&` и `||` гарантированно осуществляются слева направо и прекращаются сразу, как только устанавливается истина или ложь. Возьмем для примера выражение `a && f()`. Если `a` ложно, то тогда не важно, что возвращает `f()`, так как все выражение ложно, если любой из его операндов значит ложь. Следовательно, компилятор никогда не вызовет `f()`, если `a` ложно, но он должен вызвать `f()`, если `a` истинно. То же самое применимо и к `b`, но здесь `f()` вызывается, если `b`, напротив, ложно.

81.1. Операция `?:` не то же самое, что и оператор `if/else`

Последняя строка в таблице 1 относится к другому спорному вопросу. Условная операция — это простой оператор. Она используется лишь в выражении и передает значение. Условная операция является не очень привычной заменой для оператора **`if`**/**`else`**, но не менее, чем операции `&&` или `||` приемлемы для замены простого **`if`**. Хотя большинство людей

и не принимают во внимание замену:

```
if( z )
    i = j;
else
    i = k;
```

на:

```
z && (i = j);
z || (i = k);
```

Мне довелось случайно увидеть подобное этому, но с использованием условной операции:

```
z ? (i = j) : (i = k) ;
```

Все предыдущие фрагменты в равной мере способны сбить с толку. Следующий код показывает, как надлежащим образом использовать условную операцию, и ее результат яснее (т.е. лучше), чем у равноценного оператора **if/else**:

```
i = z ? j : k ;
```

81.2. Помещайте тело макроса и его аргументы в круглые скобки

Это правило одно из основных, но я обнаружил, что множество людей, пользующихся Си ежедневно, его забыли. Вот классическая задача:

```
#define TWO_K 1024 + 1024
```

что при использовании в:

```
10 * TWO_K
```

расширяется до:

```
10* 1024 + 1024
```

вычисляемого как:

```
(10 * 1024) + 1024
```

Решаем эту задачу при помощи круглых скобок:

```
#define TWO_K    (1024 + 1024)
```

Вот сходная задача в следующем фрагменте кода:

```
#define SQUARE(x)    (x * x)
```

Определено:

```
SQUARE(y + 1);
```

что расширяется макросом до:

```
y + 1 * y + 1
```

и вычисляется как:

```
y + (1 * y) + 1
```

И вновь круглые скобки приходят на помощь. Следующее определение:

```
#define SQUARE(x)     ((x) * (x))
```

расширяется до:

```
((y + 1) * (y + 1))
```

82. `enum` и `const` лучше, чем макрос

Директива **`#define`** должна быть вашим последним средством при определении значения константы. Рассмотрим следующую рассмотренную ранее распространенную ошибку:

```
#define TWO_K 1024 + 1024

x = TWO_K * 10
```

что в результате вычисления дает 11264 (1024+(1024*10)) вместо требуемых 20480. Определение перечисления типа:

```
enum { two_k = 1024 + 1024 };
```

или константы типа:

```
const  int  Two_k = 1024 + 1024;
```

не вызывает трудностей, связанных с макросом. И круглые скобки не требуются.

Перечисление **`enum`** на несколько очков превосходит константу: во-первых, определение **`const int`** в языке Си на самом деле выделяет память под тип **`int`** и инициализирует ее. Вы не можете модифицировать эту область памяти, но память при этом занята. Следовательно, определение константы в Си нельзя поместить в заголовочном файле; вы нужно будет воспользоваться модификатором **`extern`** как для какой-нибудь глобальной переменной. (В Си++ все это несущественно, так как там память выделяется лишь тогда, когда вы определяете адрес константы или передаете его по ссылке. Определения констант в Си++ могут — а на деле часто и должны — помещаться в заголовочном файле).

Перечисление отличается тем, что память для него никогда не выделяется. Подобно макросу, оно может вычисляться во время компиляции. Следовательно, при использовании вами перечисления не

происходит потери производительности.

Второй проблемой является порча области глобальных имен. Область действия перечисления легко ограничивается. Например, в следующем фрагменте кода перечисление `default_i` действует лишь внутри функции `f()`:

```
void f(  int  i  )
{
    enum { default_i = 1024 };

    if ( !i )
    i = default_i ;
}
```

В фрагменте:

```
void f(  int  i  )
{
#define DEFAULT_I 1024

    if ( !i )
       i = DEFAULT_I ;
}
```

макрос `DEFAULT_I` виден всем функциям, чьи определения следуют после определения этого макроса. Если `DEFAULT_I` определяется в заголовочном файле, то он будет виден в нескольких файлах — даже если он не используется кодом в этих файлах. Та же самая проблема касается также константы, определенной на глобальном уровне.

Перечислитель **`enum`** особенно полезен в Си++, потому что он может быть ограничен областью действия класса и инициализироваться в самом определении класса вместо конструктора. Эти вопросы рассматриваются далее в той части книги, что посвящена правилам Си++.

Наконец, перечислитель может быть использован в качестве аргумента оператора **`case`** и размера при объявлении массива. Ни в одной из указанных ситуаций константа использоваться не может[♣].

83. Аргумент параметризированного макроса не должен появляться в правой части более одного раза

Макрос `SQUARE()` даже в своем модифицированном виде представил выше серьезную проблему. Дано:

```
#define SQUARE(x)    ((x)*(x))
```

[♣] С этим утверждением автора, так и следующим за ним примером инкрементирования аргумента макроса нельзя согласиться. — *Ред*.

Выражение `SQUARE(++x)` дважды инкрементирует `x`. После чего макрос в этом случае дает неверный результат. Если `x` вначале содержит 2, то `SQUARE(++x)` вычисляется как 3 * 4. Такое поведение есть пример побочного эффекта макроса — ситуации, когда макрос ведет себя неожиданно.

`SQUARE(++x)` также показывает пример ситуации, в которой использование макроса просто слишком рискованно для оправдания сложностей сопровождения. Встроенная функция Си++ или шаблон, расширяемый до встроенной функции, являются более удачными решениями. Даже в Си простую функцию с неудачными аргументами легче сопровождать, чем эквивалентный макрос:

```
double  square(  double  x )
{
    return x * x;
}
```

Но, тем не менее, у меня есть серьезное сомнение в том, что использование функции для скрытия простого умножения является стоящим делом.

83.1. Никогда не используйте макросы для символьных констант

Например:

```
#define SPACE  '  '
```

имеет смысл, если только вы намерены использовать вместо пробела другой символ (как если бы вы испытывали, например, программу для замены символов табуляции).

Никогда не делайте так:

```
#define SPACE   0x20
```

Действительное значение символьной константы для пробела (`'  '`) изменяется компилятором так, чтобы оно соответствовало операционной среде, для которой ведется компиляция. Для среды, поддерживающей ASCII, это значение будет `0x20`, а для EBDCDIC — уже нечто другое. Не думайте, что у какого-то символа свое постоянное значение.

84. Если все альтернативы отпали, то используйте препроцессор

Мы увидим в главе, посвященной Си++, что препроцессор Си не играет большой роли в Си++. Хотя есть немного мест, где он все еще кстати. Вот первое из них:

```
#ifdef  DEBUG
#       define D(x)   x
#else
#       define D(X) /* пусто */
#endif
```

Вместо макроса `D()` подставляется его аргумент, если вы занимаетесь отладкой, иначе он расширяется до пустой строки. Он используется так:

```
f()
{
    D( printf("Это отладочная информация\n"); )
}
```

В данном случае аргументом `D()` является целиком оператор `printf()`, который исчезает после того, как вы закончите отладку.

Другой подобный вариант использования кстати, когда вы должны инициализировать те несколько неизбежных глобальных переменных в большой программе. Проблема заключается в синхронизации объявлений переменных (в заголовочном файле) с их определениями (в файле *.c*), где реально выделяется память и переменные инициализируются. Вот образец заголовочного файла:

```
#ifdef ALLOC
#       define    I(x)         x
#       define    EXPORTED    /* пусто */
#else
#       define    I(x)        /* пусто */
#       define    EXPORTED    extern
#endif

EXPORTED int          glob_x[10] I( ={1, 2, 3, 4} );
EXPORTED some_object glob_y     I( ("конструктор", "аргументы"));
```

В определенном месте своей программы (я обычно делаю это в файле с именем *globals.cpp*) вы помещаете следующие строки:

```
#define  ALLOC
#include "globals.h"
```

Далее везде вы просто включаете этот файл без предварительной директивы **#define** ALLOC. Когда вы компилируете *globals.cpp*,

директива **#define** ALLOC вызывает следующую подстановку:

```
/* пусто */ int         glob_x[10]  ={1, 2, 3, 4} ;
/* пусто */ some_object glob_y ("конструктор", "аргументы");
```

Отсутствие **#define** ALLOC везде приводит к следующей подстановке:

```
extern  int          glob_x[10]   /* пусто */ ;
extern  some_object  glob_y       /* пусто */ ;
```

Последним примером использования препроцессора будет макрос ASSERT(), который выводит сообщение об ошибке и завершает программу, лишь если вы осуществляете отладку (директивой **#define** определена константа DEBUG) и аргумент ASSERT() имеет значение "ложь". Он очень удобен для тестирования, например, аргументов типа указателей со значением NULL. Вариант ASSERT(), используемый в виде:

```
f( char *p)
{
   ASSERT( p, "f() : Неожиданный аргумент NULL." );
}
```

определяется следующим образом:

```
#ifdef  DEBUG
#define ASSERT(условие, сообщение)
if (  !(условие) ) \
{\
   fprintf(stderr, "ASSERT(" #условие ") НЕ ВЫПОЛНЕНО "\
                   "[Файл " __FILE__ ", Строка %d]:\n\t%s\n",\
                   __LINE__, (сообщение) );\
   exit( -1 );\
}\
else
#else
#   efine ASSERT(c,m)    /*  пусто  */
#endif
```

В вышеуказанном примере ASSERT() выводит следующую строку при отрицательном результате проверки:

```
ASSERT(p) НЕ ВЫПОЛНЕНО [Файл whatever.cpp, Строка 123]:
            f() : Неожиданный аргумент NULL.
```

и затем выходит в вызывающую программу. Он получает текущее имя файла и номер строки от препроцессора, используя предопределенные макросы __FILE__ и __LINE__. Условие, вызывающее отрицательный результат, выводится посредством оператора получения строки ANSI Си (символ #), который фактически окружает расширенный аргумент кавычками после выполнения подстановки аргумента. Строка #условие расширяется до "p" в настоящем примере). Затем вступает в действие

обычная конкатенация строк Си для слияния вместе разных строк, создавая единый отформатированный аргумент для `fprintf()`.

Здесь следует использовать препроцессор, потому что вам нужно вывести на консоль имя файла и номер строки, для которых выполнена проверка. Встроенная функция Си++ может вывести лишь имя того файла с номером строки, в котором определена встроенная функция.

Все компиляторы, поддерживающие стандарт ANSI Си, должны реализовывать макрос `assert(expr)` в заголовочном файле *assert.h*, но макрос ANSI Си не может выводить заказанное сообщение об ошибке. Макрос ANSI Си `assert()` действует, если не определена константа `NDEBUG` (вариант по умолчанию).

Часть

7

Правила, относящиеся к языку Си

В этой главе рассматриваются специфичные для Си правила программирования, не встречавшиеся в предыдущих разделах.

85. Подавляйте демонов сложности (часть 2)

Демоны запутанности особенно опасны в Си. Кажется, что этот язык сам собой поощряет выбор неестественно усложненных решений для простых задач. Последующие правила посвящаются этой проблеме.

85.1. Устраняйте беспорядок

Язык Си предоставляет богатый набор операторов и, как следствие, предлагает множество способов ничего не делать, что и иллюстрируется примерами из таблицы 2.

Таблица 2. Как ничего не делать в Си

Плохо	Хорошо	Комментарии
`type *end = array;` `end += len-1;`	`type *end =` `array+(len-1)`	*Инициализируйте при объявлении.*
`while (*p++ != '\0')`	`while ( *p++)`	`!=0` *ничего не делает в выражении отношения*
`while (gets(buf) != NULL)`	`while (gets(buf) )`	
`if ( p != NULL )`	`if ( p )`	
`if ( p == NULL )`	`if ( !p )`	
`if (условие != 0)`	`if ( условие )`	
`if (условие == 0)`	`if ( !условие )`	

`if( условие )`
`  return TRUE;`
`else`
`  return FALSE;`
	`return условие;`	*(или* `return` условие `!= 0`*). Если оно не было верным, то вы не сможете выполнить* `return TRUE.`

`return условие?0:1;` `return условие?1:0;`	`return !условие;` `return условие!=0;`	*Используйте соответствующий оператор. Операторы отношения типа* `!` *и* `!=` *выполняют по определению сравнение с* `1` *или* `0`*.*

`++x;`
`f(x);`
`--x;`
	`f( x-1 );`	*Не модифицируйте значение, если вам после этого не нужно его использовать более одного раза.*

`return ++x;`	`return x+1;`	*См. предыдущее правило.*
`int x;` `f( (int)x );`	`f(x);`	*Переменная* `x` *и так имеет тип* `int.`
`(void) printf("все в порядке");`	`printf("все в порядке");`	*Попросту опускайте возвращаемый тип, если*

		он вам не нужен.

`if ( x > y )`
`else if ( x < y )`
`else if ( x ==y )`
	`if ( x > y )`
	`else if ( x < y )`
	`else`
		Если первое значение не больше и не меньше второго, то они должны быть равны.

`*(p+i)`	`p[i];`	*Это, по сути, единственное исключение из приводимого ниже в данной главе правила об использовании указателей. При реализации действительно случайного доступа к элементам массива запись со скобками легче читается, чем вариант с указателем, в равной степени неэффективный при случайном доступе.*

Раз мы уж заговорили о *ничегонеделаньи*, то имейте в виду, что Си с удовольствием допускает выражения, которые ничего не делают. Например, следующий оператор, показываемый полностью, совершенно законен и даже не вызовет предупреждающего сообщения компилятора:

```
a + b;
```

Конечно, если вы хотели записать:

```
a += b;
```

то вы, должно быть, попали в беду.

85.2. Избегайте битовых масок; используйте битовые поля

Многие программисты, в особенности те, кто начинал жизнь с языком ассемблера, привыкли пользоваться битовыми масками, а не битовыми полями. Мне довелось видеть много программ, подобных следующей:

```
struct fred
{
   int status;
   // ...♣
};

#define CONDITION_A    0x01
#define CONDITION_B    0x02
```

♣ Комментарий в языке Си должен быть заключен в `/*  */`. — *Ред.*

```
#define CONDITION_C    0x03

#define SET_CONDITION_A(p)    ((p)->status |= CONDITION_A)
#define SET_CONDITION_B(p)    ((p)->status |= CONDITION_B)
#define SET_CONDITION_C(p)    ((p)->status |= CONDITION_C)

#define CLEAR_CONDITION_A(p)  ((p)->status &= ~CONDITION_A)
#define CLEAR_CONDITION_B(p)  ((p)->status &= ~CONDITION_B)
#define CLEAR_CONDITION_C(p)  ((p)->status &= ~CONDITION_C)

#define IN_CONDITION_A(p)     ((p)->status & CONDITION_A)
#define IN_CONDITION_B(p)     ((p)->status & CONDITION_B)
#define IN_CONDITION_C(p)     ((p)->status & CONDITION_C)

#define POSSIBILITIES(x)      ((x) & 0x0030)
#define POSSIBILITY_A          0x0000
#define POSSIBILITY_B          0x0010
#define POSSIBILITY_C          0x0020
#define POSSIBILITY_D          0x0030
```

Это означает необходимость в дополнение к полю из структуры данных сопровождать 17 макросов, которые к тому же будут, вероятно, спрятаны где-то в заголовочном файле, а не в том, где они используются. Ситуация еще более ухудшится, если вы не включите эти макросы и организуете проверки прямо в программе. Что-нибудь типа:

```
if ( struct.status &= ~CONDITION_A )
 // ...
```

по меньшей мере, с трудом читается. Еще хуже нечто, подобное следующему:

```
struct.status = POSSIBILITY_A;
if ( POSSIBILITIES(struct.status) == POSSIBILITY_A )
// ...
```

Лучшее решение использует битовые поля; они не требуют дополнительного места и заведомо эффективно реализуются на большинстве машин. (Некоторые люди утверждают, что второй пример лучше, чем битовое поле, потому что здесь нет неявного сдвига, но многие машины поддерживают команду проверки бита, которая устраняет какую-либо потребность в сдвиге, который в случае своего использования вызывает очень незначительные накладные расходы. Необходимость в устранении ненужной путаницы обычно перевешивает подобные соображения о снижении эффективности).

```
enum { possibility_a, possibility_b, possibility_b,
possibility_d };

struct fred
```

```
{
unsigned in_condition_a : 1;
unsigned in_condition_b : 1;
unsigned in_condition_c : 1;

unsigned possibilities : 2;
};
```

Вам теперь вообще не нужны макросы, потому что код, подобный следующему, превосходно читается без них:

```
struct fred flintstone;
flintstone.in_condition_a = 1;
if ( flintstone.in_condition_a )
// ...

flintstone.possibilities = possibility_b;

if ( flintstone.possibilities == possibility_a )
// ...
```

Единственным очевидным исключением из этого правила является взаимодействие с архитектурами со страничной организацией памяти; битовые поля не гарантируют какого-то упорядочивания в типе **int**, из которого выделяются биты.

85.3. Не используйте флагов завершения

Флаг завершения типа "готов" едва ли нужен в Си или Си++. Его использование просто добавляет одну лишнюю переменную в процедуру. Не делайте так:

```
BOOL готов = FALSE;
while ( !готов )
{
    if ( некоторое_условие() )
    готов = 1;
}
```

Поступайте следующим образом:

```
while ( 1 )
{
    if ( некоторое_условие() )
    break;
}
```

Многие программисты привыкли использовать флаги завершения, когда они учились программированию, в основном потому, что языки программирования типа Паскаля не поддерживают богатый набор управляющих операторов, имеющийся в Си.

Единственным исключением из этого правила является выход из вложенных циклов в Си++, где оператор **goto** может привести к пропуску программой вызова конструктора или деструктора. Эта проблема была рассмотрена в правиле 54.

85.4. Рассчитывайте, что ваш читатель знает Си

Не делайте чего-то подобного этому:

```
#define SHIFT_LEFT(x, bits)      ((x) << (bits))
```

Программисты на Си знают, что << означает "сдвиг влево". Аналогично, не делайте таких вещей:

```
x++;        // инкрементировать x
```

Проблема в том, что комментарии, подобные вышеуказанному, часто встречаются в учебниках по языку программирования, ибо их читатель не знаком с Си. Поэтому вы не должны делать вывод, что раз вы видите их в таком учебнике, то это является хорошей повсеместной практикой.

85.5. Не делайте вид, что Си поддерживает булевый тип (#define TRUE)

Нижеследующее может скорее вызвать проблемы, чем нет:

```
#define TRUE    1
#define FALSE   0
```

Любая отличная от нуля величина в Си означает истину, поэтому в следующем фрагменте `f()` может вернуть совершенно законное значение "истина", которое не совпало с `1`, и проверка даст отрицательный результат:

```
if( f() == TRUE ) // Вызов не выполняется, если f() возвращает
                  // значение "истина", отличное от 1.
    // ...
```

Следующий вариант надежен, но довольно неудобен. Я не думаю, что можно рекомендовать что-либо из подобной практики:

```
#define FALSE  0
if( f() !=  FALSE )
    // ...
```

В действительности здесь проявляется настоящая проблема, связанная с непониманием различий между языком Си и Паскалем. Си, в отличие от Паскаля, не поддерживает встроенный булевый тип, и полагать обратное означает просто навлечь на себя неприятности.

Часто необходимость в явном сравнении на истину или ложь можно устранить при помощи переименования:

```
if( я_сонливый(р) )
```

значительно лучше, чем:

```
if( я_сонливый(р) != FALSE )
```

Так как определения `TRUE` и `FALSE` спрятаны в макросах, то хороший сопровождающий программист не может делать каких-либо предположений об их действительных значениях. Например, `FALSE` может быть `-1`, а `TRUE` — `0`. И следственно, если функция возвращает в явном виде `TRUE` или `FALSE`, то наш прилежный сопровождающий программист должен будет потратить несколько дней, чтобы убедиться, что при проверке возвращаемого значения для каждого вызова используется явная проверка на равенство `TRUE` или `FALSE` (сравните для примера с простым логическим отрицанием `!` перед вызовом). Следующий фрагмент:

```
if( я_сердитый() )
```

более не может удовлетворять, так как компилятор ожидает, что ложь обозначается `0`.

И напоследок — имейте в виду, что следующий вариант не будет работать:

```
#define FALSE   0
#define TRUE    !FALSE
```

Операция `!`, подобно всем операторам отношений, преобразует операнд в `1`, если он имеет значение "истина" (отличен от нуля), и `0`, если наоборот. Предыдущий вариант идентичен следующему:

```
#define FALSE   0
#define TRUE    1
```

Вот более надежный, но нелепый вариант:

```
#define IS_TRUE(x)    ((x) == 0)
#define IS_FALSE(x)   ((x) != 0)
```

86. Для битового поля размером 1 бит должен быть определен тип `unsigned`

После того, как ANSI Си позволил назначать битовому полю знаковый тип, мне доводилось видеть код, подобный:

```
struct fred
{
   int i : 1;
}
a_fred;
```

Возможными значениями являются `0` и `-1`. Оператор типа:

```
#define TRUE    1
// ...
if( a_fred.i == TRUE )
    // ...
```

не будет работать, потому что поле `a_fred.i` может иметь значение `0` или `-1`, но оно никогда не будет равняться `1`. Следовательно, оператор **`if`** никогда не выполняется.

87. Указатели должны указывать на адрес, больший, чем базовый для массива

Это правило подтверждено стандартом ANSI Си, но многие программисты, похоже, не подозревают о том способе, которым язык должен работать. ANSI Си говорит, что указатель может переходить на ячейку, следующую после окончания массива, но он не может иметь величину меньше, чем базовый адрес массива. Нарушение этого правила может прервать программу, которую пытаются выполнить, например, в сегментной модели памяти процессоров 80x86. Следующий код не будет работать:

```
int array[ SIZE ];
int *p = array + SIZE;  // Здесь все в порядке; вы можете
                        // двигаться дальше.
while ( --p >= array )  // Это не работает - возможен
                        // бесконечный цикл.
//...
```

Проблема состоит в том, что при сегментной архитектуре есть возможность того, что массив попадет на начало сегмента и получит исполнительный адрес `0x0000`. (В архитектуре 8086 это будет смещением — частью адреса любого байта, состоящего из адреса сегмента и смещения). Если `p` установлен на начало массива (`0x0000`), то операция `--p` вызывает его перемещение на адрес `0xfffe` (если у типа

int размер 2 байта), который считается большим, чем p. Другими словами, предыдущий цикл никогда не закончится. Исправьте эту ситуацию следующим образом:

```
while ( --p >= array )
{
// ...
if ( p == array )
    break;
}
```

Вы можете выйти из положения так:

```
int *p = array + (SIZE - 1);
do
{
    // ...

} while ( p-- > array );
```

но позаботьтесь, чтобы p был внутри массива перед началом цикла. (Указатель должен быть инициализирован значением p+(SIZE-1), а не p+SIZE).

88. Используйте указатели вместо индексов массива

Вообще, инкрементирование указателя — лучший способ перемещения по массиву, чем индекс массива. Например, простой цикл, подобный следующему, страшно неэффективен:

```
struct thing
{
   int field;
   int another_field;
   int another_field;
};

thing array[ nrows ][ ncols ];
int row, col;

for (  row = 0;  row < nrows ;  ++nrows )
   for (  col = 0;  col < ncols;  ++cols )
      array[row][col].field = 0;
```

Выражение array[row][col] требует двух умножений и одного сложения во время выполнения. Вот что происходит на самом деле:

```
array + (row * size_of_one_row) + (col * size_of_a_thing)
```

Каждая структура имеет размер 12 байтов, и 12 не является степенью 2, поэтому вместо умножения нельзя использовать более эффективный

сдвиг.

Вы можете сделать то же самое посредством указателей следующим образом:

```
thing  *p       = (thing *)array;
int     n_cells = nrows * ncols;
while ( --n_cells  >=  0 )
   (p++)->field = 0;
```

При этом здесь вообще нет умножения во время выполнения. Оператор инкрементирования `p++` просто прибавляет `12` к `p`.

С другой стороны, указатель лучше только тогда, когда вы можете его инкрементировать, то есть когда вы обращаетесь к последовательным элементам. Если вам нужен по настоящему случайный доступ в массив, то запись с квадратными скобками намного проще читается, и разницы в скорости выполнения нет.

Аналогично, если внутренняя часть цикла в принципе неэффективна — скажем, например, мы сделали следующее:

```
for ( row = 0; row < nrows ; ++nrows )
    for ( col = 0; col < ncols ; ++cols )
        f( array[row][col] );
```

и `f()` требует для выполнения две секунды — тогда относительный выигрыш от использования указателей будет существенно перевешен накладными расходами на вызов функции, и, естественно, вы можете утверждать, что квадратные скобки легче читаются. Конечно, если `f()` является встроенной функцией Си++, то накладные расходы на вызов функции могут быть минимальными и есть смысл использовать указатель, поэтому вы можете возразить, что вариант с указателем лучше, ибо накладные расходы тяжело определить.

Наконец, верно, что оптимизатор часто может преобразовать вариант цикла с индексами массива в вариант с указателями, но я думаю, что это плохой стиль — писать неэффективный код в надежде на то, что оптимизатор очистит его после вас. Указатели так же хорошо читаемы, как и индексы массивов, для того, кто знает язык программирования.

89. Избегайте `goto`, за исключением…

Правила в этом разделе применяйте только к программам на Си. Оператор **`goto`** не должен никогда использоваться в Си++ по причинам, рассмотренным в правиле 54 — существует вероятность того, что конструкторы и деструкторы будет невозможно вызвать.

Вообще вы должны избегать оператора **`goto`** не потому, что **`goto`** — унаследованный порок, а потому что существуют лучшие решения. Язык

Си, например, дает вам массу отличающихся от **goto** способов выхода из циклов.

Оператор **goto** может также ухудшать читаемость. Я на самом деле видел код, подобный нижеследующему, и чтобы разобраться, как он работает, потребовалось полчаса:

```
while ( 1 )
{
    while ( условие )
    {
        // ...
        while ( другое_условие )
        {
            метка1:
            // ...
            goto метка2;
        }
        // ...
    }

    if ( третье_условие )
    {
    // ...
        if ( другое_условие )
            goto метка1;
        else
        {
            метка2:
            // ...
        }
    }
}
```

Но самое интересное, что после того, как я разобрался с этим, стало легко переписать его, исключив переходы **goto**.

Проблема читаемости все же сохраняется, даже если **goto** в явном виде отсутствует. Оператор **switch**, например, неявно выполняет **goto** для перехода к оператору **case**. Последующий пример вполне законен с точки зрения Си, но я не стал бы его вам рекомендовать:

```
switch( некоторое_условие )
{
    case A: if ( некоторое_другое_условие )
    // ...
    else
    {
        case b:                  // ...
    }
}
```

Оператор **goto** полезен в некоторых случаях. Вот два из них:

- Множество переходов **goto** к единственной метке, стоящей перед оператором **return**, лучше, чем множество операторов **return**. Такую процедуру легче отлаживать, так как для перехвата выхода из нее вы можете установить единственную точку прерывания. Имейте в виду, что метка должна предшествовать оператору; она не может стоять перед закрывающей фигурной скобкой. При необходимости пользуйтесь следующим приемом:

```
        // ...
    exit:
        return ;
}
```

- Переходы **goto** вниз по программе, обеспечивающие выход из системы вложенных циклов, лучше, чем флаг завершения типа "готов", который должен проверяться в каждом операторе управления циклом. Если каждый из операторов **while** в следующем примере выполнить по 100 раз, то флаг "готов" нужно проверить 1000000 раз, хотя он установлен всего лишь на случай ошибки

```
int  готов = 0;
int  условие1,  условие2,  условие3;
// ...

while ( !готов  &&  условие1 )
{
    while ( !готов  &&  условие2 )
    {
        while ( !готов  &&  условие3 )
        {
        if ( нечто_ужасное )
            готов = 1;
        }
    }
}
```

Исключите миллионы ненужных проверок при помощи **goto** следующим образом:

```
while ( условие1 )
{
    while ( условие2 )
    {
        while ( условие3 )
        {
            if ( нечто_ужасное )
                goto  выход;
        }
    }
}
```

```
выход:
// ...
```

Проверка в операторе управления циклом — единственное место, где эффективность действительно является важным обстоятельством, потому что код выполняется многократно. Это особенно верно для внутренних операторов управления вложенных циклов. Проверка флага завершения во внутреннем цикле может существенно замедлить выполнение, и ее лучше избегать.

Часть

8

Правила программирования на Си++

Эта часть книги содержит правила, уникальные для программирования на Си++. Как мной было сказано во "Введении", эта книга не является учебником по Си++, так что следующие правила предполагают, что вы по крайней мере знакомы с синтаксисом этого языка. Я не буду тратить слова попусту, описывая, как работает Си++. Имеется множество хороших книг, которые познакомят вас с Си++, включая и мою собственную "*C+C++*". Вы должны также ознакомиться с принципами объектно-ориентированного проектирования. Я рекомендую 2-е издание книги Гради Буча "*Object-Oriented Analysis and Design with Applications*" (Redwood City: Benjamin Cummings, 1994).

Так же, как и в книге в целом, правила вначале адресуются к общим вопросам, переходя затем к частностям.

Часть 8а. Вопросы проектирования и реализации

90. Не смешивайте объектно-ориентированное и "структурное" проектирование

90.1. Если проект не ориетирован на объекты, то используйте Си

Позвольте мне начать, сказав, что нет абсолютно ничего дурного в хорошо выполненном структурном проектировании. Как-то так получилось, что я предпочитаю объектно-ориентированный (ОО) подход, ибо мне кажется, что я мыслю ОО способом, но было бы самонадеянным назвать ОО проектирование "лучшим". Я верю, что ОО подход дает вам легче сопровождаемый код, если программа большая. Выгода менее явна в случае программ меньшего размера, потому что объектная ориентация обычно добавляет сложность на уровне приложения. (Главная выгода ОО заключается в лучшем сопровождении за счет абстракции данных, а не в сокращении сложности).

Си++ особенно не выносит небрежного проектирования. Мой опыт говорит, что программы на Си++, которые не придерживаются объектно-ориентированного подхода, почти несопровождаемы, соединяя все худшие свойства структурного и объектно-ориентированного проектов и не давая каких-либо выгод как ни от того, так и ни от другого. Со мной не проходит такой аргумент, что можно использовать Си++ как "улучшенный" Си. Для того, чтобы это было правдой, этот язык слишком сложный — кривая обучения слишком крутая. Если вы не используете преимущества объектно-ориентированных свойств этого языка, то в его использовании мало смысла. Некорректное использование объектно-ориентированных свойств лишь увеличит число проблем.

К сожалению, многие программисты знают, как сделать объектно-ориентированный проект, но на самом деле этого не делают. Оправдания варьируются в пределах от "слишком много хлопот (или у меня нет времени), чтобы делать все правильно" до "строгий объектно-ориентированный проект — это учебное упражнение: на него нет времени в реальной жизни, где вы вынуждены работать быстро и не очень чисто". Возможно, что наиболее возмутительным оправданием, слышанным мной по поводу плохого проекта (в этом случае библиотеки классов), было

следующее: "Недостаточное число наших заказчиков знают Си++ достаточно хорошо, чтобы его правильно использовать, поэтому мы спроектировали библиотеку классов так, чтобы ей было легче пользоваться". (В переводе на нормальный язык: "Средние пользователи слишком тупые, чтобы делать все правильно; на самом деле они даже не заинтересованы в том, чтобы научиться работать правильно, и научить их будет очень трудно. Так что мы даже не будем делать ни того, ни другого. Мы просто оглупим свой продукт"). Проблема была отягощена учебным руководством, которое нарушало объектно-ориентированные принципы налево и направо, и, к сожалению, это руководство используется тысячами программистов, которые не знают ничего лучшего в качестве примера того, как написать приложение при помощи этой библиотеки классов. Они вполне разумно ожидают, что руководство покажет им, как сделать все правильно, поэтому они никогда не подозревают, что все было намеренно сделано неверно, чтобы сделать руководство "более понятным".

Си++ — язык трудный как для изучения, так и для использования. При написании программ на Си++ столько тонкостей, что даже опытные программисты временами их забывают. Кроме того, даже простой поиск достаточного количества программистов на Си++, чтобы писать, и значительно меньшего, чтобы сопровождать ваш код — трудный процесс. Вы вводите себя в заблуждение, если верите, что Си++ может быть использован безыскусно. Слишком просто для неопытного программиста сделать что-нибудь неправильно и даже не знать об этом, способствуя бесполезным затратам времени на выслеживание ошибки, которая и так хорошо видна. Многие ошибки такого типа даже проникают необнаруженными через этап тестирования и попадают в конечный продукт, делая сопровождение сомнительным предприятием.

Зачем же вообще использовать Си++? Ответ состоит в том, что *должным образом использованный* Си++ дает вам существенные выгоды в сопровождении. Вы можете делать значительные изменения в поведении программы (типа перевода всей программы с английского языка на японский или переноса в другую операционную среду) при помощи незначительных изменений в исходном коде, ограниченных малым уголком этого кода. Подобные изменения в структурной системе обычно потребуют модификации поистине каждой функции в программе.

Однако если вы не придерживаетесь правил, то вы в итоге получите в свое распоряжение недостатки обоих систем. Структурные проекты обычно имеют проблемы с отношениями сцепления, которые не встречаются в хорошем объектно-ориентированном проекте, но если вы остановитесь на полдороге, многие из этих ошибок будут скрыты в

классах, где их будет трудно найти. Кроме того, многие объектно-ориентированные проекты обычно бывают очень сложными, а взаимоотношения между объектами иногда непредсказуемы. (Это также одно из главных преимуществ методологии: возможно моделирование системы такой сложности, что ее поведение заранее предсказать невозможно). Инструменты типа диаграмм объектов становятся необходимыми, потому что если эта система не работает, то вероятнее всего причина в потоке сообщений. Если не работает индивидуальный объект, то его легко исправить при условии, что его интерфейс корректен, потому что изменения будут ограничены определением одного класса. Когда вы делаете что-то неверно, то эти проблемы становится очень тяжело выследить, потому что для передачи информации используются тайные ходы, а изменения в одном классе могут передаваться в другие.

Поэтому, если у вас "нет времени, чтобы делать все правильно", то вам гораздо лучше остановиться на структурном проектировании и простой реализации на языке Си. Будет проще искать ошибки, потому что код более однороден, и у вас не будет дополнительных сложностей с системой передачи сообщений, сбивающей с толку. Введение упрощений сегодня может сделать программу "типа объектно-ориентированной" неподдающейся сопровождению год спустя: вам придется выбросить всю программу и начать сначала. Перспектива лучшего сопровождения может реализоваться, лишь если вы следуете правилам.

Так как эта книга не является книгой по ОО-проектированию, то я отсылаю вас к книге Буча, упомянутой во введении к этой главе, если вам нужно познакомиться с процессом объектно-ориентированного проектирования. Эта книга рассматривает правила, которые облегчают протекание этого процесса.

91. Рассчитывайте потратить больше времени на проектирование и меньше на разработку

Мой опыт свидетельствует, что, если исключить период изучения Си++, объектно-ориентированные системы требуют на разработку столько же времени, сколько и структурные системы. Тем не менее, при объектно-ориентированном подходе вы затрачиваете гораздо более высокую долю общего времени на проектирование, и процесс программирования идет быстрее. На практике этап проектирования большой системы может продолжаться от четырех до шести месяцев, прежде чем будет написана первая строка кода. К несчастью, это слишком горькая пилюля для тех, кто измеряет производительность числом строк кода в день, чтобы проглотить ее. Так как общее время разработки остается прежним, то рост

производительности происходит после того, как начинается сопровождение кода. *Корректно выполненные* объектно-проектированные системы проще кодировать и проще сопровождать.

92. Библиотеки классов Си++ обычно не могут быть использованы неискушенными пользователями

Одна большая ложь о Си++, которая распространяется продавцами с острыми зубами и зачесанными назад волосами, сводится к тому, что ваши второсортные программисты на Си могут использовать библиотеки классов, созданные гуру, без реальной необходимости знать Си++. К несчастью, любая, кроме самой тривиальной, библиотека классов будет использовать сложные для понимания разделы Си++ типа наследования и виртуальных функций — по крайней мере, она должна их использовать, если спроектирована как следует. Библиотека, не использующая эти свойства Си++, могла бы запросто быть реализована на Си. Пользователи библиотеки будут должны довольно хорошо знать Си++.

Любая программа, написанная людьми, которые не слишком сведущи в используемом ими языке программирования, будет в лучшем случае иметь много ошибок, в худшем случае она будет несопровождаемой. Вероятно, тяжелее всего найти ту ошибку, про которую вы не думаете, что это ошибка. Если ваше понимание того, как работает этот язык, неполное, то вы можете думать, что все в порядке с фрагментом кода, патологически напичканным ошибками, потому что этот код внешне кажется правильным.

Программная индустрия сталкивалась с этой проблемой и раньше, когда коллективы разработчиков были вынуждены переходить с языка КОБОЛ на Си, но при этом не была обеспечена необходимая тренировка программистов, позволяющая им использовать Си правильно. После этого в качестве урока осталась масса несопровождаемого, переполненного ошибками кода на Си. Си++ показывает все признаки еще более серьезной проблемы, так как руководители часто делают ставку на популярность Си++, в действительности не зная, во что они впутываются. Масса кишащего ошибками кода на Си++ пишется ежедневно людьми, которые даже не знают язык в степени, достаточной, чтобы понять, что они делают что-то неправильно.

93. Пользуйтесь контрольными таблицами

Одной из причин того, что Си++ имеет такую крутую кривую обучения, заключается в том, что вы должны отслеживать большое количество деталей, чтобы выполнить даже простые задачи. Просто забыть что-то, даже если вы это сделаете не надолго. Я решаю эту проблему, применяя повсюду несколько образцовых шаблонных файлов - по одному для каждой распространенной ситуации. (У меня есть один для определения базового класса, один — для определения производного класса, и т.д.). Я начинаю с копирования соответствующего шаблона в свой текущий рабочий файл и затем использую возможности своего редактора по поиску и замене для заполнения пустот. Я также перемещаю подходящие функции в файлы *.cpp*, когда нужно, и т.п.. Листинги 5 и 6 показывают простые шаблонные (в смысле естественного языка, а не языка С++) файлы для базового и производного классов (где кое-что опущено по сравнению с теми, которыми я пользуюсь на самом деле, но идею вы поняли).

Листинг 5. *base.tem* — контрольная таблица для определения базового класса

```
 1  class base
 2  {
 3     cls obj;
 4  public:
 5     virtual
 6        ~base ( void );
 7         base ( void );
 8         base ( const base &r );
 9
10     const base &operator=( const base &r );
11  private:
12  };
13  //-------------------------------------------------------
14  /* виртуальный */ base:: ~base( void )
15  {
16  }
17  //-------------------------------------------------------
18  inline base::base( void ) : obj( value )
19  {
20  }
21  /-------------------------------------------------------
22  inline base::base( const base &r ) : obj( r.obj )
23  {}
24  //-------------------------------------------------------
25  inline const base& base::operator=( const base &r )
26  {
27     if( this != &r )
```

```
28    {
29       obj = r.obj;
30    }
31    return *this;
32 }
```

Листинг 6. *derived.tem* — контрольная таблица для определения производного класса

```
 1  class derived : public base
 2  {
 3     cls obj;
 4  public:
 5     virtual
 6        ~derived ( void );
 7         derived ( void );
 8         derived ( const derived& r );
 9
10     const derived &operator=( const derived &r );
11
12  private:
13  };
14  //-------------------------------------------------------
15  /* виртуальный */ derived:: ~derived( void )
16  {
17  }
18  //-------------------------------------------------------
19  inline derived::derived( void ) : base( value ) ,
20     obj( value )
21  {
22  }
23  //-------------------------------------------------------
24  inline derived::derived( const derived &r ) : base ( r ),
25     obj( r.obj )
26  {}
27  //-------------------------------------------------------
28  inline const derived& derived::operator=( const derived &r )
29  {
30     if( this != &r )
31     {
32         *((base *)this) = r;
33        obj = r.obj;
34     }
35     return *this;
36  }
```

94. Сообщения должны выражать возможности, а не запрашивать информацию

Объектно-ориентированные и структурные системы склонны подходить к проблемам с диаметрально противоположных направлений. Возьмите в качестве примера скромную запись `employee`. В структурных системах вы бы использовали тип **`struct`** и имели бы доступ к полям этого типа повсюду из своей программы. Например, код для печати записи мог бы свободно повторяться в нескольких сотнях мест программы. Если вы меняете что-то в основе, вроде изменения типа поля `name` с массива **`char`** на 16-битные символы Unicode, то вы должны разыскать каждую ссылку на `name` и модифицировать ее для работы с новым типом.

В хорошо спроектированной объектно-ориентированной системе было бы невозможно получить доступ к полю `name`.[4] Позвольте мне повторить это, потому что эта концепция так фундаментальна: невозможно получить доступ к полю внутри объекта, даже такому простому, как `name` в объекте `employee`. Скорее всего вы попросите `employee` проявить какую-нибудь способность, такую как "напечатать себя", "сохранить себя в базе данных" или "модифицировать себя, взаимодействуя с пользователем". В этом последнем случае обработчик сообщений вывел бы диалоговое окно, которое бы использовалось пользователем для ввода или изменения данных.

Главным преимуществом этого подхода является то, что отправитель сообщения может меньше волноваться о том, как организовано внутреннее хранение данных. Пока объект может себя печатать, модифицировать или делать что-нибудь еще — проблемы нет. Вы можете перевести `name` на Unicode, не затрагивая отправителя сообщения. Этот вопрос рассматривается далее во многих правилах этой главы книги.

95. Вам обычно не удастся переделать имеющуюся структурную программу в объектно-ориентированную

Одним из побочных эффектов только что описанной организации является то, что обычно невозможно преобразовать структурный подход в соответствии с этим образом мыслей без полного переписывания кода.

[4] Чтобы быть строго корректным, по крайней мере на языке выражений Си++, я должен называть поле "компонентом данных-членов". Однако довольно неудобно говорить "компонент данных-членов `name`", поэтому буду использовать просто "поле", когда его значение ясно из контекста.

Возвращаясь вновь к печати, отметим, что соответствующим сообщением могло бы быть "воспроизвести себя на этом устройстве", а обработчику сообщения могла быть передана ссылка на обобщенный объект `device`, которому нужно переслать данные. Код, который фактически выполняет воспроизведение, на самом деле находится *внутри* этого объекта. (В порядке разъяснения: нет причины, из-за которой нельзя поддерживать несколько сообщений типа "воспроизведи себя". Например, объект электронной таблицы мог бы поддерживать сообщения "воспроизвести себя в виде таблицы", "воспроизвести себя в виде графика" и "воспроизвести себя в виде круговой диаграммы").

В структурной системе код, который выполняет воспроизведение, является *внешним*. Некая функция получает откуда-то объект, после чего делает различные системные вызовы для вывода его на экран. Если вы говорите о `printf()`, то вызовы не очень сложные, но если речь заходит о Windows или Motif — у вас появляется проблема. Объектно-ориентированный проект фактически является вывернутым наизнанку в сравнении со структурным проектом.

Преимущество объектно-ориентированного подхода — в том, что вы можете менять операционную среду путем изменения реализации объекта `device`, и при этом остальной код в программе не затрагивается. Несмотря на это, вызванные изменения столь фундаментальны, что полный перевод возможен лишь после переработки в программе каждой функции, которая прямо вызывает функцию операционной системы. Это нетривиальное мероприятие, которое вероятно потребует отбрасывания большей части кода в существующем приложении.

Сообщение "модифицировать себя" аналогично: диалоговое окно модификации в стандартной структурной программе изображается внешним кодом. В объектно-ориентированном проекте объект сам взаимодействует с пользователем в ответ на получение сообщения "модифицировать себя" — наизнанку в сравнении со структурным подходом. И снова преимущество в том, что изменения полей, которые должны быть модифицированы, концентрируются в определении класса. Вам не нужно искать по всей программе код, который использует объекты класса, каждый раз, когда меняется поле в определении класса.

Мой опыт с гибридными приложениями не очень удачен: кажется, что в них соединяются все проблемы как структурных, так и объектно-ориентированных систем без каких то преимуществ тех и других. Это реальная опасность для тех, у кого "нет времени, чтобы делать все правильно" — они могут получить в итоге несопровождаемый гибрид.

Ошибочно рассматривать тело существующего кода, вне зависимости от его размера, в качестве "ценного имущества", в которое вы должны постоянно инвестировать. Вы не выбросите деньги, потраченные на написание существующего кода, когда решитесь от него отказаться. Деньги, потраченные на написание кода, уже, наверное, окупились за счет продаж, и теперь, чтобы чего-то достичь, вы должны не дорабатывать существующий код, а писать новый. Начинающая фирма-конкурент, способная лишь куснуть вас за пятку, разрабатывает свой продукт с самого начала и получает преимущество за счет использования современной технологии и идей по методологии проектирования. Между тем ваш существующий код запирает вас в рамках обветшалого проекта и устаревшей технологии. Нельзя просто откинуться на спинку кресла и почить на лаврах — вы должны постоянно переписывать свой продукт заново, чтобы совершенствовать его сколь-нибудь заметным образом.

Я должен сказать, что многие со мной в этом месте не согласны. Рецензент одной из моих недавних статей ответил на (возможно, слишком упрощенное) утверждение, что гибридные приложения "не работают", заявив: "Я знаю массу торгово-транспортных приложений, которые написаны именно таким образом, приносят прибыль своим создателям и, следовательно, работают". С другой стороны, тот факт, что этот рецензент работает в фирме, владеющей несколькими огромными гибридными приложениями, очевидно, повлиял на его отзыв. Один из этих гибридов задерживался с выходом на рынок более чем на год во время, когда писалась рецензия, и сопровождение было постоянным кошмаром для большинства остальных, но я думаю, что эти проблемы были недостаточно важными, чтобы их учитывать, потому что этот конкретный программист не занимался сопровождением.

96. Объект производного класса является объектом базового класса

97. Наследование — это процесс добавления полей данных и методов-членов

В Си++ производный класс может рассматриваться как механизм добавления полей данных и обработчиков сообщений к существующему определению класса — к базовому классу. (Вы можете также смотреть на наследование как на средство изменения поведения объекта базового класса при получении им конкретного сообщения. Я вернусь к такой точке зрения при обсуждении виртуальных функций). В таком случае иерархия классов является просто средством представления полей данных и методов, определяемых для конкретного объекта. Объект содержит все

данные и методы, объявленные на его уровне, а также на всех вышележащих уровнях.

Общая ошибка, совершаемая начинающими программистами на Си++, состоит в том, что они смотрят на иерархию классов и думают, что сообщения передаются от объектов производного класса к объектам базового класса. Помните, что иерархия классов Си++ не существует во время выполнения. Все, что у вас есть во время выполнения, это фактические объекты, чьи поля определяются во время компиляции при помощи иерархии классов.

В этом вопросе путаница создана многими книгами по языку Smalltalk, описывающими реализацию во время выполнения системы обработки сообщений так, как если бы сообщения передавались от производного к базовому классу.[5] Это просто неверно (и в случае Smalltalk, и в случае Си++). Си++ использует наследование. Производный класс — это тот же базовый класс, но с несколькими добавленными полями и обработчиками сообщений. Следовательно, когда объект Си++ получает сообщение, он или обрабатывает его, или нет; он или определяет обработчик, или получает его в наследство. Если ни то, ни другое не имеет места, то сообщение просто не может быть обработано. И оно никуда не передается.

Не путайте отношение наследования с *объединением*. При объединении в один класс (контейнер) вложен объект другого класса (в отличие от наследования от другого класса). Объединение в действительности лучше, чем наследование, если у вас есть возможность выбора, потому что отношения сцепления между вложенным объектом и внешним миром гораздо слабее, чем отношения между базовым классом и внешним миром.

---

[5] Они не передаются. Даже в Smalltalk есть только один объект, который или получает сообщение, или нет. Несмотря на это, интерпретаторы Smalltalk склоняются к реализации обработки сообщений при помощи нескольких таблиц указателей на функции, по одной на каждый класс. Если интерпретатор не может найти обработчик сообщения в таблице диспетчеризации производного класса, то он просматривает таблицу базового класса. Этот механизм не используется в Си++, который является компилируемым языком и поэтому не использует многоуровневый просмотр таблиц в время выполнения. Например, даже если бы все функции в базовом классе были виртуальными, то таблица виртуальных функций производного класса имела бы по ячейке для каждой виртуальной функции базового класса. Среда времени выполнения Си++ не просматривает иерархию таблиц, а просто использует таблицу для текущего объекта. Подробнее об этом позднее.

Объединение позволяет методам контейнера действовать подобно фильтру, через который передаются сообщения, предназначенные для вложенного объекта. Обработчики сообщений часто будут иметь одинаковые имена в контейнере и во вложенном объекте. Например:

```
class string                          // строка
{
// ...
public:
   const string &operator=( const string &r );
};

class numeric_string                  // строка, содержащая число
{
   string str;
   // ...
public:
   const string &operator=( const string &r );
}

const string &numeric_string::operator=( const string &r )
{
   if( r.all_characters_are_digits() )  // все символы - цифры
      str = r;
   else
      throw invalid_assignment();

   return *this;
}
```

Это на самом деле довольно слабый пример объединения, потому что, если бы функция **operator**=() была виртуальной в базовом классе, то объект numeric_string мог бы наследовать от string и заместить оператор присваивания для проверки на верное числовое значение. С другой стороны, если перегруженная операция сложения + в классе string выполняет конкатенацию, то вам может понадобиться перегрузить + в классе numeric_string для выполнения арифметического сложения (т.е. преобразовывать строки в числа, которые складывать и затем присваивать результат строке). Объединение в последнем случае решило бы немного проблем.

Возвращаясь к наследованию, отметим, что объекты классов, показанных в таблице 3, вероятно будут размещаться в памяти одинаково. Каждое из этих определений, как вы заметили, имеет компонент some_cls, но доступ к этому компоненту требует совершенно разных процедур и механизмов. В этой книге я использую выражение "компонент базового класса" по отношению к той части объекта, которая определена на уровне базового класса, а не к вложенному объекту. То есть, когда я

говорю, что объект производного класса имеет "компонент базового класса", то имею в виду, что некоторые из его полей и обработчиков сообщений определены на уровне базового класса. При рассмотрении вложенного объекта я буду называть его "полем" или "вложенным объектом".

Таблица 3. Два определения класса, одинаково представляемые на уровне машинного кода

Объединение	Наследование

`class container`
`{`
`  some_cls contained;`
`  // ...`
`};`
	`class base : public some_cls`
	`{`
	`// ...`
	
	`};`

98. Сначала проектируйте объекты

Первым пунктом повестки дня всегда должно быть проектирование системы обмена сообщениями, обычно посредством диаграмм объектов типа описанных Бучем. Начиная с иерархии классов, вы проявляете склонность к избыточному проектированию, реализуя возможности, которые не нужны. Кроме того, не зная, как нужно связать объекты друг с другом, обычно трудно сказать заранее, какие возможности потребуются в каждом классе. Тяжело обобщать, когда у вас нет деталей.

99. Затем проектируйте иерархию снизу вверх

После того, как вы спроектировали систему объектов и сообщений, вы можете приступать к иерархии. Откиньтесь на спинку кресла и взгляните на различные объекты, и вы увидите, что многие из них получают похожие сообщения. Если два сообщения, посылаемые к разным объектам, похожи, но не одинаковы, то вам может подойти слегка более общее компромиссное, которое сможет работать и в том, и в другом месте. Обработчики для всех общих сообщений должны быть сконцентрированы в единый базовый класс. Например, имея один объект, получающий сообщения A, B и C, и второй объект, получающий A, B, D и E, вы должны остановиться на маленькой иерархии классов, в которой базовый класс реализует обработчики сообщений для A и B, один производный класс реализует обработчик для C, а второй производный класс — обработчики для D и E. Вы продолжаете этот процесс соединения общих элементов в общие базовые классы до тех пор, пока нечего будет соединять. Теперь у вас есть иерархия базовых классов.

Вы заметите, что, чем более общим является класс, тем выше он

расположен в иерархии. Например, класс управляющего `manager` вероятно имеет все свойства класса обобщенного служащего `employee`, а также несколько дополнительных свойств (таких, как список подчиненных служащих). Тогда для `manager` имеет смысл наследовать от `employee`, потому что это добавит возможности, отсутствующие в базовом классе `employee`.

На этом этапе процесса проектирования вы все еще даже не подумали о том, что же должно быть внутри объектов. Вы по-прежнему имеете дело только с системой обмена сообщениями.

Последним шагом на этом этапе проектирования — после того, как вы сделали эскиз проекта иерархии классов — остается запись определений классов. Вы добавите открытые (**`public`**) функции-члены для каждого сообщения, получаемого объектом. *Эти обработчики сообщений являются единственными открытыми членами вашего определения класса*. Все остальное *должно быть* закрытым или защищенным. Подробнее об этом далее.

99.1. Базовые классы должны иметь более одного производного объекта

Это просто другая точка зрения на предыдущее правило. Если базовый класс является способом концентрации сходных свойств в одном месте, то есть смысл в том, чтобы никогда не иметь всего один производный класс. Если он у вас один, то возможности этого единственного потомка должны быть переданы родителю.

100. Возможности, определенные в базовом классе, должны использоваться *всеми* производными классами

101. Си++ — это не Smalltalk: избегайте общего класса `object`

Процесс разработки иерархии снизу вверх обычно дает вам лес из маленьких деревьев, скорее широких, чем высоких. Построение иерархии снизу вверх поможет вам избежать общей проблемы для иерархий классов Си++: класса `object`, от которого наследуется все в системе, как в Smalltalk. Такой проект хорош для Smalltalk, но, как правило, не работает в Си++. Какое свойство мог бы реализовывать этот общий `object`? То есть, какое свойство должен иметь каждый объект каждого класса в вашей программе? Единственное, что приходит на ум, это — управление памятью, способность объекта себя создать. Это делается в

Си++ посредством оператора **new**, который в действительности является функцией глобального уровня. Фактически вы можете смотреть на глобальный уровень Си++, как на функциональный эквивалент `object` в Smalltalk. Хорошая иерархия классов Си++ представляет собой обычно коллекцию иерархий меньшего размера. Процитируем такого авторитета, как самого Бьярна Страуструпа — создателя Си++ — по этому поводу[6] :

> *Смысл здесь заключается в том, что те стили, которые подходят и хорошо поддерживаются в Smalltalk, не обязательно подходят для Си++. В частности, рабское следование стилю Smalltalk в Си++ ведет к неэффективным, уродливым и с трудом сопровождаемым программам на Си++. Причина в том, что хороший Си++ требует проекта, который извлекает преимущества из системы статических типов Си++, а не борется с ней. Smalltalk поддерживает систему динамических типов (и только) и эта точка зрения, переведенная на Си++, ведет к чрезвычайно ненадежному и уродливому приведению типов.*
>
> *...Вдобавок, Smalltalk поощряет людей смотреть на наследование, как на единственный, или, по меньшей мере, основной метод организации программ, и организовывать классы в иерархии с единственной вершиной. В Си++ классы являются типами, и наследование ни в коем случае не является единственным средством организации программ. В частности, шаблоны являются основным средством представления контейнерных классов.*

Одной из больших проблем плохо организованных иерархий является превышение багажной нормы. Базовые классы должны иметь поля для поддержки возможностей, реализуемых различными обработчиками. Если производный класс не использует такую возможность, тогда его объект распространяет всюду связанные с ним накладные расходы, не давая выгод. Это одна из проблем иерархии в стиле Smalltalk, имеющей одну вершину в виде общего объекта. Все поля, помещенные в него вами (и все ячейки в таблице виртуальных функций), будут переняты каждым объектом в системе, независимо от того, использует объект эти поля, или нет.

Лучшим способом избежать этой проблемы является использование множественного наследования для реализации классов-смешений. Вот как

---

[6] Эта цитата является отрывком из статьи, размещенной Страуструпом в телеконференции BIX в декабре 1992 г. Полностью статья опубликована в книге Мартина Хеллера "*Advanced Win32 Programming*"(New York: Wiley,1993), pp.72-78.

смешение работает. Возвратившись к нашему примеру с `employee`, вы могли бы реализовать его в виде системы классов следующим образом:

```
class employee
{
// содержит всю информацию, общую для всех служащих:
// фамилия, адрес и т.д.
};

class manager : public employee
{
// добавляет информацию, специфичную для управляющего,
// такую, как список подчиненных служащих. Управляющий тоже
// является служащим, поэтому применимо наследование

   database list_of_managed_emploees;
}

class peon : public employee
{
// добавляет информацию, специфичную для поденщика

   manager *this_boss;
}
```

Все это приемлемо до тех пор, пока не приходит время создавать наш список объектов `employee`, который поддерживается объектом `manager`. Во многих реализациях структур данных объект делается сохраняемым путем наследования его класса от класса, который реализует то, что нужно структуре данных для работы по назначению. Вы могли бы сделать это здесь так:

```
class storable;                    // сохраняемый

class employee : public storable { /* ... */ };
class manager  : public employee { /* ... */ };
class peon     : public employee { /* ... */ };
```

Например, метод `add()` класса `database` мог бы получать указатель на объект `storable` в качестве своего аргумента. Таким способом любой объект `storable` (или объект, производный от `storable`) может быть добавлен в `database` без необходимости модифицировать что-либо в программе, в состав которой входит класс `database`.

Все кажется правильным до тех пор, пока мы реально не взглянем на то, как используются классы. Давайте скажем, что это средняя фирма, где число управляющих относится к числу поденщиков как 100 к 1. Однако списка управляющих нет, есть лишь список поденщиков. Тем не менее, каждый `manager` обладает излишней сохраняемостью, которая никогда не используется. Решим эту проблему при помощи множественного

наследования.

```
class storable;

class employee                                        { /* ... */ };
class manager : public employee                       { /* ... */ };
class peon    : public employee, public storable      { /* ... */ };
```

Проблема здесь в том, что эта "сохраняемость" является атрибутом объекта. Это не является базовым классом в стандартном смысле типа "круг *является* фигурой", а скорее — "поденщик *является* сохраняемым". Здесь важна замена существительного на прилагательное. Базовый класс, который реализует "свойство" типа сохраняемости, называется классом-*смешением*, потому что вы можете примешивать это свойство к тем классам, которым оно нужно, и только к этим классам. Хороший метод распознавания этих двух употреблений наследования состоит в том, что имя класса-смешения обычно выражено прилагательным (сохраняемый, сортируемый, устойчивый, динамический и т.д.). Именем настоящего базового класса обычно является существительное.

Вследствие природы Си++ во всех учебниках рассматривается несколько проблем с множественным наследованием, большинство из которых вызывается ромбовидной иерархией классов:

```
class parent {};                                      // родитель

class mother : public parent {};                      // мать
class father : public parent {};                      // отец

class child  : public mother, public father {}  // потомок
```

Здесь имеется две трудности. Если у `parent` есть метод для укладывания спать с названием `go_to_sleep()`, то вы получите ошибку, попытавшись послать такое сообщение:

```
child philip;                               // Филипп - потомок

philip.go_to_sleep();                       // Филипп, иди спать!
```

Проблема состоит в том, что в объекте `child` на самом деле два объекта `parent`. Запомните, что наследование просто добавляет поля (данные-члены) и обработчики сообщений (функции-члены). Объект `mother` имеет компонент `parent`: он содержит дополнительно к своим собственным все поля `parent`.[7] То же самое относится и к `father`.

[7] Не путайте этот процесс с объединением. У `mother` нет поля `parent`, скорее та часть `mother`, которая определена на уровне базового класса, изображается как "компонент `parent`".

Затем, у `child` есть `mother` и `father`, у каждого из которых есть `parent`. Проблема с `philip.go_to_sleep()` состоит в том, что компилятор не знает, какой из объектов parent должен получить это сообщение: тот, который в `mother`, или тот, который в `father`.[8]

Одним из путей решения этой проблемы является введение уточняющей функции, которая направляет сообщение нужному классу (или обоим):

```
class parent { public: go_to_sleep(); };

class mother : public parent {};
class father : public parent {};

class child : public mother, public father
{
public:
   go_to_sleep()
   {
      mother::go_to_sleep();
      father::go_to_sleep();
   }
}
```

Другим решением является виртуальный базовый класс:

```
class parent {};

class mother : virtual public parent {};
class father : virtual public parent {};

class child  : public mother, public father {}
```

который заставляет компилятор помещать в объект `child` лишь один объект `parent`, совместно используемый объектами `mother` и `father`. Двусмысленность исчезает, но появляются другие проблемы. Во-первых, нет возможности показать на уровне потомка, хотите вы или нет виртуальный базовый класс. Например, в следующем коде `tree_list_node` может быть членом как дерева, так и списка одновременно:

```
class node;
```

[8] На самом деле правильнее сказать, что во время компиляции компилятор не знает, от какого из базовых классов `parent` объект `child` наследует обработчик сообщения `go_to_sleep()`, хотя эта правильность и может сбить с толку. Вы можете спросить, почему неопределенность имеет значение, ведь эта функция одна и та же в обоих классах. Компилятор не может создать ветвление времени выполнения, так как не знает, какое значение присвоить указателю **`this`**, когда он вызывает функцию-член базового класса.

```
class list_node : public node {};
class tree_node : public node {};

class tree_list_node : public list_node, public tree_node {};
```

В следующем варианте `tree_list_node` может быть членом или дерева, или списка, но не обоих одновременно:

```
class node;
class list_node : virtual public node {};
class tree_node : virtual public node {};

class tree_list_node : public list_node, public tree_node {};
```

Вам бы хотелось делать этот выбор при создании `tree_list_node`, но такой возможности нет.

Второй проблемой является инициализация. Конструкторы в `list_node` и `tree_node`, вероятно, инициализируют базовый класс `node`, но разными значениями. Если имеется всего один `node`, то какой из конструкторов выполнит эту инициализацию? Ответ неприятный. Инициализировать `node` должен наследуемый последним производный класс (`tree_list_node`). Хотя это действительно плохая мысль — требовать, чтобы класс знал о чем-либо в иерархии, кроме своих непосредственных родителей — иначе было бы слишком сильное внутреннее связывание.

Обратная сторона той же самой проблемы проявляется, если у вас есть виртуальные функции как в следующем коде:

```
class persistent
{
public:
   virtual flush() = 0;
};

class doc1: virtual public persistent
{
public:
   virtual flush() { /* сохранить данные doc1 на диске */ }
};

class doc2: virtual public persistent
{
public:
virtual flush() { /* сохранить данные doc2 на диске */ }
};

class superdoc : public doc1, public doc2 {};

persistent *p = new superdoc();
p->flush();      // ОШИБКА: какая из функций flush() вызвана?
```

102. Смешения не должны наследоваться от чего попало

103. Смешения должны быть виртуальными базовыми классами

104. Инициализируйте виртуальные базовые классы при помощи конструктора, используемого по умолчанию

Вы можете свести до минимума рассмотренные ранее проблемы, стараясь придерживаться следующих правил (многие смешения не могут соответствовать им всем, но вы делайте все от вас зависящее):

- Если можно, то смешения не должны наследоваться от чего попало, тем самым полностью устраняя проблему ромбовидной иерархии при множественном наследовании.
- Для смешения должна обеспечиваться возможность быть виртуальным базовым классом для того, чтобы не возникала проблема неопределенности в случае, если у вас все же получилась ромбовидная структура классов.
- Если можно, то смешение должно всегда строиться с использованием только конструктора по умолчанию (не имеющего аргументов). Это упрощает оформление смешения в качестве виртуального базового класса, потому что вам не нужно будет заботиться об инициализации большей части наследуемого объекта. В конце концов, по умолчанию всегда используется конструктор по умолчанию.

105. Наследование не подходит, если вы никогда не посылаете сообщения базового класса объекту производного класса

106. Везде, где можно, предпочитайте включение наследованию

107. Используйте закрытые базовые классы лишь когда вы должны обеспечить виртуальные замещения

Главная выгода от наследования состоит в том, что вы можете писать универсальный код, манипулирующий объектами обобщенного базового класса, и тот же самый код может также манипулировать объектами производного класса (или точнее, может манипулировать компонентом базового класса в объекте производного класса). Например, вы можете написать функцию, которая печатает список объектов фигура, но этот список на самом деле содержит объекты, которые унаследованы от фигуры, такие как круг и линия. Тем не менее, функции печати этого знать не нужно. Она вполне довольна, считая их обобщенными фигурами. Это качество является тем, что имеют в виду, когда говорят о повторном использовании кода. Вы повторно используете один и тот же код для разных дел: временами он печатает круг, временами — линию.

Если вы обнаружили у себя объект производного класса, от которого никогда не требуется использовать возможности базового класса, то, вероятно, в проектировании иерархии есть какая-то ошибка, хотя встречаются редкие случаи, когда такое поведение приемлемо; поэтому в языке есть закрытые базовые классы. Но все же включение (назначение объекта полем в классе, а не базовым классом) всегда лучше, чем наследование (при условии, конечно, что у вас есть выбор).

Если объект производного класса никогда не получает сообщения базового класса, то вероятнее всего компонент базового класса в объекте производного класса действительно должен быть полем, и наследование вовсе не должно использоваться. Вместо вот этого:

```
class derived : public base
{
};
```

вам почти всегда лучше делать так:

```
class derived
{
   base base_obj;
```

```
};
```

Используйте закрытые базовые классы лишь в случаях, когда вам нужно в производном классе перегружать виртуальные функции базового класса.

Удачный пример подобного неправильного использования наследования есть во многих иерархиях классов для Windows, которые наследуют классы типа "диалоговое окно" от "окна". Однако в реальной программе вы никогда не посылаете относящиеся к окну сообщения (типа "сдвинуться" или "изменить размер") в диалоговое окно. То есть диалоговое окно *не является* окном, по крайней мере, с точки зрения того, как диалоговое окно используется в программе. Скорее диалоговое окно использует окно, чтобы себя показать. Слово "*является*" подразумевает наследование, а "*использует*" — включение, которое здесь лучше подходит.

Подобное плохое проектирование, между прочим, обычно имеет причиной отступление от правила определения объектов в первую очередь. То есть концепция "окна" в Microsoft Windows имеет смысл только для подсистемы визуального вывода. Диалоговое окно изображается в виде окна, но это *не значит*, что это окно, даже если подсистема визуального вывода предпочитает его рассматривать в этом качестве. Плохое проектирование получается, когда исходят из существующей системы визуального вывода и затем помещают вокруг нее оболочку при помощи библиотеки классов, вместо того, чтобы исходить из описания программы, решая затем, как реализовать в программе реальные объекты.

108. Проектируйте структуры данных в последнюю очередь

Добавление полей данных выполняется в процессе проектирования в последнюю очередь. Другими словами, после того, как вы разработали сообщения, вам нужно понять, как реализовать возможности, запрашиваемые этими сообщениями. Вероятно, это труднейшая часть процесса объектно-ориентированного проектирования для структурного программиста: заставить себя не думать о лежащей в основе структуре данных до тех пор, пока не будет готовы полностью система обмена сообщениями и иерархия классов.

В этот момент процесса проектирования вы также добавляете закрытые (**`private`**) "рабочие" (или "вспомогательные") функции, которые помогают обработчикам сообщений справиться со своей работой.

109. Все данные в определении класса должны быть закрытыми

110. Никогда не допускайте открытого доступа к закрытым данным

Все данные в определении класса должны быть закрытыми. Точка. Никаких исключений. Проблема здесь заключается в тесном сцеплении между классом и его пользователями, если они имеют прямой доступ к полям данных. Я приведу вам несколько примеров. Скажем, у вас есть класс `string`, который использует массив типа **`char`** для хранения своих данных. Спустя год к вам обращается заказчик из Пакистана, поэтому вам нужно перевести все свои строки на урду, что вынуждает перейти на Unicode. Если ваш строковый класс позволяет какой-либо доступ к локальному буферу **`char`***, или сделав это поле открытым (**`public`**), или определив функцию, возвращающую **`char`***, то вы в большой беде.

Взглянем на код. Вот *действительно* плохой проект:

```
class string
{
public:
   char *buf;
   // ...
};

f()
{
   string s;
   // ...
printf("%s/n", s.buf );
}
```

Если вы попробуете изменить определение `buf` на `wchar_t*` для работы с Unicode (что предписывается ANSI Си), то все функции, которые имели прямой доступ к полю `buf`, перестают работать. И вы будете должны их все переписывать.

Другие родственные проблемы проявляются во внутренней согласованности. Если строковый объект содержит поле `length`, то вы могли бы модифицировать буфер без модификации `length`, тем самым разрушив эту строку. Аналогично, деструктор строки мог бы предположить, что, так как конструктор разместил этот буфер посредством **`new`**, то будет безопаснее передать указатель на buf оператору **`delete`**. Однако если у вас прямой доступ, то вы могли бы сделать что-нибудь типа:

```
string s;
char  array[128];
s.buf = array;
```

и организация памяти разрушается, когда эта строка покидает область действия.

Простое закрытие при помощи модификатора **private** поля `buf` не помогает, если вы продолжаете обеспечивать доступ посредством функции. Листинг 7 показывает фрагмент простого определения строки, которое будет использоваться мной несколько раз в оставшейся части этой главы. (Упрощение, сделанное мной, свелось к помещению всего в один листинг; обычно определение класса и встроенные функции будут в заголовочном файле, а остальной код — в файле *.cpp*).

Листинг 7. Простой строковый класс

```
1  class string
2  {
3     char *buf;
4     int   length;       // длина буфера (не строки);
5
6  public:
7     virtual
8       ~string( void );
9        string( const char *input_str = "" );
10       string( const string &r );
11
12    virtual const string &operator=( const string &r );
13
14    virtual int operator< ( const string &r ) const;
15    virtual int operator> ( const string &r ) const;
16    virtual int operator==( const string &r ) const;
17
18    virtual void print( ostream &output ) const;
19    // ...
20 };
21 //-----------------------------------------------------------------
22 inline string::string( const char *input_str /*= ""*/ )
23 {
24    length = strlen(input_str) + 1;
25    buf = new char[ length ];
26    strcpy( buf, input_str );
27 }
28 //-----------------------------------------------------------------
29 inline string::string( const string &r )
30 {
31    length = r.length;
32    buf = new char[ length ];
33    strcpy( buf, r.buf );
34 }
35 //-----------------------------------------------------------------
36 /* виртуальный */ string:: ~string( void )
37 {
```

```
38    delete buf;
39 }
40 //---------------------------------------------------------------------
41 /* виртуальный */ const string &string::operator=( const string &r)
42 {
43    if( this != &r )
44    {
45       if( length != r.length )
46       {
47          free( buf );
48          length = r.length;
49          buf = new char[ length ];
50       }
51       strcpy( buf, r.buf );
52    }
53    return *this;
54 }
55
56 //---------------------------------------------------------------------
57 /* виртуальный */ int string::operator< ( const string &r ) const
58 {
59    return strcmp(buf, r.buf) < 0;
60 }
61 //---------------------------------------------------------------------
62 /* виртуальный */ int string::operator> ( const string &r ) const
63 {
64    return strcmp(buf, r.buf) > 0;
65 }
66 //---------------------------------------------------------------------
67 /* виртуальный */ int string::operator==( const string &r ) const
68 {
69    return strcmp(buf, r.buf) == 0;
70 }
71 //---------------------------------------------------------------------
72 /* виртуальный */ void string::print( ostream &output ) const
73 {
74    cout << buf;
75 }
76 //---------------------------------------------------------------------
77 inline ostream &operator<<( ostream &output, const string &s )
78 {
79 // Эта функция не является функцией-членом класса string,
80 // но не должна быть дружественной, потому что мной тут
81 // реализован метод вывода строкой своего значения.
82
83    s.print(output);
84    return output;
85 }
```

Вы заметите, что я умышленно не реализовал следующую функцию в листинге 7:

```
string::operator const char*() { return buf; }
```

Если бы реализовал, то мог бы сделать следующее:

```
void f( void )
{
   string s;
   // ...
   printf("%s\n", (const char*)s );
}
```

но я не смогу реализовать функцию **operator char***(), которая бы работала со строкой Unicode, использующей для символа 16-бит. Я должен бы был написать функцию **operator** wchar_t*(), тем самым модифицировав код в функции f():

```
printf("%s/n", (const wchar_t*)s );
```

Тем не менее, одним из главных случаев, которых я стараюсь избежать при помощи объектно-ориентированного подхода, является необходимость модификации пользователя объекта при изменении внутреннего определения этого объекта, поэтому преобразование в **char*** неприемлемо.

Также есть проблемы со стороны внутренней согласованности. Имея указатель на buf, возвращенный функцией **operator const char***(), вы все же можете модифицировать строку при помощи указателя и испортить поле length, хотя для этого вам придется немного постараться:

```
string s;
// ...
char *p = (char *)(const char *)s;
gets( p );
```

В равной степени серьезная, но труднее обнаруживаемая проблема возникает в следующем коде:

```
const char *g( void )
{
 string s;
// ...
return (const char *)s;
}
```

Операция приведения вызывает функцию **operator const char***(), возвращающую buf. Тем не менее, деструктор класса string передает этот буфер оператору **delete**, когда строка покидает область действия. Следовательно, функция g() возвращает указатель на освобожденную память. В отличие от предыдущего примера, при этой второй проблеме нет закрученного оператора приведения в два этапа, намекающего нам, что что-то не так.

Реализация в листинге 7 исправляет это, заменив преобразование

char* на обработчиков сообщений типа метода самовывода (`print()`). Я бы вывел строку при помощи:

```
string s;
s.print( cout )
```

или:

```
cout << s;
```

а не используя `printf()`. При этом совсем нет открытого доступа к внутреннему буферу. Функции окружения могут меньше беспокоиться о том, как хранятся символы, до тех пор, пока строковый объект правильно отвечает на сообщение о самовыводе. Вы можете менять свойства представления строки как хотите, не влияя на отправителя сообщения `print()`. Например, строковый объект мог бы содержать два буфера — один для строк Unicode и другой для строк **char*** — и обеспечивать перевод одной строки в другую. Вы могли бы даже добавить для перевода на французский язык сообщение `translate_to_French()` и получить многоязыкую строку. Такая степень изоляции и является целью объектно-ориентированного программирования, но вы ее не добьетесь, если не будете непреклонно следовать этим правилам. Здесь нет места ковбоям от программирования.

110.1. Не пользуйтесь функциями типа `get/set` (чтения и присваивания значений)

Это правило в действительности то же, что и предыдущее "все данные должны быть закрытыми". Я выделил его, потому что есть такая распространенная ошибка среди начинающих программистов на Си++. Нет разницы между:

```
struct xxx
{
   int x;
};
```

и:

```
class xxx {
private:
   int x;

public:
   void setx ( int ix ){ x = ix;       }
   int  getx ( void )  { return x; }
}
```

за исключением той, что второй вариант труднее читать. Просто сделать

данные закрытыми недостаточно: вам нужно изменить образ мыслей. Подведем итог по нескольким упомянутым ранее пунктам:

- Сообщение реализует свойство. Открытая (**public**) функция реализует обработчик сообщения. Поля данных — лишние во внешнем мире; вы добавляете их лишь для того, чтобы иметь возможность реализовать свойство. Доступ к ним должен быть невозможен.

Заметьте, что вы будете изредка видеть обработчик сообщений, который ничего не делает, кроме возврата содержимого поля или помещает в поле значение, переданное в виде аргумента. Этот обработчик тем не менее не является функцией типа `get/set`. Вопрос в том, как возникает такая ситуация. Нет абсолютно ничего плохого в том, если вы начинаете с ряда сообщений и затем решаете, что самым простым способом реализации сообщения является помещение специального поля в определение класса. Другими словами, этот обработчик сообщений не является усложненным способом доступа к полю; скорее, это поле является простым способом реализовать сообщение. Хотя вы попали в то же место, вы попали туда совершенно другим путем.

Конечно, эта организация означает, что Си++ не может быть эффективно использован в гибридной среде Си/Си++, потому что интерфейс между двумя половинами программы уничтожает инкапсуляцию, которой вы так сильно старались добиться. В известном смысле жаль, что Си++ создан на основе Си, потому что это просто подстрекает нас к ошибкам.

Закончу этот раздел более реальным примером. Как-то раз я видел интерфейс, в котором объект "`календарь`" позволял пользователю интерактивно выбирать дату, щелкая мышью на каком-либо из дней, показанных на изображении календаря. "`Календарь`" затем экспортирует эту дату в другие части программы, помещая ее в объект "`дата`", который возвращается из сообщения `get_date()`. Проблема здесь в том, что проектирование выполнено выполнено наизнанку. Программист мыслил структурными категориями, а не объектно-ориентированными.

При выполнении должным образом единственным видимым в других частях программы объектом был бы объект "`дата`". "`Дата`" использовала бы объект "`календарь`" для реализации сообщения "`инициализируй_себя`" (которое могло бы быть конструктором), но "`календарь`" бы содержался внутри "`даты`". Определение класса "`календарь`" можно было бы даже вложить в определение класса "`дата`". Объект "дата" также мог бы поддерживать другие инициализирующие сообщения, такие как

"инициализируй_себя_от_редактируемого_ввода" или "инициализируй_себя_из_строки", но во всех случаях объект "дата" отвечает за нужное для инициализации взаимодействие с пользовательским интерфейсом. Остальная часть программы просто бы непосредственно использовала "дату"; никто, кроме "даты", даже бы не знал о существовании объекта "календарь". То есть вы бы объявили "дату" и приказали ей себя инициализировать. Затем вы можете передавать объект "дата" всюду, куда необходимо. Конечно, "дата" должна также уметь себя вывести, переслать в файл или из файла, сравнить себя с другими датами и так далее.

111. Откажитесь от выражений языка Си, когда программируете на Си++

Многие из проблем, рассмотренных в предыдущих правилах, вызваны программистами на Си, не желающими отказаться от знакомых выражений Си при переходе на Си++. Та же самая проблема существует и в естественных языках: вам будет тяжело заставить себя понять по-французски, если вы просто переведете английские выражения в их буквальные эквиваленты.

Хорошим примером этой проблемы в Си++ является **char***. Большинство программистов на Си ни за что не соглашаются отказаться от использования строк в виде **char***. Проблема заключается в том, что вы привыкли смотреть на **char*** и думать, что это строка. Это не строка. Это указатель. Убежденность в том, что указатель — это строка, обычно вызывает проблемы, некоторые из которых я уже рассматривал, а другие будут рассмотрены позднее.

Симптомами этой проблемы является появление **char*** где-нибудь в программе, которая поддерживает класс string; вы должны делать все на языке string. Обобщим это: чтобы заставить объектно-ориентированную систему работать, *все должно быть* объектами. Основные типы Си не очень применимы, за исключением глубоких недр низкоуровневых функций-членов класса низкого уровня. Инкапсуляция вашего **char*** в классе **string** решит множество проблем, и потратите массу времени, пытаясь поддерживать **char***, при том, что существует вполне хороший класс string, который может делать ту же работу.

Определение класса не обязательно увеличивает накладные расходы, поэтому это не может быть оправданием. Если ваш класс `string` имеет единственное поле **`char`***, и если все из методов являются встроенными функциями, то ваши накладные расходы не превысят те, которые бы у вас были при прямом использовании **`char`***, но зато вы получите все выгоды сопровождения, предоставляемые классами Си++. Более того, у вас будет возможность наследовать от `string`, что невозможно с **`char`***.

Возьмем в качестве примера управляющий элемент-редактор Windows — маленькое окно, в котором пользователь вводит данные. (Программисты для X-Window, для вас "управляющий элемент" Windows — это примерный эквивалент `widget`). Управляющий элемент-редактор имеет все свойства как окна, так и строки, и, следовательно, вам было бы желательно его реализовать, наследуя одновременно от класса `window` и от класса `string`.

112. Проектируйте с учетом наследования

Никогда не надейтесь, что класс не будет использоваться в качестве базового класса. Сосредоточимся на случае с примером управляющего элемента-редактора из предыдущего правила. Я бы хотел реализовать такой элемент, наследуя одновременно от класса `window` и от класса `string`, потому что он обладает свойствами обоих. У меня ничего бы не получилось, если бы многие из функций `string` не были виртуальными. То есть, так как я могу делать со строкой следующее:

```
string str = "xxx"; // инициализировать строку значением "xxx"
str = "Абв";        // заменить предыдущее значение на "Абв"
str += "где";       // присоединяет "где" к имеющейся строке.
```

то хотел иметь возможность делать следующее, чтобы поместить текст как в буфер, принадлежащий управляющему элементу-редактору, так и в соответствующее окно:

```
class edit_control : public string
   , public window
{/* ... */}

edit_control edit = "xxx";
edit = "Абв";
edit += "где";
```

Я бы также хотел передавать свой объект `edit_control` в функцию, ожидающую в качестве аргумента `string`, так чтобы любые изменения, которые эта функция делает в (том, что она принимает за) `string`, автоматически отображались и в окне управляющего элемента-редактора.

Все это не возможно, если функции, подобные **operator**=() и **operator**+=(), не виртуальные в классе string и, тем самым, не позволяющие мне менять их поведение в производном классе edit_control. Например, так как функция **operator**=() класса string из листинга 7 со страницы 155 является виртуальной, то я могу сделать следующее:

```
class edit_control : public string
   , public window
{
// ...
   virtual string &operator=( const string &r );
}

virtual string &edit_control::operator=( const string &r )
{
   *(string *)this = r;
   window::caption() = r;  // операция разрешения видимости
                           // window:: просто для ясности
}
```

Следующей функции может быть передан или простой объект string, или объект edit_control; она не знает или ей все равно, какой конкретно:

```
f( string *s )
{
   // ...
   *s = "Новое значение" ;
}
```

В случае объекта string внутренний буфер обновляется. В случае edit_control буфер обновляется, но также модифицируется заголовок его окна.

112.1. Функция-член должна обычно использовать закрытые поля данных класса

Так как все открытые функции-члены являются обработчиками сообщений, а все закрытые функции и поля данных просто поддерживают открытых обработчиков сообщений, то где-то есть ошибка, если функция не имеет доступа к полям данных или не может вызвать функцию, имеющую к ним доступ. Эта функция должна, вероятно, быть передвинута на глобальный уровень или в другой класс.

Ясным признаком того, что вы сделали что-то неправильно, является функция из одного класса, требующая для своей работы доступа к полям объекта другого класса (в отличие от того, чтобы иметь указатель на

другой объект для передачи этому объекту сообщения). В самом худшем случае класс "хозяин" дает статус дружественного классу "гость", и функция-член класса "гость" использует указатель "хозяина" для доступа к его полям, но не может получить никакого доступа к любому из полей своего собственного класса. Механизм дружественности часто неверно используется таким способом, но класс должен давать статус друга только так, чтобы друг мог посылать закрытые сообщения классу, дарящему дружбу. Дружественный класс никогда не должен иметь доступ к данным другого класса; это сцепление слишком сильное.

Вы часто видите эту ошибку в архитектурах "документ/отображение" типа MacApp и MFC. С точки зрения архитектуры, "документ" содержит данные, а "отображение" реализует пользовательский интерфейс. Трудности возникают, когда вы хотите показать какие-нибудь данные в своем "отображении". Никогда не позволяйте "отображению" доступ к полям "документа" для их показа. Данные любого класса, включая "документ", должны быть тщательно охраняемым секретом. Лучшим подходом является передача "отображением" в "документ" сообщения "отобразить себя в этом окне".[9]

113. Используйте константы

В программы на Си класс памяти **`const`** часто не включается. На самом деле это просто небрежность, но она мало влияет на возможности программы на Си. Так как Си++ гораздо разборчивее в отношении типов, чем Си, то в Си++ это гораздо более крупная проблема. Вы должны использовать модификатор **`const`** везде, где можно; это делает код более надежным, и часто компилятор не принимает код, который его не использует. Особенно важно:

- Всегда передавать указатели на константные объекты, если вы не модифицируете эти объекты. Объявление:

  ```
  puts( const char *p )
  ```

  сообщает компилятору, что функция `puts()` не намерена модифицировать символы в массиве, переданном при помощи `p`. Это является чрезвычайно полезной порцией информации для сопровождения.

[9] Пользователи MFC могут обратиться за более глубоким обсуждением этого вопроса к моей статье "Rewriting the MFC Scribble Program Using an Object-Oriented Design Approach" в августовском номере журнала "*Microsoft Systems Journal*" за 1995 г.

- Все сообщения, не меняющие внутреннее состояние объекта, объявлять с модификатором **const** подобным образом:

```
class cls
{
   public: int operator==( const cls &p ) const ;
};
```

(Это тот модификатор **const** справа, относительно которого я тут распинаюсь). Этот **const** говорит компилятору, что передача сообщения объекту, объявленному константным, безопасна. Заметьте, что этот самый правый модификатор **const** в действительности создает следующее определение для указателя **this**:

```
const current_class *this;
```

Если код в этой константной функции попытается модифицировать любое поле данных класса или предпримет вызов другой функции-члена, не помеченной **const**, то вы получите сообщение об ошибке компиляции такого примерно содержания "не могу преобразовать указатель на **const** current_class в указатель на current_class". Упомянутым указателем в данном случае является **this**, и никогда не будет дозволено преобразование указателя на константу в указатель на переменную (потому что вы тогда могли бы модифицировать константу при помощи указателя).

Константные ссылки тоже важны и рассматриваются позже.

114. Используйте структуры только тогда, когда все данные открытые и нет функций-членов

Это правило является вариантом принципа "если это похоже на Си, то должно и действовать как Си". Используйте структуры, только если вы делаете что-то в стиле Си.

Следует также избегать наследования от структуры. Даже если мне многое не удалось изложить четко, надеюсь, что я прояснил смысл тезиса "закрытые данные или никакие". Зная о проблемах с прямым доступом к открытым данным, вы можете понять, почему следующее не является очень хорошей идеей:

```
typedef struct tagSIZE     // Существующее определение из
                           // заголовочного файла Си
{
   LONG  cx;
   LONG  cy;
}
SIZE;
```

```
class CSize : public SIZE // Определение в файле Си++
{
   // ...
}
```

Я видел определения классов, подобные следующему, где требуется доступ к полям `cx` и `cy` базового класса через указатель производного класса для того, чтобы определить соответствующее им значение третьей координаты — высоты. Например:

```
CSize some_size;
   some_size.cy;                  // тьфу!
```

Вы должны иметь возможность написать:

```
some_size.height();
```

У предшествующего кода есть другая, более трудно уловимая проблема. Наследование от существующей структуры Си часто выполняется программистом, который верит, что сможет передать объект Си++ в существующую функцию Си. То есть программист полагает, что раз наследование *фактически* добавляет поля к базовому классу, то производный класс в *буквальном смысле* будет расположен точно так же, как и базовый класс, но с присоединением нескольких дополнительных полей. Однако, это может быть и не так. Если производный класс добавляет, например, виртуальную функцию, то в базовый класс может быть добавлен указатель на таблицу виртуальных функций. Аналогично, если производный класс использует множественное наследование одновременно от структуры Си и чего-то еще, то нет никакой гарантии, что структура Си будет в начале.

115. Не размещайте тела функций в определениях классов

Здесь есть несколько проблем. Если вы действительно поместите тело функции в определение класса таким образом:

```
class amanda
{
public:
   void peekaboo( void ){ cout << "ку-ку\n"; } // функция игры
                                               // в прятки с
                                               // Амандой
}
```

Си++ делает этот класс встроенным. Первая проблема заключается в том, что такие функции с течением времени имеют тенденцию разрастаться и становятся слишком большими, чтобы быть встроенными. Поэтому

лучше помещать определения своих встроенных функций вне определения класса, но в том же заголовочном файле, где размещается определение класса:

```
class amanda
{
public:
   void peekaboo( void );
}

class amanda::peekaboo( void )
{
   cout << "ку-ку\n";
}
```

Путаница — более крупная проблема, чем размер. Часто определение класса является единственной имеющейся у вас определенной документацией по членам класса. Вам на самом деле нужно, чтобы все поместилось на одной странице, и чтобы это определение давало краткий список прототипов функций. Если имена функции и аргумента выбраны точно, то это часто вся документация, которая вам необходима.

Как только вы начинаете добавлять тела функций, даже если они состоят из одной строки, к определению класса - вы эту ясность теряете. Определение класса начинает распространяться на несколько страниц, и становится трудно найти что-нибудь, используя определение класса в качестве средства документирования.

Третья проблема более коварна и потребует нескольких часов на устранение, если вы не будете аккуратны. Рассмотрим фрагмент реализации связанного списка на листинге 8 (который не будет компилироваться). Классы `linked_list` и `list_node` посылают сообщения друг другу. Компилятор должен увидеть определение класса до того, как он позволит вам послать сообщение объекту этого класса. (Вы можете объявить указатель на объект, лишь глядя на **`class`** `xxx`; но вы не можете ничего сделать при помощи этого указателя до завершения определения всего класса). Так как в листинге 8 используются встроенные функции, то невозможно устроить эти определения классов так, чтобы избежать предварительных ссылок. Вы можете решить эту проблему, поместив определения функций в конце того файла, где они объявлены. Я сделал это в листинге 9.

Листинг 8. Фрагмент реализации связанного списка

```
 1  class list_node;
 2
 3  class linked_list
 4  {
 5     int number_of_elements_in_list;
 6     list_node *root;
 7
 8  private:     // этот раздел содержит сообщения, получаемые
 9     friend class list_node; // только от объектов list_node
10     void have_removed_an_element(void)
11     {
12        --number_of_elements_in_list;
13     }
14
15 public:
16     void remove_this_node( list_node *p )
17     {
18     // Следующая строка генерирует ошибку при компиляции,
19     // так как компилятор не знает, что list_node
20     // имеет сообщение remove_yourself_from_me( &root ).
21
22        p->remove_yourself_from_me( &root );
23     }
24
25 // ...
26 };
27
28 class list_node
29 {
30     linked_list *owner;
31 private:                        // Этот раздел содержит
32     friend class linked_list; // сообщения,получаемые только
33                                 // от объектов linked_list
34     void remove_yourself_from_me( list_node *root )
35     {
36     // ... Выполнить удаление
37        owner->have_removed_an_element();
38     }
39 };
```

Листинг 9. Улучшенный вариант реализации связанного списка

```
 1  class list_node;
 2
 3  class linked_list
 4  {
 5     int number_of_elements_in_list;
 6     list_node *root;
 7
 8  private:
```

```
 9     friend class list_node;
10     void have_removed_an_element( void );
11
12 public:
13     void remove_this_node( list_node *p );
14
15 //...
16 };
17 //=========================================================
18 class list_node
19 {
20     linked_list *owner;
21 private:                  // Этот раздел содержит сообщения,
22     friend class linked_list; // получаемые только от
23                               // объектов linked_list
24
25     void remove_yourself_from_me( list_node *root );
26 };
27
28 //=========================================================
29 // функции класса linked_list:
30 //=========================================================
31 inline void linked_list::remove_this_node( list_node *p )
32 {
33     p->remove_yourself_from_me( &root );
34 }
35 //---------------------------------------------------------
36 inline void linked_list::have_removed_an_element( void )
37 {
38     --number_of_elements_in_list;
39 }
40
41 //=========================================================
42 // функции класса list_node:
43 //=========================================================
44 void list_node::remove_yourself_from_me( list_node *root )
45 {
46 // ... Выполнить удаление
47     owner->have_removed_an_element();
48 }
```

116. Избегайте перегрузки функций и аргументов, используемых по умолчанию

Это правило не применяется к конструкторам и функциям перегрузки операций.

Перегрузка функций, подобно многим другим свойствам Си++, была добавлена к этому языку по особым причинам. Не позволяйте себя увлечь этим. Функции, которые делают разные вещи, должны иметь и разные имена.

Перегруженные функции обычно вызывают больше проблем, чем их решают. Во-первых, проблема двусмысленности:

```
f( int, long );
f( long, int );

f( 10, 10 );          // ОШИБКА: Какую из функций я вызываю?
```

Более коварно следующее:

```
f( int );
f( void* );

f( 0 );               // ОШИБКА: Вызов двусмысленный
```

Проблемой здесь является Си++, который считает, что `0` может быть как указателем, так и типом **`int`**. Если вы делаете так:

```
const void *NULL = 0;
const int   ZERO = 0;
```

то вы можете записать `f(NULL)` для выбора варианта с указателем и `f(ZERO)` для доступа к целочисленному варианту, но это ведет к большой путанице. В такой ситуации вам бы лучше просто использовать функции с двумя разными именами.

Аргументы по умолчанию, создающие на самом деле перегруженные функции (по одной на каждую возможную комбинацию аргументов), также вызывают проблемы. Например, если вы написали:

```
f( int x = 0 );
```

и затем случайно вызвали `f()` без аргументов, компилятор успешно и без возражений вставит `0`. Все, чего вы добились, — это устранили то, что в ином случае вызвало бы полезное сообщение об ошибке во время компиляции, и сдвинули ошибку на этап выполнения.

Исключениями из сказанного выше являются перегруженные операции и конструкторы; многие классы имеют их по нескольку, и аргументы по умолчанию часто имеют смысл в конструкторах. Код, подобный

следующему, вполне приемлем:

```
class string
{
public:
   string( char *s = ""     );
   string( const string &r  );
   string( const CString &r ); // преобразование из класса MFC.
   // ...
};
```

Для пояснения: разные классы будут часто обрабатывать одно и то же сообщение, реализуя функции-обработчики с совпадающими именами. Например, большинство классов реализуют сообщение `print()`. Смысл того, что я пытаюсь здесь добиться, такой: плохая мысль - в одном классе иметь много обработчиков сообщений с одним и тем же именем. Вместо:

```
class string
{
 // ...
public:
   print( FILE     *fp  );
   print( iostream &ios );
   print( window   &win );
```

я бы рекомендовал:

```
class string
{
// ...
public:
   print_file   ( FILE     *fp  );
   print_stream ( iostream &ios );
   print_window ( window   &win );
```

Еще лучше, если бы у вас был класс устройства `device`, который бы мог представлять типы: файловый `FILE`, потоковый `iostream` и оконный `window`, в зависимости от того, как он инициализируется — тогда бы вы могли реализовать единственную функцию `print()`, принимающую в качестве аргумента `device`.

Я должен сказать, что сам порой нарушаю это правило, но делаю это, зная, что, переступив черту, могу навлечь на себя беду.

Часть 8б. Проблемы сцепления

Концепция сцепления описана ранее в общем виде. Я также указал наиболее важное правило Си++ для сокращения числа отношений сцепления: "Все данные должны быть закрытыми". Идея минимизации связей на самом деле центральная для Си++. Вы можете возразить, что главной целью объектно-ориентированного проектирования является минимизация отношений связи посредством инкапсуляции. Этот раздел содержит специфические для Си++ правила, касающиеся связывания.

117. Избегайте дружественных классов

Сцепление происходит лишь до определенной степени. Тесное сцепление между классами происходит, когда вы используете ключевое слово **`friend`**. В этом случае, когда вы что-либо меняете в классе, который предоставляет дружественный статус, то должны также проверить каждую функцию в дружественном классе, чтобы убедиться, что она еще работает.

Эта особенность явно не нужна; на самом деле вы хотите ограничить доступ, обеспечиваемый дружественным механизмом. Мне бы понравилось что-нибудь, работающее подобно защищенной части, но по отношению к друзьям. В связанном списке, например, я бы хотел разрешить объекту `list_node` посылать множество сообщений объекту `list`, но я не хочу, чтобы эти сообщения были реализованы посредством открытых функций, потому что никто не будет их посылать, кроме объектов `list_node`. Мой `list` может сделать эти функции закрытыми и предоставить статус дружественного объекту `list_node`, но `list_node` тогда сможет получить доступ к каждому закрытому члену `list`. На самом деле я хочу следующего: "Функции-члены этого дружественного класса могут вызвать вот эти три закрытые функции-члена, но не могут получить доступ к чему-либо еще закрытому". К сожалению, язык Си++ не располагает методом ограничения доступа к заданному подмножеству обработчиков сообщений: доступно все или ничего.

Хотя мы не можем изменить это поведение, но, по крайней мере, мы можем ограничить ущерб путем соглашения. Другими словами, мы можем предоставлять статус друга с подразумеваемым пониманием того, что дружественный объект будет обращаться лишь к ограниченному числу функций в предоставляющем дружбу классе. Отразим это

документально следующим образом:

```
class granting
{
   // ...

private: friend class grantee

   // Функции, определенные в этом разделе, будут доступны
   // членам класса grantee, но не доступны для открытого
   // использования извне.

   message_sent _from_grantee();
   another_message_sent_from_grantee();

private:

   // Настоящие закрытые функции располагаются здесь. Хотя
   // grantee мог бы получить доступ к этим функциям, но не
   // получает.

   // ...
};
```

Помните, что мы на самом деле не ограничиваем дружбы; это просто соглашение о записи, чтобы помочь читателю нашего определения класса угадать наше намерение. Надеемся, что кто-бы ни писал класс `grantee`, он будет достаточно взрослым, чтобы не обмануть нашего дружелюбия нежелательными улучшениями.

118. Наследование — это форма сцепления

Наследование — не панацея, потому что оно является, прежде всего, формой сцепления. Когда вы изменяете базовый класс, то изменение затрагивает все объекты производного класса и всех пользователей объектов производных классов (которые могут передавать им сообщения, обработчики которых унаследованы от базового класса). Вообще, вы должны делать свою иерархию классов как можно менее глубокой для ослабления этого вредного эффекта. К тому же, защищенный класс памяти является подозрительным, так как тут имеется более тесное сцепление между базовыми и производными классами, чем должно быть при использовании производным классом только открытого интерфейса с базовым классом.

119. Не портьте область глобальных имен: проблемы Си++

Определение класса обеспечивает отличный способ вывода идентификатора из области глобальных имен, потому что эти идентификаторы должны быть доступны или через объект, или посредством явного имени класса. Функция `x.f()` отличается от `y.f()`, если `x` и `y` являются объектами разных классов. Аналогично, `x::f()` отличается от `y::f()`. Вы должны смотреть на имя класса и `::` как эффективную часть имени функции, которая может быть опущена лишь тогда, когда что-нибудь еще (типа `.` или `->`) служит для уточнения.

Я часто использую перечислитель для ограничения видимости идентификатора константы областью видимости класса:

```
class tree
{
   enum { max_nodes = 128 };

public:
   enum traversal_mechanism { inorder, preorder, postorder };

   print( traversal_mechanism how = inorder );
   // ...
}

// ...
f()
{
   tree t;
   // ...
   t.print( tree::postorder );
}
```

Константа `tree::postorder`, переданная в функцию `print()`, определенно не в глобальной области имен, потому что для доступа к ней требуется префикс `tree::`. При этом не возникает конфликта имен, так как если другой класс имеет член с именем `postorder`, то он вне класса будет именоваться `other_class::postorder`. Более того, константа `max_nodes` является закрытой, поэтому к ней можно получить доступ лишь посредством функций-членов и друзей класса `tree`, что обеспечивает дальнейшее ограничение видимости.

Преимущество перечислителя над членом-константой класса состоит в том, что его значение может быть инициализировано прямо в объявлении класса. Член-константа должен инициализироваться в функции-конструкторе, который может быть в другом файле. Перечислитель может быть также использован в качестве размера в объявлении массива и в

качестве значения **case** в операторе **switch**; константа ни в одном из этих мест работать не будет♣.

Константа-член имеет свое предназначение. Во-первых, вы можете помещать в нее значения с типом, отличным от **int**. Во-вторых, вы можете инициализировать ее во время выполнения. Рассмотрим следующее определение глобальной переменной в Си++:

```
const int default_size = get_default_size_from_ini_file();
```

Ее значение считывается из файла во время загрузки программы, и оно не может быть изменено во время выполнения.

Вышеупомянутое также применимо к константам-членам класса, которые могут быть инициализированы через аргумент конструктора, но не могут меняться функциями-членами. Так как объект типа **const** не может стоять слева от знака равенства, константы-члены должны инициализироваться посредством списка инициализации членов следующим образом:

```
class fixed_size_window
{
   const size height;
   const size width;

   fixed_size_window( size the_height, size the_width )
                                  : height( the_height )
                                  , width ( the_width )
   {}
}
```

Вложенные классы также полезны. Вам часто будет нужно создать "вспомогательный" класс, о котором ваш пользователь даже не будет знать. Например, текст программы из Листинга 10 реализует класс `int_array` — динамический двухмерный массив, размер которого может быть неизвестен до времени выполнения. Вы можете получить доступ к его элементам, используя стандартный для Си/Си++ синтаксис массива (`a[row][col]`). Класс `int_array` делает это, используя вспомогательный класс, о котором пользователь `int_array` ничего не знает. Я использовал вложенное определение для удаления определения этого вспомогательного класса из области видимости глобальных имен. Вот как это работает: Выражение `a[row][col]` оценивается как `(a[row])[col]`. `a[row]` вызывает `int_array::operator[]()`, который возвращает объект `int_array::row`, ссылающийся на целую строку. `[col]` применяется к этому объекту `int_array::row`, приводя

♣ Утверждение автора не соответствует стандарту языка. — *Ред.*

к вызову `int_array::row::operator[]()`. Эта вторая версия `operator[]()` возвращает ссылку на индивидуальную ячейку. Заметьте, что конструктор класса `int_array::row` является закрытым, потому что я не хочу, чтобы любой пользователь имел возможность создать строку `row`. Строка должна предоставить дружественный статус массиву `int_array` с тем, чтобы `int_array` мог ее создать.

Листинг 10. Вспомогательные классы

```
#include <iostream.h>

class int_array
{
   class row
   {
      friend class int_array;
      int *first_cell_in_row;

      row( int *p ) : first_cell_in_row(p) {}
   public:
      int &operator[] ( int index );
   };

   int nrows;
   int ncols;
   int *the_array;

public:
   virtual
      ~int_array( void           );
       int_array( int rows, int cols );

       row operator[] (int index);
};
//=========================================================
// функции-члены класса int_array
//=========================================================
int_array::int_array( int rows, int cols )
                        : nrows ( rows )
                        , ncols ( cols )
                        , the_array ( new int[rows * cols])
{}
//---------------------------------------------------------
int_array::~int_array( void )
{
   delete [] the_array;
}
//---------------------------------------------------------
inline int_array::row int_array::operator[]( int index )
{
   return row( the_array + (ncols * index) );
```

```
 43 }
 44 //=========================================================
 45 // функции-члены класса int_array::row
 46 //=========================================================
 47 inline int &int_array::row::operator[]( int index )
 48 {
 49    return first_cell_in_row[ index ];
 50 }
 51
 52 //=========================================================
 53 void main ( void )       // ..♣
 54 {
 55    int_array ar(10,20); // то же самое, что и ar[10][20], но
 55                         // размерность во время компиляции
 56    ar[1][2] = 100;      // может быть не определена.
 57    cout << ar[1][2];
 59 }
```

♣ В соответствии со стандартом должно быть `int main ( void )`. — *Ред.*

Часть 8в. Ссылки

120. Ссылочные аргументы всегда должны быть константами

121. Никогда не используйте ссылки в качестве результатов, пользуйтесь указателями

Использование ссылочных аргументов в языке программирования вызвано четырьмя причинами:

- Они нужны вам для определения конструктора копии.
- Они нужны вам для определения перегруженных операций. Если вы определили:

  ```
  some_class *operator+( some_class *left, some_class *right
  );
  ```

  то вы должны сделать такое дополнение:

  ```
  some_class x, y;
  x = *(&x + &y)
  ```

  Использование ссылок для аргумента и возвращаемого значения позволяет вам написать:

  ```
  x = x + 1;
  ```

- Вы часто хотите передать объекты по значению, исходя из логики. Например, вы обычно в функцию передаете тип **double**, а не указатель на **double**. Тем не менее, тип **double** представляет собой 8-байтовую упакованную структуру с тремя полями: знаковым битом, мантиссой и порядком. Передавайте в этой ситуации ссылку на константный объект.
- Если объект какого-нибудь определенного пользователем класса обычно передается по значению, то используйте вместо этого ссылку на константный объект, чтобы избежать неявного вызова конструктора копии.

Ссылки в языке не предназначены для имитации Паскаля и не должны использоваться так, как используются в программе на Паскале.

Проблема ссылочных аргументов — сопровождение. В прошлом году один из наших сотрудников написал следующую подпрограмму:

```
void copy_word( char *target, char *&src ) // src является
                                           // ссылкой на char*
{
   while( isspace(*src) )
      ++src;                    // Инкрементировать указатель,
                                // на который ссылается src.
   while( *src && !isspace(*src) )
   *target++ = *src++;          // Передвинуть указатель,
                                // на который ссылается src,
                                // за  текущее слово.
}
```

Автор полагал, что вы будете вызывать `copy_word()` многократно. Каждый раз подпрограмма копировала бы следующее слово в буфер `target` и продвигала бы указатель в источнике.

Вчера вы написали следующий код:

```
f( const char *p )
{
   char *p = new char[1024];
   load( p );

   char word[64];
   copy_word( word, p );
   delete( p ); // Сюрприз! p был модифицирован, поэтому весь
}               // этот участок памяти обращается в кучу
мусора!
```

Главная проблема состоит в том, что, глядя на вызов `copy_word( word, p )`, вы не получаете подсказки о возможном изменении p в подпрограмме. Чтобы добраться до этой информации, вы должны взглянуть на прототип этой функции (который, вероятно, скрыт на 6-ом уровне вложенности в заголовочном файле). Огромные проблемы при сопровождении.

Если что-то похоже на обычный вызов функции Си, то оно должно и действовать как вызов обычной функции Си. Если бы автор `copy_word()` использовал указатель для второго аргумента, то вызов выглядел бы подобным образом:

```
copy_word( word, &p );
```

Этот дополнительный знак `&` является решающим. Средний сопровождающий программист полагает, что единственная причина передачи адреса локальной переменной в другую функцию состоит в том, чтобы разрешить функции модифицировать эту локальную переменную. Другими словами, вариант с указателем является самодокументирующимся; вы сообщаете своему читателю, что этот объект изменяется функцией. Ссылочный аргумент не дает вам такой

информации.

Это не значит, что вы должны избегать ссылок. Четвертая причина в начале этого раздела вполне законна: ссылки являются замечательным способом избегать ненужных затрат на копирование, неявных при передаче по значению. Тем не менее, для обеспечения безопасности ссылочные аргументы должны всегда ссылаться на константные объекты. Для данного прототипа:

```
f( const some_class &obj );
```

этот код вполне законен:

```
some_class an_object;
f( an_object );
```

Он похож на вызов по значению и при этом, что более важно, действует подобно вызову по значению — модификатор **const** предотвращает модификацию `an_object` в функции `f()`. Вы получили эффективность вызова по ссылке без его проблем.

Подведем итог: Я решаю, нужно или нет использовать ссылку, вначале игнорируя факт существования ссылок. Входные аргументы функций передаются по значению, а выходные — используют указатели на то место, где будут храниться результаты. Я затем преобразую те аргументы, которые передаются по значению, в ссылки на константные объекты, если эти аргументы:

- являются объектами какого-то класса (в отличие от основных типов, подобных **int**);
- не модифицируются где-то внутри функции.

Объекты, которые передаются по значению и затем модифицируются внутри функции, конечно должны по-прежнему передаваться по значению.

В заключение этого обсуждения рассмотрим пример из реальной жизни того, как *не надо* использовать ссылки. Объект `CDocument` содержит список объектов `CView`. Вы можете получить доступ к элементам этого списка следующим образом:

```
CDocument  *doc;
CView      *view;

POSITION pos = doc->GetFirstViewPosition();
while(  view = GetNextView(pos) )
   view->Invalidate();
```

Здесь есть две проблемы. Во-первых, у функции `GetNextView()` неудачное имя. Она должна быть названа

`GetCurrentViewAndAdvancePosition()`, потому что она на самом деле возвращает текущий элемент и затем продвигает указатель положения (который является ссылочным аргументом результата) на следующий элемент. Что приводит нас ко второй проблеме: средний читатель смотрит на предыдущий код и задумывается над тем, как завершается этот цикл. Другими словами, здесь скрывается сюрприз. Операция итерации цикла скрыта в `GetNextView(pos)`, поэтому неясно, где она происходит. Ситуация могла быть хуже, если бы цикл был больше и содержал бы несколько функций, использующих `pos` в качестве аргумента — вы бы не имели никакого представления о том, какая из них вызывает перемещение.

Есть множество лучших способов решения этой проблемы. Простейший заключается в использовании в качестве аргумента `GetNextView()` указателя вместо ссылки:

```
POSITION pos = doc->GetFirstViewPosition();
while( p = GetNextView( &pos ) )
   p->Invalidate();
```

Таким способом `&pos` сообщает вам, что `pos` будет модифицироваться; иначе зачем передавать указатель? Тем не менее, существуют и лучшие решения. Вот первое:

```
for( CView *p = doc->GetFirstView(); p ; p = p->NextView() )
   p->Invalidate();
```

Вот второе:

```
POSITION pos = doc->GetFirstViewPosition();
for( ; pos ; pos = doc->GetNextView(pos) )
   (pos->current())->Invalidate();
```

Вот третье:

```
CPosition pos = doc->GetFirstViewPosition();
for( ; pos; pos.Advance() )
   ( pos->CurrentView() )->Invalidate();
```

Вот четвертое:

```
ViewListIterator cur_view = doc->View_list(); // Просмотреть
                                              // весь список
                                              // отображений
                                              // этого
                                              // документа.
for( ; cur_view ; ++cur_view ) // ++ переходит к следующему
                               // отображению.
cur_view->Invalidate();    // -> возвращает указатель View*.
```

Вероятно, есть еще дюжина других возможностей. Все предыдущее варианты обладают требуемым свойством — в них нет скрытых операций

и ясно, как происходит переход к "текущему положению".

122. Не возвращайте ссылки (или указатели) на локальные переменные

Эта проблема проявляется и в Си, где вы не можете вернуть указатель на локальную переменную. Не возвращайте ссылку на объект, который не существует после этого возврата. Следующий код не работает:

```
some_class &f()
{
   some_class x;
   // ...
   return x;
}
```

Действительной проблемой здесь является синтаксис Си++. Оператор **return** может располагаться на отдалении от определения возвращаемой величины. Единственный способ узнать, что на самом деле делает **return** x, — это взглянуть на заголовок функции и посмотреть, возвращает она ссылку, или нет.

123. Не возвращайте ссылки на память, выделенную оператором new

Каждый вызов **new** должен сопровождаться **delete** — подобно malloc() и free(). Я иногда видел людей, старающихся избежать накладных расходов от конструкторов копии перегруженной бинарной операции подобным образом:

```
const some_class &some_class::operator+( const some_class &r )
                                                          const
{
   some_class *p = new some_class;
   // ...
   return *p;
}
```

Этот код не работает, потому что вы не можете вернуться к этой памяти, чтобы освободить ее. Когда вы пишите:

```
some_class a, b, c;

c = a + b;
```

то a + b возвращает объект, а не указатель. Единственным способом получить указатель, который вы можете передать в оператор **delete**, является:

```
some_class *p;
c = *(p = &(a + b));
```

Это даже страшно выговорить. Функция **operator**+() не может прямо возвратить указатель. Если она выглядит подобным образом:

```
const some_class *some_class::operator+( const some_class &r )
                                                          const
{
   some_class *p = new some_class;
   // ...
   return p;
}
```

то вы должны записать:

```
c = *(p = a + b);
```

что не так страшно, как в предыдущем примере, но все еще довольно плохо. Единственное решение этой задачи состоит в том, чтобы стиснуть зубы и вернуть объект:

```
const some_class some_class::operator+( const some_class &r )
                                                          const
{
   some_class obj;
   // ...
   return obj;
}
```

Если вам удастся вызвать конструктор копии в операторе **return**, то быть по сему.

Часть 8г. Конструкторы, деструкторы и operator=( )

Функции конструкторов, деструкторов и операций **operator**=() имеют ту особенность, что их создает компилятор в том случае, если не создаете вы. Генерируемый компилятором конструктор по умолчанию (не имеющий аргументов) и генерируемый компилятором деструктор нужны для создания указателя на таблицу виртуальных функций (подробнее об этом вскоре).

Генерируемый компилятором конструктор копии (чьим аргументом является ссылка на текущий класс) нужен еще по двум причинам, кроме таблицы виртуальных функций. Во-первых, код на Си++, который выглядит как Си, должен и работать как Си. Так как правила копирования, которые относятся к классу, относятся также и к структуре, поэтому компилятор будет вынужден обычно генерировать конструктор копии в структуре, чтобы обрабатывать копирование структур в стиле Си. Этот конструктор копии используется явно подобным образом:

```
some_class x;              // конструктор по умолчанию
some_class y = x;       // конструктор копии
```

но кроме этого он используется и неявно в двух ситуациях. Первой является вызов по значению:

```
some_class x;
f( some_class x ) // передается по значению, а не по ссылке.

f( x );        // вызывается конструктор копии для передачи x
               // по значению. Оно должно скопироваться в стек.
```

Второй является возврат по значению:

```
some_class g() // Помните, что x - локальная, автоматическая
               // переменная.Она исчезает после возвращения
               // функцией значения.
{
   some_class x; // Оператор return после этого должен
   return x;     // скопировать x куда-нибудь в надежное место
}              // (обычно в стек после аргументов).Он
               // использует для этой цели конструктор копии.
```

Генерируемая компилятором функция-операция **operator**=() нужна лишь для поддержки копирования структур в стиле Си там, где не определена операция присваивания.

124. Операция operator=( ) должна возвращать ссылку на константу

125. Присваивание самому себе должно работать

Определение **operator**=( ) должно всегда иметь следующую форму:

```
class class_name
{
   const class_name &operator=( const class_name &r );
};

const class_name &class_name::operator=( const class_name &r )
{
   if( this != &r )
   {
      // здесь скопировать
   }
   return *this;
}
```

Аргумент, представляющий операнд источника данных, является ссылкой, чтобы избежать накладных расходов вызова по значению; это ссылка на константу, потому что аргумент не предназначен для модификации.

Эта функция возвращает ссылку, потому что она может это сделать. То есть вы могли бы удалить & из объявления возвращаемой величины, и все бы работало прекрасно, но вы бы получили ненужный вызов конструктора копии, вынужденный возвратом по значению. Так как у нас уже есть объект, инициализированный по типу правой части (***this**), то мы просто можем его вернуть. Даже если возврат объекта вместо ссылки в действительности является ошибкой для функции **operator**=(), компилятор просто выполнит то, что вы ему приказали. Здесь не будет сообщения об ошибке; и на самом деле все будет работать. Код просто будет выполняться более медленно, чем нужно.

Наконец, **operator**=() должен возвращать ссылку на константу просто потому, что не хотите, чтобы кто-нибудь имел возможность модифицировать возвращенный объект после того, как произошло присваивание. Следующее будет недопустимым в случае возврата ссылки на константу:

```
(x =y) = z;
```

Причина состоит в том, что (x=y) расценивается как возвращаемое значение функции **operator**=(), т.е. константная ссылка. Получателем сообщения =z является объект, только что возвращенный от x=y. Тем не

менее, вы не можете послать сообщение **operator**=() константному объекту, потому что его объявление не имеет в конце **const**:

```
                                    // НЕ ДЕЛАЙТЕ ЭТОГО В ФУНКЦИИ
                                    // С ИСПОЛЬЗОВАНИЕМ operator=().
                                    //                  |
                                    //                  V
const class_name &operator=( const class_name &r ) const;
```

Компилятор должен выдать вам ошибку типа "не могу преобразовать ссылку на переменную в ссылку на константу", если вы попробуете (x=y)=z.

Другим спорным моментом в предыдущем коде является сравнение:

```
if( this != &r )
```

в функции **operator**=(). Выражение:

```
class_name x;
// ...
x = x;
```

должно всегда срабатывать, и сравнение **this** с адресом входного правого аргумента является простым способом в этом убедиться. Имейте в виду, что многие алгоритмы полагают самоприсваивание безвредным, поэтому не делайте его особым случаем. Также имейте в виду, что самоприсваивание могло бы быть затушевано при помощи указателя, как в:

```
class_name array[10];
class_name *p = array;
// ...
*p = array[0];
```

126. Классы, имеющие члены-указатели, должны всегда определять конструктор копии и функцию operator=()

Если класс не определяет методы копирования — конструктор копии и функцию **operator**=(), то это делает компилятор. Созданный компилятором конструктор должен выполнять "почленное" копирование, которое осуществляется таким образом, как будто вы написали **this**->field = src.field для каждого члена. Это означает, что теоретически должны вызываться конструкторы копий и функции **operator**=() вложенных объектов и базовых классов. Даже если все работает правильно, все же указатели копируются как указатели. То есть, строка string, представленная как **char***, — не строка, а указатель, и будет скопирован лишь указатель. Представьте, что определение string

на листинге 7 со страницы 155 не имеет конструктора копии или функции **operator**=(). Если вы запишите

```
string s1 = "фу", s2;
// ...
s2 = s1;
```

то это присваивание вместо поля указателя s2 запишет указатель от s1. Та память, которая была адресована посредством s1->buf, теперь потеряна, то есть у вас утечка памяти. Хуже того, если вы меняете s1, то s2 меняется также, потому что они указывают на один и тот же буфер. Наконец, когда строки выходят из области действия, они обе передают buf для освобождения, по сути, очищая его область памяти дважды, и, вероятно, разрушают структуру динамической памяти. Решайте эту проблему путем добавления конструктора копии и функции **operator**=(), как было сделано на листинге 7 со страницы 155. Теперь копия будет иметь свой собственный буфер с тем же содержанием, что и у буфера строки-источника.

Последнее замечание: я выше написал "должен выполнять" и "теоретически" в первом абзаце, потому что встречал компиляторы, которые фактически выполняли функцию memcpy() в качестве операции копирования по умолчанию, просто как это бы сделал компилятор Си. В этом случае конструктор копии и функция **operator**=() вложенных объектов не будут вызваны, и вы *всегда* будете должны обеспечивать конструктор копии и функцию **operator**=() для копирования вложенных объектов. Если вы желаете достигнуть при этом абсолютной надежности, вы будете должны проделать это для всех классов, чьи члены не являются основными числовыми типами Си++.

127. Если у вас есть доступ к объекту, то он должен быть инициализирован

128. Используйте списки инициализации членов

129. Исходите из того, что члены и базовые классы инициализируются в случайном порядке

Многие неопытные программисты на Си++ избегают списков инициализации членов, как я полагаю, потому, что они выглядят так причудливо. Фактом является то, что большинство программ, которые их не используют, попросту некорректны. Возьмите, например, следующий код (определение строкового класса из листинга 7 со страницы 155):

```
class base
```

```
{
   string s;
public:
   base( const  char *init_value );
}
//------------------------------
base::base( const  char *init_value )
{
   s = init_value;
}
```

Основной принцип такой: если у вас есть доступ к объекту, то он должен быть инициализирован. Так как поле `s` видимо для конструктора `base`, то Си++ гарантирует, что оно инициализировано до окончания выполнения тела конструктора. Список инициализации членов является механизмом выбора выполняемого конструктора. Если вы его опускаете, то получите конструктор по умолчанию, у которого нет аргументов, или, как в случае рассматриваемого нами класса `string`, такой, аргументы которого получают значения по умолчанию. Следовательно, компилятор вначале проинициализирует `s` пустой строкой, разместив односимвольную строку при помощи **`new`** и поместив в нее `\0`. Затем выполняется тело конструктора и вызывается функция `string::operator=()`. Эта функция освобождает только что размещенный буфер, размещает буфер большей длины и инициализирует его значением `init_value`. Ужасно много работы. Лучше сразу проинициализировать объект корректным начальным значением. Используйте:

```
base( const  char *init_value ) : s(init_value)
{}
```

Теперь строка `s` будет инициализирована правильно, и не нужен вызов **`operator`**`=()` для ее повторной инициализации.

Настоящее правило также применимо к базовым классам, доступным из конструктора производного класса, поэтому они должны инициализироваться до выполнения конструктора производного класса. Базовые классы инициализируются перед членами производного класса, потому что члены производного класса невидимы в базовом классе. Подведем итог - объекты инициализируются в следующем порядке:

- Базовые классы в порядке объявления.
- Поля данных в порядке объявления.

Лишь затем выполняется конструктор производного класса. Одно последнее предостережение. Заметьте, что порядок объявления управляет порядком инициализации. Порядок, в котором элементы появляются в списке инициализации членов, является несущественным. Более того,

порядок объявления не должен рассматриваться как неизменный. Например, вы можете изменить порядок, в котором объявлены поля данных. Рассмотрим следующее определение класса где-нибудь в заголовочном файле:

```
class wilma
{
   int y;
   int x;
public:
   wilma( int ix );
};
```

Вот определение конструктора в файле *.c*:

```
wilma::wilma( int ix ) : y(ix * 10), x(y + 1)
{}
```

Теперь допустим, что какой-то сопровождающий программист переставит поля данных в алфавитном порядке, поменяв местами `x` и `y`. Этот конструктор больше не работает: поле `x` инициализируется первым, потому что оно первое в определении класса, и инициализируется значением `y+1`, но поле `y` еще не инициализировалось.

Исправьте код, исключив расчет на определенный порядок инициализации:

```
wilma::wilma( int ix ) : y(ix * 10), x((ix *10) + 1)
{}
```

130. Конструкторы копий должны использовать списки инициализации членов

У наследования тоже есть свои проблемы с копированием. Конструктор копии все же остается конструктором, поэтому здесь также применимы результаты обсуждения предыдущего правила. Если у конструктора копии нет списка инициализации членов, то для базовых классов и вложенных объектов используется конструктор по умолчанию. Так как список инициализации членов отсутствует в следующем определении конструктора копии, то компонент базового класса в объекте производного класса инициализируется с использованием `base(void)`, а поле `s` инициализируется с использованием `string::string(void)`:

```
class base
{
public:
   base( void );                    // конструктор по умолчанию
   base( const base &r );           // конструктор копии
   const base &operator=( const base &r );
```

```
};

class derived
{
   string s;              // класс имеет конструктор копии
public:
   derived( const derived &r )
};

derived::derived( const derived &r )
{}
```

Чтобы гарантировать копирование также поля `string` и компонента базового класса в объекте производного класса, используйте следующее:

```
derived::derived( const derived &r ) : base(r), s(r.s) {}
```

131. Производные классы должны обычно определять конструктор копии и функцию operator=( )

При наследовании есть и другая связанная с копированием проблема. В одном месте руководства[10] по языку Си++ недвусмысленно заявлено: "конструкторы и функция **operator**=() не наследуются". Однако далее в этом же документе говорится, что существуют ситуации, в которых компилятор не может создать конструктор копии или функцию **operator**=(), которые бы корректно вызывались вслед за функциями базового класса. Так как нет практической разницы между унаследованной и сгенерированной функциями **operator**=(), которые ничего не делают, кроме вызова функции базового класса, то эта неопределенность вызвала много бед.

Я наблюдал два полностью несовместимых поведения компиляторов, столкнувшихся с этой дилеммой. Некоторые компиляторы считали правильным, чтобы сгенерированные компилятором конструкторы копий и функции **operator**=() вызывались автоматически после конструкторов и функций **operator**=() базового класса (и вложенного объекта).[11] Это как раз тот способ, который, по мнению большинства,

---

[10] Книга Эллис и Страуструпа *"The Annotated C++ Reference Manual"* (Reading: Addison Wesley, 1990), использованная в качестве базового документа комитетом ISO/ANSI по Си++♣.

♣Имеется перевод на русский язык под редакцией А.Гутмана *"Справочное руководство по языку программирования Си++ с комментариями"* (М.: Мир, 1992). —*Прим.перев*.

[11] Конечно, конструкторы копий и функции **operator**=(), создаваемые вами (в отличие от компилятора), никогда не вызывают своих двойников из базового класса автоматически.

реализуется языком программирования. Другими словами, со следующим кодом проблем не будет:

```
class base
{
public:
   base( const base &r );
   const base &operator=( const base &r );
};

class derived : public base
{
   string s;
   // нет операции operator=() или конструктора копии
};

derived x;
derived y = x; // вызывает конструктор копии базового класса
               // для копирования базового класса. Также
               // вызывает конструктор копии строки для
               // копирования поля s.
x = y; // вызывает функцию базового класса operator=() для
       // копирования базового класса. Также вызывает
       // строковую функцию operator=() для копирования поля s.
```

Если бы все компиляторы работали таким образом, то проблемы бы не было. К несчастью, некоторые компиляторы принимают ту самую директиву "не наследуются" за чистую монету. Только что представленный код не будет работать с этими компиляторами. В них сгенерированные компилятором конструктор копии и функция **operator**=() производного класса действуют так, как будто бы их эквиваленты в базовом классе (и вложенном объекте) просто не существуют. Другими словами, конструктор по умолчанию — без аргументов — вызывается для копирования компонента базового класса, а почленное копирование — которое может выполняться просто функцией memcpy() — используется для поля. Мое понимание пересмотренного проекта стандарта Си++ ISO/ANSI позволяет сделать вывод, что такое поведение некорректно, но в течение некоторого времени вам придется рассчитывать на худшее, чтобы обеспечивать переносимость. Следовательно, это, вероятно, хорошая мысль — всегда помещать в производный класс конструктор копии и функцию **operator**=(), которые явно вызывают своих двойников из базового класса. Вот реализация предыдущего производного класса для самого худшего случая:

```
class derived : public base
{
   string s;
```

```
public:
   derived( const derived &r );
   const derived &operator=( const derived &r );
};
//-----------------------------------------------------------
derived::derived( const derived &r ) : base(r), s(r.s)
{}
//-----------------------------------------------------------
const derived &derived::operator=( const derived &r )
{
   (* (base*)this) = r;
   s = r.s;
}
```

Список инициализации членов в конструкторе копии описан ранее. Следующий отрывок из функции **operator**=() нуждается в некотором пояснении:

```
(* (base*)this) = r;
```

Указатель **this** указывает на весь текущий объект; добавление оператора приведения преобразует его в указатель на компонент базового класса в текущем объекте — (base*)**this**. (* (base*)**this**) является самим объектом, а выражение (* (base*)**this**) = r передает этому объекту сообщение, вызывая функцию **operator**=() базового класса для перезаписи информации из правого операнда в текущий объект. Вы могли бы заменить этот код таким образом:

```
base::operator=( r );
```

но я видел компиляторы, которые бракуют этот оператор, если в базовом классе не объявлена явно функция **operator**=(). Первая форма работает независимо от того, объявлена явно **operator**=(), или нет. (Если не объявлена, то у вас будет по умолчанию реализовано почленное копирование).

132. Конструкторы, не предназначенные для преобразования типов, должны иметь два или более аргумента♣

Си++ использует конструкторы для преобразования типов. Например, конструктор **char*** в 9-ой строке листинга 7 на странице 155 также обрабатывает следующую операцию приведения:

```
char *pchar = "абвг";
(string) pchar;
```

Запомните, что приведение является операцией времени выполнения, которая создает временную переменную нужного типа и инициализирует ее из аргумента. Если приводится класс, то для инициализации используется конструктор. Следующий код работает прекрасно, потому что строковая константа **char*** беспрепятственно преобразуется в `string` для передачи в функцию `f()`:

```
f( const string &s );
// ...
f( "белиберда" );
```

Проблема состоит в том, что мы иногда не желаем использовать конструктор для неявного преобразования типов. Рассмотрим следующий контейнер массива, которым поддерживается целочисленный конструктор, определяющий размер этого массива:

```
class array
{
   // ...
public:
   array( int initial_size );
};
```

Вероятно вы все же не захотите, чтобы следующий код работал:

```
f( const array &a );
// ...
f( isupper(*str) );
```

(Этот вызов передает `f()` пустой одноэлементный массив, если `*str` состоит из заглавных букв, или массив без элементов, если `*str` — из строчных букв).

Единственным способом подавления такого поведения является добавление второго аргумента в конструктор, потому что конструкторы с несколькими аргументами никогда не используются неявно:

♣ Стандартом языка для этого предусмотрено ключевое слово **explicit**. — *Ред.*

```
class array
{
   // ...
public:
   enum bogus { set_size_to };
   array( bogus, int initial_size );
};

array ar( array::set_size_to, 128 );
```

Это по настоящему уродливо, но у нас нет выбора. Заметьте, что я не дал аргументу `bogus` имени, потому что он используется только для выбора функции.

133. Используйте счетчики экземпляров объектов для инициализации на уровне класса

Несколько разделов назад я рассматривал использование счетчика статических глобальных объектов для управления инициализациями на уровне библиотеки. В Си++ у нас есть лучшие варианты, потому что мы может использовать определение класса для ограничения области действия:

```
class window
{
   static int num_windows;
public:
   window();
   ~window();
};

int window::num_windows = 0;

window::window()
{
   if( ++num_windows == 1 )  // только что создано первое окно
      initialize_video_system();
}

window::~window()
{
   if( --num_windows == 0 )       // только что уничтожено
      shut_down_video_system();  // последнее окно
}
```

Наконец, счетчик экземпляров объектов может быть также использован в качестве счетчика числа вызовов для обеспечения инициализации на уровне подпрограммы:

```
f()
{
   static int have_been_called = 0;
   if( !have_been_called )
   {
      have_been_called = 1;
      do_one_time_initializations();
   }
}
```

134. Избегайте инициализации в два приема

135. Суперобложки на Си++ для существующих интерфейсов редко хорошо работают

Как правило, переменная должна инициализироваться во время объявления. Разделение инициализации и объявления иногда обусловливается плохим проектированием в программе, которая написана не вами, как в следующем фрагменте, написанном для выполнения совместно с библиотекой MFC Microsoft:

```
f( CWnd *win ) // CWnd - это окно
{
   // Следующая строка загружает "буфер" с шапкой окна
   // (текстом в строке заголовка)

   char buf[80];       /* = */
   win->GetWindowText(buf, sizeof(buf));
   // ...
}
```

Так как я должен выполнить инициализацию при помощи явного вызова функции, то умышленно нарушаю свое правило "один оператор в строке" для того, чтобы, по крайней мере, вместить объявление и инициализацию в одной и той же строке.

Здесь имеется несколько проблем, первая из которых заключается в плохом проектировании класса `CWnd` (представляющем окно). Так как у окна есть "текстовый" атрибут, хранящий заголовок, то вы должны иметь возможность доступа к этому атрибуту подобным образом:

```
CString caption = win->caption();
```

и вы должны иметь возможность модифицировать этот атрибут так:

```
win->caption() = "новое содержание";
```

но вы не можете сделать этого в текущей реализации. Главная проблема состоит в том, библиотека MFC не была спроектирована в объектно-ориентированном духе — т.е. начать с объектов, затем выбрать, какие сообщения передавать между ними и какими атрибутами их наделить.

Вместо этого проектировщики Microsoft начали от существующего процедурного интерфейса (API Си — интерфейса прикладного программирования для Windows на Си) и добавили к нему суперобложку на Си++, тем самым увековечив все проблемы существующего интерфейса. Так как в API Си была функция с именем `GetWindowText()`, то проектировщики беззаботно сымитировали такой вызов при помощи функции-члена в своей оболочке `CWnd`. Они поставили заплату на интерфейс при помощи следующего вызова:

```
CString str;
win->GetWindowText( str );
```

но это — не решение по двум причинам: по-прежнему требуется инициализация в два приема, и аргумент является ссылкой на результат.

Главный урок состоит в том, что проекты, основанные на процедурном подходе, радикально отличаются от объектно-ориентированных проектов. Обычно невозможно использовать код из одного проекта в другом без большой переработки. Простая оболочка из классов Си++ вокруг процедурного проекта не сделает его объектно-ориентированным.

Поучительно, я думаю, пошарить вокруг в поисках решения текущей проблемы с помощью Си++, но предупреждаю вас — здесь нет хорошего решения (кроме перепроектирования библиотеки классов). Моя первая попытка сделать оболочку вокруг `CWnd` показана на листинге 11.

Для обеспечения возможности `win->text() = "Новый заголовок"` необходим вспомогательный класс (`window::caption`). Вызов `text()` возвращает объект заголовка, которому затем передается сообщение присваиванием.

Главная проблема на листинге 11 заключается в том, что библиотека MFC имеет много классов, унаследованных от `CWnd`, и интерфейс, реализованный в классе `window`, не будет отражен в других потомках `CWnd`. Си++ является компилируемым языком, поэтому нет возможности вставлять класс в середину иерархии классов без изменения исходного кода.

Листинг 12 определяет другое решение для смеси Си++ с MFC. Я выделил класс `window::caption` в отдельный класс, который присоединяется к окну, когда оно инициализируется. Используется подобным образом:

```
f(CWnd *win)
{
   caption cap( win )

   CString s = cap; // поддерживается преобразование в CString.
   cap = "Новый заголовок";  // использует операцию
```

```
                                    // operator=(CString&)
}
```

Мне не нравится то, что изменение заголовка `caption` меняет также окно, к которому этот заголовок присоединен в этом последнем примере. Скрытая связь между двумя объектами может сама по себе быть источником недоразумений, будучи слишком похожей на побочный эффект макроса. Как бы то ни было, листинг 12 решает проблему инициализации.

Листинг 11. Обертка для `CWnd`: первая попытка

```
1  class window : public CWnd
2  {
3  public:
4     class caption
5     {
6        CWnd *target_window;
7
8     private: friend class window;
9        caption( CWnd *p ) : target_window(p) {}
10
11     public:
12        operator CString ( void ) const;
13        const caption &operator=( const CString &s );
14     };
15
16     caption text( void );
17  };
18  //-------------------------------------------------------
19  caption window::text( void )
20  {
21     return caption( this );
22  }
23  //-------------------------------------------------------
24  window::caption::operator CString( void ) const
25  {
26     CString output;
27     target_window->GetWindowText( output );
28     return output;                  // возвращает копию
29  }
30  //-------------------------------------------------------
31  const caption &window::caption::operator=(const CString &s)
32  {
33     target_window->SetWindowText( s );
34     return *this;
35  }
```

Листинг 12. Заголовочный объект

```
 1  class caption
 2  {
 3     CWnd target_window;
 4  public:
 5     window_text( CWnd *win ) : target_window( win ) {};
 6
 7     operator const CString( void );
 8     const CString &operator=( const CString &r );
 9  };
10
11   inline caption::operator CString( void );
12   {
13      CString output;
14      target_window->GetWindowText( output );
15          return output;
16   }
17
18   inline const CString &caption::operator=(const CString &s )
19   {
20      // возвращает тип CString (вместо типа заголовка
21      // "caption"), поэтому будет срабатывать a = b = "абв"
22
23      target_window->SetWindowText( s );
24      return s;
26   }
```

Часть 8д. Виртуальные функции

Виртуальные функции придают объекту производного класса способность модифицировать поведение, определенное на уровне базового класса (или предоставить какие-то возможности, в которых базовый класс испытывал потребность, но не мог их реализовать обычно из-за того, что информация, нужная для этой реализации, объявляется на уровне производного класса). Виртуальные функции являются центральными для объектно-ориентированного проектирования, потому что они позволяют вам определить базовый класс общего назначения, не требуя знания особенностей, которые могут быть предусмотрены лишь производным классом. Вы можете писать программу, которая думает, что манипулирует объектами базового класса, но на самом деле во время выполнения воздействует на объекты производного класса. Например, вы можете написать код, помещающий объект в обобщенную структуру данных `data_structure`, но на самом деле во время выполнения он вставляет его в `tree` или `linked_list` (классы, производные от `data_structure`). Это настолько фундаментальная объектно-ориентированная операция, что программа на Си++, которая не использует виртуальные функции, вероятно, просто плохо спроектирована.

136. Виртуальные функции — это те функции, которые вы не можете написать на уровне базового класса

Виртуальные функции существуют ради двух целей. Во-первых, виртуальные функции определяют возможности, которые должны иметь все производные классы, но которые не могут быть реализованы на уровне базового класса. Например, вы можете сказать, что все объекты-фигуры `shape` должны быть способны себя распечатать. Вы не можете написать функцию `print()` на уровне базового класса, потому что геометрическая информация хранится в производных классах (круге `circle`, линии `line`, многоугольнике `polygon` и т.д.). Поэтому вы делаете `print()` виртуальной в базовом классе и фактически определяете эту функцию в производном классе.

Второй целью являются вспомогательные виртуальные функции. Возьмем в качестве примера наш класс `storable`. Для хранения объекта в отсортированной структуре данных сохраняемый объект должен быть способен сравнивать себя с другим сохраненным объектом. То есть эта

функция базы данных будет выглядеть примерно так:

```
add( storable *insert )
{
   storable *object_already_in_database;
   // ...
   if( object_already_in_database->cmp(insert) < 0 )
   // вставить объект в базу данных
}
```

Объект `storable` не может определить функцию `cmp()`, потому что информация, необходимая для сравнения (ключ), находится в объекте производного класса, а не в базовом классе `storable`. Поэтому вы делаете функцию виртуальной в классе `storable` и предусматриваете ее в производном классе. Кстати, эти вспомогательные функции никогда не будут открытыми (**`public`**).

137. Виртуальная функция не является виртуальной, если вызывается из конструктора или деструктора

Это не столько правило, сколько констатация факта, хотя она и будет для многих неожиданностью. Базовые классы инициализируются перед производными классами. К тому же, по-видимому, функции производного класса имеют доступ к данным этого класса; в ином случае не было бы смысла в помещении этих функций в производный класс. Если бы конструктор базового класса мог вызывать функцию производного класса через механизм виртуальных функций, то эта функция могла бы с пользой использовать инициализированные поля данных производного класса.

Чтобы сделать суть кристально ясной, давайте взглянем на то, что происходит под капотом. Механизм виртуальных функций реализован посредством таблицы указателей на функции. Когда вы объявляете класс, подобный следующему:

```
class storable
{
   int stuff;
public:
   storable( void );

   virtual void print( void              );
   virtual void virtf( void              );
   virtual int  cmp  ( const storable &r ) = 0;

   int nonvirtual( void );
};
```

```
storable::storable  ( void ) { stuff = 0;
}
void storable::print( void ) { /* материал для отладки print */
}
void storable::virtf( void ) { /* делай что-нибудь */
}
int  storable::nonvirtual( void ) {
}
```

Лежащее в основе определение класса (сгенерированное компилятором) может выглядеть подобно этому:

```
int _storable__print          ( storable *this ) { /* ... */ }
int _storable__virtf          ( storable *this ) { /* ... */ }
int _storable__nonvirtual     ( storable *this ) { /* ... */ }

typedef void (*_vtab[])(...); // массив указателей на функции

_vtab _storable__vtab
{
   _storable__print,
   _storable__virtf,
   NULL             // метка-заполнитель для функции сравнения
};

typedef struct storable
{
   _storable__vtab *_vtable;
   int stuff;
}
storable;

_storable__ctor( void )       // конструктор
{
   _vtable = _storable__vtable; // Эту строку добавляет
                                // компилятор.
   stuff = 0;                   // Эта строка из исходного
кода.
}
```

Когда вы вызываете невиртуальную функцию, используя такой код, как:

```
storable *p;
p->nonvirtual();
```

то компилятор в действительности генерирует:

```
_storable__nonvirtual( p )
```

Если вы вызываете виртуальную функцию, подобную этой:

```
p->print();
```

то получаете нечто совершенно отличное:

```
( p->_vtable[0] )( p );
```

Вот таким-то окольным путем, посредством этой таблицы и работают виртуальные функции. Когда вы вызываете функцию производного класса при помощи указателя базового класса, то компилятор даже не знает, что он обращается к функции производного класса. Например, вот определение производного класса на уровне исходного кода:

```
class employee : public storable
{
   int derived_stuff;
   // ...
public:
   virtual int cmp( const storable &r );
};

/* виртуальный */ int employee::print( const storable &r ) { }
/* виртуальный */ int employee::cmp  ( const storable &r ) { }
```

А вот что сделает с ним компилятор:

```
int _employee__print( employee *this          ) { /* ... */ }
int _employee__cmp  ( employee *this, const storable *ref_r )
{ /* ... */ }

_vtab _employee_vtable =
{
   _employee__print,
   _storable_virtf,  // Тут нет замещения в производном классе,
                     // поэтому используется указатель на
                     // функцию базового класса.
   _employee_cmp
};

typedef struct employee
{
   _vtab *_vtable;    // Генерируемое компилятором поле данных.
   int stuff;         // Поле базового класса.
   int derived_stuff; // Поле, добавленное в объявлении
                      // производного класса.
}
employee;

_employee__ctor( employee *this ) // Конструктор по умолчанию,
{                                 // генерируемый компилятором.
   _storable_ctor();        // Базовые классы инициализируются
                            // в первую очередь.
_vtable = _employee_vtable; // Создается таблица виртуальных
}                           // функций.
```

Компилятор переписал те ячейки в таблице виртуальных функций, которые содержат замещенные в производном классе виртуальные

функции. Виртуальная функция (`virtf`), которая не была замещена в производном классе, остается инициализированной функцией базового класса.

Когда вы создаете во время выполнения объект таким образом:

```
storable *p = new employee();
```

то компилятор на самом деле генерирует:

```
storable *p;
p = (storable *)malloc( sizeof(employee) );
_employee_ctor( p );
```

Вызов `_employee_ctor()` сначала инициализирует компонент базового класса посредством вызова `_sortable_ctor()`, которая добавляет таблицу этой виртуальной функции к своей таблице и выполняется. Затем управление передается обратно к `_employee_ctor()` и указатель в таблице виртуальной функции переписывается так, чтобы он указывал на таблицу производного класса.

Отметьте, что, хотя p теперь указывает на `employee`, код `p->print()` генерирует точно такой же код, как и раньше:

```
( p->_vtable[0] )( p );
```

Несмотря на это, теперь `p` указывает на объект производного класса, поэтому вызывается версия `print()` из производного класса (так как `_vtable` в объекте производного класса указывает на таблицу производного класса). Крайне необходимо, чтобы эти две функции `print()` располагались в одной и той же ячейке своих таблиц смешений, но это обеспечивается компилятором.

Возвращаясь к основному смыслу данного правила, отметим, что при рассмотрении того, как работает конструктор, важен порядок инициализации. Конструктор производного класса перед тем, как он что-либо сделает, вызывает конструктор базового класса. Так как `_vtable` в конструкторе базового класса указывает на таблицу виртуальных функций базового класса, то вы лишаетесь доступа к виртуальным функциям базового класса после того, как вызвали их. Вызов `print` в конструкторе базового класса все так же дает:

```
( this->_vtable[0] )( p );
```

но `_vtable` указывает на таблицу базового класса и `_vtable[0]` указывает на функцию базового класса. Тот же самый вызов в конструкторе производного класса даст версию `print()` производного класса, потому что `_vtable` будет перекрыта указателем на таблицу производного класса к тому времени, когда была вызвана `print()`.

Хотя я и не показывал этого прежде, то же самое происходит в деструкторе. Первое, что делает деструктор, — это помещает в `_vtable` указатель на таблицу своего собственного класса. Только после этого он выполняет написанный вами код. Деструктор производного класса вызывает деструктор базового класса на выходе (в самом конце — после того, как выполнен написанный пользователем код).

138. Не вызывайте чисто виртуальные функции из конструкторов

Это правило вытекает из только что рассмотренной картины. Определение "чисто" виртуальной функции (у которой `=0` вместо тела) приводит к тому, что в таблицу виртуальных функций базового класса помещается `NULL` вместо обычного указателя на функцию. (В случае "чисто" виртуальной функции нет функции, на которую необходимо указывать). Если вы вызываете чисто виртуальную функцию из конструктора, то используете таблицу базового класса и на самом деле вызываете функцию при помощи указателя `NULL`. Вы получите дамп оперативной памяти на машине с UNIX и "Общая ошибка защиты" в системе Windows, но MS-DOS просто исполнит то, что вы просили, и попытается выполнить код по адресу `0`, считая его правильным.

139. Деструкторы всегда должны быть виртуальными

Рассмотрим такой код:

```
class base
{
   char *p;

   ~base() { p = new char[SOME_SIZE]; }
   base() { delete p; }
};

class derived : public base
{
   char *dp;

   ~derived() { dp = new char[[SOME_SIZE]; }
   derived() { delete dp; }
};
```

Теперь рассмотрим этот вызов:

```
base *p = new derived;
// ...
delete p;
```

Запомните, что компилятор не знает, что p на самом деле указывает на объект производного класса. Он исходит из того, что p указывает на объявленный тип base. Следовательно, **delete** p в действительности превращается в:

```
_base__destructor(p);
free(p);
```

Деструктор производного класса никогда не вызывается. Если вы переопределите эти классы, сделав деструктор виртуальным:

```
virtual ~base() { /* ... */ }
```

то компилятор получит доступ к нему при помощи таблицы виртуальных функций, просто как к любой другой виртуальной функции. Так как деструктор теперь виртуальный, то **delete** p превращается в:

```
( p->_vtable[DESTRUCTOR_SLOT] ) (p);
```

Так как p указывает на объект производного класса, то вы получаете деструктор производного класса, который после выполнения компоненты производного класса вызывает деструктор базового.

140. Функции базового класса, имеющие то же имя, что и функции производного класса, обычно должны быть виртуальными

Помните, что открытая (**public**) функция является обработчиком сообщений. Если базовый класс и производный класс оба имеют обработчики сообщений с одним и тем же именем, то вы скажете, что объект производного класса должен делать что-то отличное от объекта базового класса, чтобы обрабатывать то же самое сообщение. Весь смысл наследования в том, чтобы иметь возможность писать код общего назначения на языке объектов базового класса и обеспечивать работу этого кода даже с объектами производного класса. Следовательно, сообщение должно обрабатываться функцией производного класса, а не базового.

Одним распространенным исключением из этого правила является перегрузка операций, где базовый класс может определять некий набор перегруженных операций, а производный класс желает добавить дополнительные перегрузки (в отличие от изменения поведения перегруженных операций базового класса). Хотя перегруженные функции в этих двух классах будут иметь одинаковые имена, у них непременно будут различные сигнатуры, поэтому они не могут быть виртуальными.

141. Не делайте функцию виртуальной, если вы не желаете, чтобы производный класс получил контроль над ней

Я читал, что все функции-члены необходимо делать виртуальными "просто на всякий случай". Это плохой совет. Ведь вы не желаете, конечно, чтобы производный класс получил контроль надо всеми вашими вспомогательными функциями; иначе вы никогда не будете способны писать надежный код.

142. Защищенные функции обычно должны быть виртуальными

Одним из смягчающих факторов в ранее описанной ситуации со сцеплением базового и производного классов является то, что объекту производного класса Си++ едва когда-либо нужно посылать сообщение компоненту своего базового класса. Производный класс наследует назначение (и члены) от базового класса и обычно добавляет к нему назначение (и члены), но производный класс часто не вызывает функции базового класса. (Естественно, производный класс никогда не должен получать доступ к данным базового класса). Единственным иисключением являются виртуальные функции, которые можно рассматривать как средство изменения поведения базового класса. Сообщения часто передаются замещающей функцией производного класса в эквивалентную функцию базового класса. То есть, виртуальное замещение производного класса часто образует цепь с функцией базового класса, которую оно заместило. Например, класс `CDialog` из MFC реализует диалоговое окно Windows (тип окна для ввода данных). Этот класс располагает виртуальной функцией `OnOk()`, которая закрывает диалоговое окно, если пользователь щелкнул по кнопке с меткой "OK". Вы определяете свое собственное диалоговое окно путем наследования от `CDialog` и можете создать замещение `OnOk()`, которое будет выполнять проверку правильности данных перед тем, как позволить закрыть это диалоговое окно. Ваше замещение образует цепь с функцией базового класса для действительного выполнения закрытия:

```
class mydialog : public CDialog
{
   // ...
private:
   virtual OnOk( void );
};
```

```
/* виртуальный */ mydialog::OnOk( void )
{
   if( data_is_valid() )
      CDialog::OnOk();    // Послать сообщение базовому классу
   else
      beep();             // Обычно содержательное сообщение
                          // Windows об ошибке
}
```

Функция `OnOk()` является закрытой в производном классе, потому что никто не будет посылать сообщение `OnOk()` объекту `mydialog`. `OnOk()` базового класса не может быть закрытой, потому что вам нужно образовать цепь с ней из замещения производного класса. Вы не желаете, чтобы `CDialog::OnOk()` была открытой, потому что снова никто не должен посылать сообщение `OnOk()` объекту `CDialog`. Поэтому вы делаете ее защищенной. Теперь замещение из производного класса может образовать цепочку с `OnOk()`, но эта функция не доступна извне.

Это не очень удачная мысль — использовать защищенный раздел описания класса для обеспечения секретного интерфейса с базовым классом, которым сможет пользоваться лишь производный класс, потому что это может скрыть отношение сцепления. Хотя подобная защищенная функция иногда единственный выход из ситуации, нормальный открытый интерфейс обычно является лучшей альтернативой.

Заметьте, что это правило не имеет обратного действия. Хотя защищенные функции обычно должны быть виртуальными, многие виртуальные функции являются открытыми.

143. Опасайтесь приведения типов (спорные вопросы Си++)

Приведение типов в Си рассмотрено ранее, но и в Си++ приведение вызывает проблемы. В Си++ у вас также существует проблема нисходящего приведения — приведения указателя или ссылки на базовый класс к производному классу. Эта проблема обычно появляется при замещениях виртуальных функций, потому что сигнатуры функций производного класса должны точно совпадать с сигнатурами базового класса. Рассмотрим этот код:

```
class base
{
public:
   virtual int operator==( const base &r ) = 0;
};

class derived
{
```

```
   char *key;

public:
   virtual int operator==( const base &r )
   {
      return strcmp(key, ((const derived &)r).key ) == 0;
   }
};
```

К несчастью, здесь нет гарантии, что передаваемый аргумент `r` действительно ссылается на объект производного класса. Он не может ссылаться на объект базового класса из-за того, что функция чисто виртуальная: вы не можете создать экземпляр объекта `base`. Тем не менее, `r` мог бы быть ссылкой на объект некоего другого класса, унаследованного от `base`, но не являющегося классом `derived`. С учетом предыдущего определения следующий код не работает:

```
class other_derived : public base
{
   int key;
   // ...
};

f()
{
   derived       dobj;
   other_derived other;

   if( derived == other_derived )
      id_be_shocked();
}
```

Комитет ISO/ANSI по Си++ рекомендовал механизм преобразования типов во время выполнения, который решает эту проблему, но на момент написания этой книги многие компиляторы его не поддерживают. Предложенный синтаксис выглядит подобным образом:

```
class derived : public base
{
   char *key;
public:
   virtual int operator==( const base &r )
   {
      derived *p = dynamic_cast<derived *>( &r );

      return !p ? 0 : strcmp(key, ((const derived &)r).key )==0;
   }
};
```

Шаблон функции `dynamic_cast<t>` возвращает `0`, если операнд не может быть безопасно преобразован в тип `t`, иначе он выполняет

преобразование.

Это правило является также хорошей демонстрацией того, почему вы не хотите, чтобы все классы в вашей иерархии происходили от общего класса `object`. Почти невозможно использовать аргументы класса `object` непосредственно, потому что сам по себе класс `object` почти лишен функциональности. Вы поймаете себя на том, что постоянно приводите указатели на `object` к тому типу, который на самом деле имеет переданный аргумент. Это приведение может быть опасным без использования преобразования типов во время выполнения, потому что вы можете преобразовать в неверный тип. Приведение уродливо даже в виде преобразования во время выполнения, добавляя ненужный беспорядок в программу.

144. Не вызывайте конструкторов из операции operator=( )

Хотя это правило говорит о перегруженном присваивании, на самом деле оно посвящено проблеме виртуальных функций. Соблазнительно реализовать **operator**=() следующим образом:

```
class some_class
{
public:
   virtual
     ~some_class( void                                  );
      some_class( void                                  );
      some_class( const some_class &r );

   const some_class &operator=( const some_class &r );
};

const some_class &operator=( const some_class &r )
{
   if( this != &r )
   {
      this->~some_class();
      new(this) some_class(r);
   }
   return *this;
}
```

Этот вариант оператора **new** инициализирует указываемый **this** объект как объект `some_class`, в данном случае из-за аргумента `r` используя конструктор копии.[12]

[12] Некоторые компиляторы в действительности позволяют выполнить явный вызов конструктора, поэтому вы, вероятно, сможете сделать точно так же:

Есть серьезные причины не делать показанное выше. Во-первых, это не будет работать после наследования. Если вы определяете:

```
class derived : public some_class
{
public:
   ~derived();

   // Предположим, что генерированная компилятором операция
   // operator=() выполнится за операцией operator=() базового
   // класса.
}
```

Вследствие того, что деструктор базового класса определен (правильно) как виртуальный, обращение предыдущего базового класса к:

```
this->~some_class()
```

вызывает деструктор производного класса, поэтому вы уничтожите значительно больше, чем намеревались. Вы можете попытаться исправить эту проблему, изменив вызов деструктора на:

```
this->some_class::~some_class();
```

Явное упоминание имени класса — `some_class::` в этом примере — подавляет механизм виртуальной функции. Функция вызывается, как если бы она не была виртуальной.

Деструктор не является единственной проблемой. Рассмотрим простое присваивание объектов производного класса:

```
derived d1, d2;
d1 = d2;
```

Операция производного класса **`operator`**`=()` (вне зависимости от того, генерируется она компилятором или нет) образует цепочку с **`operator`**`=()` базового класса, который в настоящем случае использует оператор **`new`**`()` для явного вызова конструктора базового класса. Конструктор, тем не менее, делает значительно больше, чем вы можете видеть в определении. В частности, он инициализирует указатель таблицы виртуальных функций так, чтобы он указывал на таблицу его

```
const some_class &operator=( const some_class &r )
{
   if( this != &r )
   {
      this->~some_class();
      this->some_class::some_class( r );
   }
}
```

Тем не менее, такое поведение является нестандартным.

класса. В текущем примере перед присваиванием указатель `vtable` указывает на таблицу производного класса. После присваивания указатель `vtable` указывает на *таблицу базового класса*; он был переинициализирован неявным вызовом конструктора при вызове **`new`** в перегруженной операции **`operator`**`=()`.

Таким образом, вызовы конструкторов в операции **`operator`**`=()` просто не будут работать, если есть таблица виртуальных функций. Так как вы можете знать или не знать, на что похожи определения вашего базового класса, то вы должны исходить из того, что таблица виртуальных функций имеется, и поэтому не вызывайте конструкторов.

Лучшим способом устранения дублирования кода в операции присваивания **`operator`**`=()` является использование простой вспомогательной функции:

```
class some_class
{
   void create  ( void                );
   void create  ( const some_class &r );
   void destroy ( void                );
public:

   virtual
      ~some_class( void ) { destroy(); }
       some_class( void ) { create();  }

   const some_class &operator=( const some_class &r );
};
inline const some_class &some_class::operator=( const
                                                 some_class &r )
{
   destroy();
   create( r );
}
inline some_class::some_class( void )
{
   create();
}
~some_class::some_class( void )
{
   destroy();
}
```

Часть 8е. Перегрузка операций

145. Операция — это сокращение (без сюрпризов)

Операция — это не произвольный значок, означающий все, что вы ни пожелаете. Это аббревиатура англоязычного слова. Например, символ + значит "прибавить", поэтому вы не должны заставлять перегруженный **`operator`**`+()` делать что-нибудь еще. Хотя здесь все ясно (вы можете определить `a + b` для вычитания `b` из `a`, но не должны делать этого), я на самом деле веду речь о проблемах более творческого характера.

Вы можете благоразумно доказывать, что, когда выполняете конкатенацию, то "прибавляете" одну строку к концу другой, поэтому перегрузка + для конкатенации может быть приемлема. Вы также можете доказывать, что разумно использовать операции сравнения для лексикографического упорядочивания в классе `string`, поэтому перегрузка операций <, == и т.д. также вероятно пойдет. Вы не сможете аргументировано доказать, что `-` или `*` имеют какой-нибудь смысл по отношению к строкам.

Другим хорошим примером того, как нельзя действовать, является интерфейс Си++ *iostream*. Использование сдвига (<<) для обозначения "вывод" является нелепым. Ваши функции вывода в Си назывались `printf()`, а не `shiftf()`. Я понимаю, что Страуструп выбрал сдвиг, потому что он сходен с механизмом перенаправления ввода/вывода различных оболочек UNIX, но этот довод на самом деле не выдерживает проверки. Страуструп исходил из того, что все программисты на Си++ понимают перенаправление в стиле UNIX, но эта концепция отсутствует в некоторых операционных системах — например, в Microsoft Windows. К тому же, для того, чтобы аналогия была полной, операция > должна быть перегружена для выполнения операции затирания, а >> — добавления в конец. Тем не менее, тот факт, что > и >> имеют различный приоритет, делает реализацию такого поведения затруднительной. Дело осложняется тем, что операторы сдвига имеют неправильный уровень приоритета. Оператор типа **`cout`** `<< x += 1` не будет работать так, как вы ожидаете, потому что у << более высокий приоритет, чем у +=, поэтому оператор интерпретируется как `(`**`cout`**` << x) += 1`, что неверно. Си++ нуждается в расширяемости, обеспечиваемой системой *iostream*, но он вынужден добиваться ее за счет введения операторов "ввода" и "вывода", имеющих низший приоритет по отношению к любому оператору языка.

Аналогия проблеме "сдвиг как вывод" может быть найдена в проектировании компьютерных систем. Большинство проектировщиков аппаратуры были бы счастливы использовать + вместо OR, а * вместо AND, потому что такая запись используется во многих системах проектирования электронных компонентов. Несмотря на это, перегрузка операции **operator**+() в качестве OR явно не нужна в Си++. К тому же, лексема << означает "сдвиг" в Си и Си++; она не означает "вывод".

Как завершающий пример этой проблемы — я иногда видел реализации класса "множество", определяющие | и & со значениями "объединение" и "пересечение". Это может иметь смысл для математика, знакомого с таким стилем записи, но при этом не является выражением ни Си, ни Си++, поэтому будет незнакомо для вашего среднего программиста на Си++ (и вследствие этого с трудом сопровождаться). Амперсанд является сокращением для AND; вы не должны назначать ему произвольное значение. Нет абсолютно ничего плохого в a.Union(b) или a.intersect(b). (Вы не можете использовать a.union(b) со строчной буквой u, потому что **union** является ключевым словом).

146. Используйте перегрузку операций только для определения операций, имеющих аналог в Си (без сюрпризов)

Перегрузка операций была реализована в языке, прежде всего, для того, чтобы вы могли интегрировать разработанный вами арифметический тип в существующую арифметическую систему языка Си. Этот механизм никогда не предназначался в качестве средства расширения этой системы. Следовательно, перегрузку операций лучше применять, используя классы для реализации лишь арифметических типов.

Тем не менее, также разумно использовать перегруженные операции и там, где аналогии с Си незаметны. Например, большинство классов будет перегружать присваивание. Перегрузка **operator**==() и **operator**!=() также разумна в большинстве классов.

Менее ясным (и более противоречивым) примером является класс "итератор". Итератор является средством просмотра каждого члена структуры данных, и он используется почти точно так же, как если бы он был указателем на массив. Например, вы можете в Си итерировать массив, просматривая каждый элемент, следующим образом:

```
string array[ size ];
string *p = array;

for( int i = size; --i >= 0 ; )
   visit( *p++ );      // функции visit() передается строка.
```

Аналог в Си++ может выглядеть вот так (`keys` является деревом, чьи узлы имеют строковые ключи; здесь могут быть любые другие структуры данных):

```
tree<string> keys;      // двоичное дерево с узлами, имеющими
                        // строковые ключи
iterator p = keys;
// ...
for( int i = keys.size(); --i >= 0 ; )
   visit( *p++ );       // функции visit() передается строка.
```

Другими словами, вы обращаетесь с деревом как с массивом, и можете итерировать его при помощи итератора, действующего как указатель на элемент. И так как `iterator(p)` ведет себя точно как указатель в Си, то правило "без сюрпризов" не нарушается.

147. Перегрузив одну операцию, вы должны перегрузить все сходные с ней операции

Это правило является продолжением предыдущего. После того, как вы сказали, что "итератор работает во всем подобно указателю", он на самом деле должен так работать. Пример в предыдущем правиле использовал лишь перегруженные `*` и `++`, но моя настоящая реализация итератора делает аналогию полной, поддерживая все операции с указателями. Таблица 4 показывает различные возможности (`t` является деревом, а `ti` — итератором для дерева). Обе операции `*++p` и `*p++` должны работать и т.д. В предыдущем примере я бы должен был также перегрузить в классе `tree` операции **`operator`**`[]` и (унарная) **`operator`**`*()` для того, чтобы аналогия дерева с массивом выдерживалась везде. Вы уловили эту мысль?

Таблица 4. Перегрузка операторов в итераторе

Операция	Описание
`ti = t;`	Возврат к началу последовательности
`--ti;`	Возврат к предыдущему элементу
`ti += i;`	Переместить вперед на `i` элементов
`ti -= i;`	Переместить назад на `i` элементов
`ti + i;` `ti - i;`	Присваивает итератору другой временной переменной значение с указанным смещением от `ti`
`ti[i];`	Элемент со смещением `i` от текущей позиции
`ti[-i];`	Элемент со смещением `-i` от текущей позиции
`t2 = ti;`	Скопировать позицию из одного итератора в другой
`t2 - ti;`	Расстояние между двумя элементами, адресуемыми различными итераторами

`ti->msg();`	Послать сообщение этому элементу
`(*ti).msg();`	Послать сообщение этому элементу

Одна из проблем здесь связана с операциями **`operator`**`==()` и **`operator`**`!=()`, которые при первом взгляде кажутся имеющими смысл в ситуациях, где другие операции сравнения бессмысленны. Например, вы можете использовать == для проверки двух окружностей на равенство, но означает ли равенство "одинаковые координаты и одинаковый радиус", или просто "одинаковый радиус"? Перегрузка других операций сравнения типа < или <= еще более сомнительна, потому что их значение не совсем очевидно. Лучше полностью избегать перегрузки операций, если есть какая-либо неясность в их значении.

148. Перегруженные операции должны работать точно так же, как они работают в Си

Главной новой проблемой здесь являются адресные типы `lvalue` и `rvalue`. Выражения типа `lvalue` легко описываются в терминах Си++: они являются просто ссылками. Компилятор Си, вычисляя выражение, выполняет операции по одной за раз в порядке, определяемом правилами сочетательности и старшинства операций. Каждый этап в вычислениях использует временную переменную, полученную при предыдущей операции. Некоторые операции генерируют "`rvalue`" — действительные объекты, на самом деле содержащие значение. Другие операции создают "`lvalue`" — ссылки на объекты. (Кстати, "`l`" и "`r`" используются потому, что в выражении `l=r` слева от `=` генерируется тип `lvalue`. Справа образуется тип `rvalue`).

Вы можете сократить эффект неожиданности для своего читателя, заставив свои перегруженные операции-функции работать тождественно их эквивалентам на Си в пределах того, что они могут. Далее описано, как работают операции Си и как имитировать их поведение:

- Операции присваивания (=, +=, -= и т.д.) и операции автоинкремента и автодекремента (++, --) требуют операндов типа `lvalue` для адресата — части, которая изменяется. Представьте ++ как эквивалент для `+=1`, чтобы понять, почему эта операция в той же категории, что и присваивание.

В перегруженных операциях функций-членов указатель **`this`** *на самом деле является* `lvalue`*, поэтому здесь не о чем беспокоиться. На глобальном уровне левый операнд перегруженной бинарной операции присваивания (и единственный операнд перегруженной унарной операции присваивания) должен быть ссылкой.*

- Все другие операции могут иметь операнды как типа `lvalue`, так и `rvalue`.

Используйте ссылку на объект типа **`const`** *для всех операндов. (Вы могли бы передавать операторы по значению, но обычно это менее эффективно).*

- Имена переменных составного типа (массивов) создают типы `rvalue` — временные переменные типа указателя на первый элемент, после инициализации на него и указывающие. Заметьте, что неверно представление о том, что вы не можете инкрементировать имя массива из-за того, что оно является константой. Вы не можете инкрементировать имя массива, потому что оно имеет тип `rvalue`, а все операции инкремента требуют операндов типа `lvalue`.
- Имена переменных несоставного типа дают `lvalue`.
- Операции `*`, `->` и `[]` генерируют `lvalue`, когда относятся к несоставной переменной, иначе они работают подобно именам составных переменных. Если `y` не является массивом, то `x->y` создает тип `lvalue`, который ссылается на этого поле данных. Если `y` — массив, то `x->y` генерирует тип `rvalue`, который ссылается на первую ячейку этого массива.

В Си++ перегруженные `*` *и* `[]` *должны возвращать ссылки на указанный объект. Операция* **`operator->`** *таинственна. Правила по существу заставляют вас использовать ее таким же образом, как вы делали бы это в Си. Операция* `->` *рассматривается как унарная с операндом слева от нее. Перегруженная функция должна возвращать указатель на что-нибудь, имеющее поля —* структуру, класс *или* объединение. *Компилятор будет затем использовать такое поле для получения* `lvalue` *или* `rvalue`. *Вы не можете перегрузить* `.` *(точку).*

- Все другие операнды генерируют тип `rvalue`.

Эквивалентные перегруженные операции должны возвращать объекты, а не ссылки или указатели.

149. Перегруженной бинарной операции лучше всего быть встроенным (inline) псевдонимом операции приведения типа

Это правило относится к числу тех, которые будут изменены с улучшением качества компиляторов. Рассмотрим следующее, простое для понимания дополнение к классу `string` из листинга 7 на странице 155:

```
class string
```

```
{
   enum special_ { special };
   string( special_ ) {};              // ничего не делает.
   // ...
public:
   const string operator+( const string &r ) const;
   // ...
};
//-------------------------------------------------------------
const string::operator+( const string &r ) const
{
   string tmp( special );          // создать пустой объект

   tmp.buf = new char[ strlen(buf) + strlen(r.buf) + 1 ];
   strcpy( tmp.buf, buf );
   strcat( tmp.buf, r.buf );
   return tmp;
}
```

Многие компиляторы, получив вышеуказанное, генерируют довольно неэффективный код. Объект `tmp` должен инициализироваться при вызове конструктора; здесь это не очень дорого, но обычно это ведет к значительно большим расходам. Конструктор копии должен быть вызван для выполнения оператора **`return`**, и сам объект также должен быть уничтожен.

Иногда вы можете улучшить такое поведение путем перегрузки встроенного псевдонима для операции приведения типа:

```
class string
{
   string(const char *left, const char *right );
public:
   const string string::operator+( const string &r ) const ;
};
//-------------------------------------------------------------
string::string(const char *left, const char *right )
{
   buf = new char[ strlen(left) + strlen(right) + 1 ];
   strcpy( buf, left );
   strcat( buf, right );
}
//-------------------------------------------------------------
inline const string::operator+( const string &r ) const
{
   return string(buf, r.buf);
}
```

Более эффективные компиляторы здесь на самом деле рассматривают следующее:

```
string s1, s2;
s1 + s2;
```

как если бы вы сказали следующее (вы не можете сделать этого сами, потому что `buf` является закрытым):

```
string(s1.buf, s2.buf)
```

Полезный результат заключается в устранении неявного вызова конструктора копии в операторе **`return`** в первом варианте реализации.

150. Не теряйте разум с операторами преобразования типов

151. Если можно, то делайте все преобразования типов с помощью конструкторов

Распространенной ошибкой среди начинающих программистов на Си++ является сумасбродство с преобразованием типов. Вы чувствуете, что должны обеспечить преобразование каждого системного типа в ваш новый класс и обратно. Это может привести к подобному коду:

```
class riches                          //  богачи
{
public:
    riches( const rags &r );
};

class rags                           //  оборванцы
{
public:
    operator riches( void );
};
```

Проблема заключается в том, что обе функции определяют преобразование из `rags` в `riches`. Следующий код генерирует "постоянную ошибку" (которая прерывает компиляцию), потому что компилятор не знает, использовать ли ему для преобразования `rags` в `riches` конструктор в классе `riches`, или перегруженную операцию в классе `rags`; конструктор и перегруженная операция утверждают, что выполнят эту работу:

```
rags horatio_alger;     //   Гораций Алгер

riches bill_gates = (riches) horatio_alger;  // Бил Гейтс
```

Эта проблема обычно не так очевидна. Например, если вы определите слишком много преобразований:

```
class some_class
{
public:
```

```
    operator int           (void);
    operator const char * (void);
};
```

то простой оператор, подобный:

```
some_class x;
cout << x;
```

не сработает. Проблема в том, что класс `stream` определяет те же два преобразования:

```
ostream &ostream::operator<<( int   x       );
ostream &ostream::operator<<( const char *s );
```

Так как имеется два варианта преобразований, то компилятор не знает, какой из них выбрать.

Лучше выполнять все преобразования типов при помощи конструкторов и определять минимально необходимый их набор. Например, если у вас есть преобразование из типа **`double`**, то вам не нужны **`int`**, **`long`** и так далее, потому что нормальные правила преобразования типов Си применяются компилятором при вызове вашего конструктора.

Часть 8ж. Управление памятью

152. Используйте new/delete вместо malloc()/free()

Нет гарантии, что оператор **new**() вызывает malloc() при запросе памяти для себя. Он может реализовывать свою собственную функцию управления памятью. Следовательно, возникает трудно обнаруживаемая ошибка при передаче функцией free() памяти, полученной при помощи **new** (и наоборот).

Избегайте неприятностей, используя всегда при работе с Си++ **new** и **delete**. Наряду с прочим, это означает, что вы не должны пользоваться strdup() или любой другой функцией, скрывающей вызов malloc().

153. Вся память, выделенная в конструкторе, должна быть освобождена в деструкторе

Невыполнение этого обычно приводит к ошибке, но я видел программу, где это делалось намеренно. Упомянутая программа на самом деле нарушала другое правило: "Не позволяй открытого доступа к закрытому классу". Функция-член не только возвращала внутренний указатель на память, выделенную **new**, но класс ожидал, что вызывающая функция передает этот указатель **delete**. Это плохая идея со всех сторон: получить при этом утечку памяти — значит легко отделаться.

С точки зрения поиска ошибок помогает близкое физическое расположение конструктора и деструктора рядом друг с другом в файле *.cpp*, чтобы сделать их заметнее при отладке.

154. Локальные перегрузки операторов new и delete опасны

Здесь основной проблемой является то, что операторы **new** и **delete**, определенные в виде членов класса, следуют другим правилам, чем перегруженные на глобальном уровне. Локальная перегрузка используется лишь тогда, когда вы размещаете единственный объект. Глобальная перегрузка используется вами всегда при размещении массива. Следовательно, этот код, скорее всего, не будет работать:

```
some_class *p = new some_class[1];  // вызывает глобальный
                                    // оператор new()
//...
delete p;           // вызывает some_class::operator delete()
```

Помните, что эти две строки могут быть в различных файлах.

Часть 8з. Шаблоны

Многие проблемы с шаблонами в действительности вызваны учебниками, которые обычно настолько упрощенно рассматривают шаблоны, что вы заканчиваете чтение, не получив и намека на то, как они должны использоваться. Этот раздел посвящен распространенным затруднениям, связанным с шаблонами.

155. Используйте встроенные шаблоны функций вместо параметризированных макросов

Приведенный ранее пример:

```
#define SQUARE(x) ((x) * (x))
```

где:

```
SQUARE(++x)
```

расширяется до:

```
((++x)*(++x))
```

инкрементируя `x` дважды. Вы не можете решить эту проблему в Си, а в Си++ можете. Простая встроенная функция работает вполне удовлетворительно, в таком виде:

```
inline int square( int x ){ return x * x; }
```

не давая побочного эффекта. Тем не менее, она допускает лишь целочисленные аргументы. Шаблон функции, который расширяется во множество перегруженных встроенных функций, является более общим решением:

```
template <class type>
inline type square( type x ){ return x * x; }
```

К несчастью, это срабатывает только в простых ситуациях. Следующий шаблон не может обработать вызов `max(10, 10L)`, потому что не совпадают типы аргументов:

```
template <class type>
inline type max( type x, type y ){ return (x > y) ? x : y; }
```

Для обработки `max(10, 10L)` вы должны использовать прототип, чтобы принудить к расширению по тому варианту `max()`, который может выполнить данную работу:

```
long max( long, long );
```

Прототип вызывает расширение шаблона. Компилятор с легкостью преобразует аргумент типа **int** в **long**, даже если ему не нужно делать это преобразование для расширения шаблона.

Заметьте, что я здесь рекомендую использование шаблонов только потому, что `square` является встроенной функцией. Если бы этого не было, то для того, чтобы такой механизм был жизнеспособным, пришлось бы генерировать слишком много кода.

156. Всегда знайте размер шаблона после его расширения

Большинство книг демонстрирует шаблоны типа простого контейнера массива, подобного показаному на листинге 13. Вы не можете использовать здесь наследование (скажем, с базовым классом `array`, от которого наследуется `int_array`). Проблема заключается в перегрузке операции **`operator`**`[]()`. Вы бы хотели, чтобы она была виртуальной функцией в базовом классе, замещенная затем в производном классе, но сигнатура версии производного класса должна отличаться от сигнатуры базового класса, чтобы все это заработало. Здесь определения функций должны отличаться лишь возвращаемыми типами: `int_array::operator[]()` должна возвращать ссылку на тип **`int`**, а `long_array::operator[]()` должна возвращать ссылку на тип **`long`**, и так далее. Так как время возврата не рассматривается как часть сигнатуры при выборе перегруженной функции, то реализация на основе наследования не жизнеспособна. Единственным решением является шаблон.

Листинг 13. Простой контейнер массива

```
 1  template <class type, int size >
 2  class array
 3  {
 4     type array[size];
 5  public:
 6     class out_of_bounds {}; // возбуждается исключение, если
 7                             // индекс за пределами массива
 8     type &operator[](int index);
 9  };
10
11   template <class type, int size >
12   inline type &array<type, size>::operator[](int index)
13   {
14      if( 0 <= index && index < size )
15         return array[ index ]
```

```
16      throw out_of_bounds;
17  }
```

Единственная причина осуществимости этого определения заключается в том, что функция-член является встроенной. Если бы этого не было, то вы могли бы получить значительное количество повторяющегося кода. Запомните, что везде далее происходит полное расширение шаблона, включая все функции-члены[♣]. Вследствие того, что каждое из следующих определений на самом деле создает разный тип, то вы должны расширить этот шаблон четыре раза, генерируя четыре идентичные функции **operator**[](), по одной для каждого расширения шаблона:

```
array<int,10>  ten_element_array;
array<int,11>  eleven_element_array;
array<int,12>  twelve_element_array;
array<int,13>  thirteen_element_array;
```

(то есть array<**int,**10>**::operator**[](), array<**int,**11>**::operator**[]() и так далее).

Вопрос состоит в том, как сократить до минимума дублирование кода. Что, если мы уберем размер за пределы шаблона, как на листинге 14? Предыдущие объявления теперь выглядят так:

```
array<int>  ten_element_array (10);
array<int>  eleven_element_array (11);
array<int>  twelve_element_array (12);
array<int>  thirteen_element_array (13);
```

Теперь у нас есть только одно определение класса (и один вариант **operator**[]()) с четырьмя объектами этого класса.

Листинг 14. Шаблон массива (второй проход)

```
 1  template <class type>
 2  class array
 3  {
 4     type *array;
 5     int size;
 6  public:
 7     virtual ~array( void );
 8     array( int size = 128 );
 9
10     class out_of_bounds {};  // возбуждается исключение, если
11                              // индекс за пределами массива
12     type &operator[](int index);
13  };
```

[♣] Функции из шаблонов генерируется, только если они используются в программе (по крайней мере, так должен поступать хороший компилятор). — *Ред.*

```
14
15  template <class type>
16  array<type>::array( int sz /*= 128*/ ): size(sz)
17                                    , array( new type[ sz ] )
18  {}
19
20  template <class type>
21  array<type>::~array( void )
22  {
23     delete [] array;
24  }
25
26  template <class type>
27  inline type &array<type>::operator[](int index)
28  {
29     if( 0 <= index && index < size )
30        return array[ index ]
31     throw out_of_bounds;
32  }
```

Главным недостатком этой второй реализации является то, что вы не можете объявить двухмерный массив. Определение на листинге 13 разрешает следующее:

```
array< array<int, 10>, 20> ar;
```

(20-элементный массив из 10-элементных массивов). Определение на листинге 14 устанавливает размер массива, используя конструктор, поэтому лучшее, что вы можете получить, это:

```
array< array<int> > ar2(20);
```

Внутренний `array<int>` создан с использованием конструктора по умолчанию, поэтому это 128-элементный массив; мы объявили 20-элементный массив из 128-элементных массивов.

Вы можете решить эту последнюю проблему при помощи наследования. Рассмотрим следующее определение производного класса:

```
template< class type, int size >
class sized_array : public array<type>
{
public:
   sized_array() : array<type>(size) {}
};
```

Здесь ничего нет, кроме единственной встроенной функции, поэтому это определение очень маленького класса. Оно совсем не будет увеличивать размер программы, вне зависимости от того, сколько раз будет расширен шаблон. Вы теперь можете записать:

```
sized_array< sized_array<int,10>, 20> ar3;
```

для того, чтобы получить 20-элементный массив из 10-элементных массивов.

157. Шаблоны классов должны обычно определять производные классы

158. Шаблоны не заменяют наследование; они его автоматизируют

Главное, что нужно запомнить о шаблонах классов, — это то, что они порождают много определений классов. Как и всякий раз, когда у вас есть множество сходных определений классов, идентичные функции должны быть соединены в общий базовый класс.

Во-первых, давайте взглянем на то, что не нужно делать. Класс `storable`, уже использованный мной, снова представляется хорошим примером. Сначала создадим объект `collection` для управления сохраняемыми объектами:

```
class collection
{
   storable *head;
public:
// ...
storable *find( const storable &a_match_of_this ) const;
};

storable *collection::find( const storable &a_match_of_this )
const
{
// Послать сообщение объекту начала списка, указывающее, что спи-
// сок просматривается на совпадение со значением a_match_of_this;

   return  head ? head->find( a_match_of_this )
                : NULL
                ;
}
```

Механизм поиска нужных объектов скрыт внутри класса `storable`. Вы можете изменить лежащую в основе структуру данных, поменяв определение `storable`, и эти изменения совсем не затронут реализацию класса `collection`.

Затем давайте реализуем класс `storable`, использующий простой связанный список в качестве лежащей в основе структуры данных:

```
class storable
{
   storable *next, *prev;
```

```
public:
   storable *find ( const storable &match_of_this ) const;
   storable *successor ( void ) const;
   virtual int operator== ( const storable &r ) const;
};

storable *storable::find( const storable &match_of_this ) const
{
// Возвращает указатель на первый элемент в списке (начиная с
// себя), имеющий тот же ключ, что и match_of_this. Обычно,
// объект-коллекция должен послать это сообщение объекту начала
// списка, указатель на который хранится в классе коллекции.

   storable *current = this;
   for( ; current; current = current->next )
      if( *current == match_of_this )   // найдено совпадение
         return current;
}

storable *storable::successor( void ) const
{
// Возвращает следующее значение в последовательности.
   return next;
}
```

Функция **operator**==() должна быть чисто виртуальной, потому что отсутствует возможность ее реализации на уровне класса `storable`. Реализация должна быть выполнена в производном классе[13] :

```
class storable_string : public storable
{
  string s;
public:
  virtual int operator==( const storable &r ) const;
// ...
};

virtual int operator==( const storable &r ) const
{
  storable_string *right = dynamic_cast<storable_string *>( &r
);

  return right ? (s == r.s) : NULL;
}
```

Я здесь использовал предложенный в ISO/ANSI Си++ безопасный механизм нисходящего приведения типов. `right` инициализируется значением `NULL`, если передаваемый объект (`r`) не относится к типу

[13] В действительности я бы использовал множественное наследование с участием класса `string`. Использованный здесь код имеет цель немного упростить пример.

`storable_string`. Например, он может принадлежать к некоторому другому классу, также являющемуся наследником `storable`.

Пока все идет хорошо. Теперь к проблемам, связанным с шаблонами. Кто-нибудь, не понимающий того, что делает, говорит: "Ребята, я могу исключить наследование и потребность в виртуальных функциях, используя шаблоны", а делает, вероятно, нечто подобное:

```
template <class t_key>
class storable
{
   storable *next, *prev;

   t_key key;

public:
// ...
   storable *find      ( const storable &match_me ) const;
   storable *successor ( void                     ) const;
   int       operator==( const storable &r        ) const;
};

template <class t_key>
int storable<t_key>::operator==( const storable<t_key> &r )
const
{
   return key == r.key ;
}

template <class t_key>
storable<t_key> *storable<t_key>::successor( void ) const
{
   return next;
}

template <class t_key>
storable *storable<t_key>::find( const storable<t_key>
                                        &match_me ) const
{
   storable<t_key> *current = this;
   for( ; current; current = current->next )
      if( *current == match_me )       // найдено совпадение
         return current;
}
```

Проблема здесь в непроизводительных затратах. Функции-члены шаблона класса сами являются шаблонами функций. Когда компилятор расширяет шаблон `storable`, он также расширяет варианты *всех* функций-членов этого шаблона[♣]. Хотя я их не показал, вероятно, в классе `storable`

[♣] См. предыдущее примечание к правилу 156. — *Ред.*

определено множество функций. Многие из этих функций будут похожи в том, что они *не используют* информацию о типе, передаваемую в шаблон. Это означает, что каждое расширение такой функции будет идентично по содержанию любому другому ее расширению. Из функций, которые не похожи на функцию `successor()`, большинство будут подобны `find()`, использующей информацию о типе, но которую легко изменить так, чтобы ее не использовать.

Вы можете решить эту проблему, используя механизм шаблонов для создания производного класса. Основываясь на предыдущей реализации, не использующей шаблоны, вы можете сделать следующее:

```
template <class t_key>
class storable_tem : public storable
{
   t_key key;
public:
   // Замещение базового класса
   virtual int operator==( const storable &r ) const;
   // ...
};

template <class t_key>
/* виртуальный */ int storable_tem<t_key>::operator==( const
storable &r ) const
{
   t_key *right = dynamic_cast<t_key *>( &r );

   return right ? (s == r.s) : NULL;
}
```

Выбрав другой путь, я сосредоточил в базовом классе все функции, которые не зависят от типа `key`. Затем я использовал механизм шаблонов для создания определения производного класса, реализующего только те функции, которым нужно знать тип `key`.

Полезным результатом является существенное сокращение размера кода. Механизм шаблонов может рассматриваться как средство автоматизации производства шаблонных производных классов.

Часть 8и. Исключения

159. Назначение исключений — не быть пойманными

Как правило, исключение должно быть возбуждено, если:

- Нет другого способа сообщить об ошибке (например, конструкторов, перегруженных операций и т.д.).
- Ошибка неисправимая (например, нехватка памяти).
- Ошибка настолько непонятная или неожиданная, что никому не придет в голову ее протестировать (например, `printf`).

Исключения были включены в язык для обработки ошибочных ситуаций, которые иначе не могут быть обработаны, таких, как ошибка, случающаяся в конструкторе или перегруженной операции. Без использования исключений единственным способом обнаружения ошибки в конструкторе будет передача этому объекту сообщения:

```
some_obj x;
if( x.is_invalid() )
// конструктор не выполнился.
```

что, по меньшей мере, неаккуратно. Перегруженные операции являют собой ту же проблему. Единственным способом, которым использованная в

```
x = a + b;
```

функция **operator**+() может сообщить об ошибке, является возврат неверного значения, которое будет скопировано в `x`. Вы могли бы затем написать:

```
if( x == INVALID )
// ...
```

или нечто подобное. Снова весьма неаккуратно. Исключения также полезны для обработки ошибок, которые обычно являются фатальными. Например, большинство программ просто вызовут `exit()`, если функция `malloc()` не выполнится. Все проверки типа:

```
if( !(p = malloc(size)) )
   fatal_error( E_NO_MEMORY );
```

бесполезны, если оператор **new** просто не возвратит значения, когда ему не хватит памяти. Так как **new** на самом деле возбуждает исключение (по сравнению с вызовом `exit()`), то вы можете перехватить это

исключение в тех редких случаях, когда вы можете что-то сделать в такой ситуации.

Также имеется и другая проблема. Одной из причин того, что комитет ISO/ANSI по Си++ требует, чтобы оператор **new** возбуждал исключение, если он не может выделить память, заключается в том, что кто-то провел исследование и обнаружил, что какая-то смехотворная доля ошибок времени выполнения в реальных программах вызвана людьми, не побеспокоившимися проверить, не вернула ли функция `malloc()` значение `NULL`. По причинам, обсуждаемым позже, я не думаю, что исключение должно быть использовано вместо возврата ошибки просто для защиты программистов от себя самих, но оно срабатывает с **new**, потому что эта ошибка обычно в любом случае неисправима. Лучшим примером может быть функция `printf()`. Большинство программистов на Си даже не знают, что `printf()` возвращает код ошибки. (Она возвращает количество выведенных символов, которое может быть равно `0`, если на диске нет места). Программисты, которые не знают о возврате ошибки, склонны ее игнорировать. А это не очень хорошо для программы, которая осуществляет запись в перенаправленный стандартный вывод, продолжать, как будто все в порядке, поэтому можно считать хорошей идеей возбудить здесь исключение.

Итак, что же плохого в исключениях? На самом деле существует две проблемы. Первой является читаемость. Вам будет тяжело меня убедить, что:

```
some_class obj;
try
{
   obj.f();
}
catch( some_class::error &r )
{
   // выполнить действие в случае ошибки
}
```

лучше читается, чем:

```
if( obj.f() == ERROR )
// выполнить действие в случае ошибки
```

В любом случае, если **try**-блок содержит более одного вызова функций, вы не сможете просто исправить ошибку, потому что вы не сможете узнать, где возникла ошибка.

Следующий пример демонстрирует вторую проблему. Класс `CFile`, реализующий основной ввод/вывод двоичных файлов, возбуждает исключение в случае переполнения диска при записи, чего легко добиться

на дискете. Более того, функция `write()` не возвращает никакого кода ошибки. Перехват исключения является единственным способом обнаружения ошибки. Вот пример того, как вы должны обнаруживать ошибку чтения:

```
char  data[128];
Cfile f( "some_file", CFile::modeWrite );

try
{
   f.Write( data, sizeof(data) );
}
catch( CFileException &r )
{
   if( r.m_cause == CfileException::diskFull )
   // что-то сделать
}
```

Имеется две проблемы. Первая явно связана с уродливостью этого кода. Я бы гораздо охотнее написал:

```
bytes_written = f.Write( data, sizeof(data));
if( bytes_written != sizeof(data) )
   // разобраться с этим
```

Вторая проблема одновременно более тонкая и более серьезная. Вы не сможете исправить эту ошибку. Во-первых, вы не знаете, сколько байтов было записано перед тем, как диск переполнился. Если `Write()` возвратила это число, то вы можете предложить пользователю сменить диск, удалить несколько ненужных файлов или сделать еще что-нибудь для освобождения места на диске. Вы не можете тут сделать это, потому что не знаете, какая часть буфера уже записана, поэтому вы не знаете, откуда начинать запись на новый диск.

Даже когда `Write()` возвратила количество записанных байтов, то вы все еще не можете исправить ошибку. Например, даже если функцию `CFile` переписать, как показано ниже, то она все равно не будет работать:

```
char  data[128];
CFile f( "some_file", CFile::modeWrite );

int bytes_written;

try
{
   bytes_written = f.Write( data, sizeof(data) );
}
catch( CFileException &r )
{
   if( r.m_cause == CFileException::diskFull )
   // что-то выполнить.
```

```
    // при этом переменная bytes_written содержит мусор.
}
```

Управление передается прямо откуда-то изнутри `Write()` в обработчик **`catch`** при возбуждении исключения, перескакивая через все операторы **`return`** внутри `Write()`, а также через оператор присваивания в вызывающейся функции; переменная `bytes_written` остается неинициализированной. Я думаю, что вы могли бы передать `Write()` указатель на переменную, которую она могла использовать для хранения числа записанных байтов перед тем, как выбросить исключение, но это не будет значительным улучшением. Лучшим решением будет отказ от возбуждения исключения и возврат или числа записанных байтов, или какого-то эквивалента индикатора ошибки.

Последней проблемой являются непроизводительные затраты. Обработка исключения вызывает очень большие непроизводительные затраты, выражающиеся в возрастании в несколько раз размера кода и времени выполнения. Это происходит даже в операционных системах типа Microsoft Windows NT, которые поддерживают обработку исключений на уровне операционной системы. Вы можете рассчитывать на 10-20% увеличение размера кода и падение скорости выполнения на несколько процентов при интенсивном использовании исключений.[14] Следовательно, исключения должны использоваться лишь тогда, когда непроизводительные затраты не берутся в расчет; обычно, при наличии возможности, предпочесть возврат ошибки.

160. По возможности возбуждайте объекты типа `error`

Листинг 15 показывает простую систему определений класса для возбуждения исключений. Я могу перехватить ошибки чтения или записи подобным образом:

```
try
{
    file f("name", "rw");
    buffer b;
    b = f.read();
    f.write( b );
}
catch( file::open_error &r )
{
    // Файл не существует или не может быть открыт.
}
```

[14] Я определил это для 32-разрядного компилятора Visual C++ Microsoft; другие компиляторы показывают или сравнимые результаты, или худшие.

```
catch( file::io_error &r )
{
    // Какая-то из неисправимых ошибок ввода/вывода.
}
```

Если меня волнует лишь то, что произошла ошибка определенного вида, и не волнует, какого конкретно, то я могу сделать так:

```
file f;

try
{
    buffer b;
    b = f.read()
    f.write( b );
}
catch( file::error &r )
{
    // ...
}
```

Листинг 15. Классы исключений

```
1  class file
2  {
3  public:
4      class error {};
5      class open_error : public error {};
6      class io_error : public error {};
7
8      // ...
9  }
```

Этот код работает, потому что объект `file::read_error` является объектом типа `file::error` (так как относится к производному классу). Вы всегда можете перехватить объект производного класса, используя ссылку или указатель базового класса.

Я мог бы также предложить другой класс, использующий тот же самый механизм:

```
class long_double
{
public:
   class error {};
   class didvide_by_zero : public error {};
   // ...
};
```

Так как классы error являются вложенными определениями, то именами на самом деле являются `file::error` и `long_double::error`, поэтому здесь нет конфликта имен.

Для упрощения сопровождения я всегда использую `error` в качестве своего базового класса для исключений. (Я не мог использовать производный класс, даже если здесь был бы возможен всего один вид ошибки). Таким образом, я знаю, что, имея возбуждающий исключение класс `some_class`, можно перехватить это исключение при помощи:

```
catch(some_class::error &r)
```

Эту ошибку искать не придется. Если применяется наследование, то я использую базовый класс `error` таким образом:

```
class employee
{
public:
   class error                                    {};
   class database_access_error : public error {};
};

class peon : public employee
{
   class error : public employee::error {};
   class aagh  : public error           {};
};
```

Этим способом исключение `aagh` может быть перехвачено как `peon::aagh`, `peon::error` или `employee::error`.

Нет смысла создавать класс глобального уровня `error`, от которого наследуются все локальные классы `error`, потому что для обработки этой ситуации вы можете использовать обработчик **`catch`**`(…)`.

161. Возбуждение исключений из конструктора ненадежно

Я начну этот раздел с замечания о том, что компиляторы, которые соответствуют рабочим документам комитета ISO/ANSI по Си++, не имеют большей части из рассматриваемых здесь проблем. Тем не менее, многие компиляторы (один из которых компилятор Microsoft) им не соответствуют.

Ошибки в конструкторах являются действительной проблемой Си++. Так как они не вызываются явно, то и не могут возвратить коды ошибок обычным путем. Задание для конструируемого объекта "неверного" значения в лучшем случае громоздко и иногда невозможно. Возбуждение исключения может быть здесь решением, но при этом нужно учесть множество вопросов. Рассмотрим следующий код:

```
class c
{
```

```
    class error {};
    int *pi;
public:
    c() { throw error(); }
    // ...
};

void f( void )
{
    try
    {
        c *cp = new c; // cp не инициализируется, если не
                       // выполняется конструктор
        // ...
        delete cp;  // эта строка в любом случае не выполнится.
    }
    catch( c::error &err )
    {
        printf ("Сбой конструктора\n");

        delete cp;     // Дефект: cp теперь содержит мусор
    }
}
```

Проблема состоит в том, что память, выделенная оператором **new**, никогда не освобождается. То есть, компилятор сначала выделяет память, затем вызывает конструктор, который возбуждает объект `error`. Затем управление передается прямо из конструктора в **catch**-блок. Код, которым возвращаемое значение оператора **new** присваивается `cp`, никогда не выполняется — управление просто перескакивает через него. Следовательно, отсутствует возможность освобождения памяти, потому что у вас нет соответствующего указателя. Чтение мной рабочих документов комитета ISO/ANSI по Си++ показало, что такое поведение некорректно — память должна освобождаться неявно. Тем не менее, многие компиляторы делают это неправильно.

Вот простой способ исправить эту сложную ситуацию (я поместил тело функции в определение класса лишь для того, чтобы сделать пример покороче):

```
class c
{
    int *pi;
public:
    c() { /*...*/ throw this; }

};

void f( void )
{
```

```
    try
    {
        c  *cp = NULL;
        cp = new c;

        c a_c_object();
    }
    catch( c *points_at_unconstructed_object )
    {
        if( !cp ) // если конструктор, вызванный посредством
                  // new, не выполняется
            delete points_at_unconstructed_object;
    }
}
```

Ситуация усложняется, когда некоторые объекты размещаются при помощи **new**, а другие — из динамической памяти. Вы должны сделать что-то похожее на следующее, чтобы понять, в чем дело:

```
void f( void )
{
   c *cp = NULL;  // cp должен быть объявлен снаружи try-блока,
                  // потому что try-блок образует область
                  // действия, поэтому cp не может быть
                  // доступным в catch-блоке будучи объявлен в
                  // try-блоке.

   try
   {
      c a_c_object;
      cp = new c;
   }
   catch( c *points_at_unconstructed_object )
   {
      if( !cp )   // если конструктор, вызванный посредством
                  // new, не выполняется
         delete points_at_unconstructed_object;
   }
}
```

Вы не можете решить эту проблему внутри конструктора, потому что для конструктора нет возможности узнать, получена ли инициализируемая им память от **new**, или из стека.

Во всех предыдущих примерах деструктор для сбойных объектов вызывается, даже если конструктор не выполнился и возбудил исключение. (Он вызывается или косвенно посредством оператора **delete**, или неявно при выходе объекта из области действия, даже если он покидает ее из-за возбуждения исключения).

Аналогично, вызов **delete** косвенно вызывает деструктор для этого объекта. Я сейчас вернусь к этой ситуации. Перед выходом из этого

деструктора незавершенный конструктор должен привести объект в исходное состояние перед тем, как сможет возбудить ошибку. С учетом предшествующего определения класса `c` следующий код будет работать при условии, что отсутствует ошибка до оператора **`new int`**`[128]` и **`new`** выполнен успешно:

```
c::c( )
{
   if( some_error() )
      throw error(this); // ДЕФЕКТ: pi не инициализирован.
   // ...
   pi = new int[128];    // ДЕФЕКТ: pi не инициализирован,
                         // если оператор new возбуждает
                         // исключение.
   // ...
   if( some_other_error() )
   {
      delete [] pi;      // Не забудьте сделать это.
      throw error(this); // Это возбуждение безопасно
   }
}

c::~c( )
{
   delete pi;
}
```

Запомните, что `pi` содержит мусор до своей инициализации оператором **`new`**. Если возбуждается исключение до вызова **`new`** или сам оператор **`new`** возбудит исключение, то тогда `pi` никогда не инициализируется. (Вероятно, оно не будет содержать `NULL`, а будет просто не инициализированно). Когда вызывается деструктор, то оператору **`delete`** передается это неопределенное значение. Решим проблему, инициализировав этот указатель безопасным значением до того, как что-либо испортится:

```
c::c( ) : pi(NULL) // инициализируется на случай, если оператор
                   // new даст сбой
{
   if( some_error() )
      throw error(this); // Это возбуждение теперь безопасно.
      // ...
   pi = new int[128];    // Сбой оператора new теперь безопасен.
   // ...
   if( some_other_error() )
   {
      delete [] pi;     // Не забудьте высвободить динамическую
                        // память.
      throw error(this); // Это возбуждение безопасно.
   }
```

```
}

c::~c( )
{
   if( pi )
      delete pi;
}
```

Следует помнить, что нужно освобождать успешно выделенную память, если исключение возбуждено после операции выделения, так, как было сделано ранее.

У вас есть возможность почистить предложенный выше код при его использовании с учетом моего совета из предыдущего правила о возбуждении исключения объекта `error` и скрытия всех сложностей в этом объекте. Однако определение этого класса получается значительно более сложным. Реализация в листинге 16 опирается на тот факт, что деструктор явно объявленного объекта должен вызываться при выходе из **`try`**-блока, перед выполнением **`catch`**-блока. Деструктор для объекта, полученного при помощи **`new`**, не будет вызван до тех пор, пока память не будет передана оператору **`delete`**, что происходит в сообщении `destroy()`, посланном из оператора **`catch`**. Следовательно, переменная `has_been_destroyed` будет содержать истину, если объект получен не при помощи **`new`**, и исключение возбуждено из конструктора, и ложь — если объект получен посредством **`new`**, потому что деструктор еще не вызван.

Конечно, вы можете вполне резонно заметить, что у меня нет причин проверять содержимое объекта, который по теории должен быть уничтожен. Здесь уже другая проблема. Некоторые компиляторы (в том числе компилятор Microsoft Visual C++ 2.2) вызывают деструктор после выполнения оператора **`catch`**, даже если объекты, определенные в **`try`**-блоке, недоступны из **`catch`**-блока. Следовательно, код из листинга 16 не будет работать после этих компиляторов. Вероятно, лучшие решение состояло бы в написании варианта **`operator new`**`()`, который мог бы надежно указывать, получена память из кучи или из стека.

Листинг 16. *except.cpp* — возбуждение исключения из конструктора

```
 1  class c
 2  {
 3  public:
 4     class error
 5     {
 6        c *p;   // NULL при успешном выполнении конструктора
 7     public:
 8        error( c *p_this );
 9        void destroy( void );
10     };
11
12  private:
13
14     unsigned has_been_destroyed : 1;
15     int *pi;
16
17  private: friend class error;
18     int been_destroyed( void );
19
20  public:
21     c() ;
22     ~c();
23
24  };
25  //=========================================================
26  c::error::error( c *p_this ) : p( p_this ) {}
27  //---------------------------------------------------------
28  void c::error::destroy( void )
29  {
30     if( p && !p->been_destroyed() )
31        delete p;
32  }
33  //=========================================================
34  c::c() : has_been_destroyed( 0 )
35  {
36     // ...
37     throw error(this);
38        // ...
39  }
40  //---------------------------------------------------------
41  c::~c()
42  {
43     // ...
44     has_beeb_destroyed = 1;
45  }
46  //---------------------------------------------------------
47  int c::been_destroyed( void )
48  {
49     return has_been_destroyed;
50  }
```

```
51 //=========================================================
52 void main( void )
53 {
54    try
55    {
56       c *cp = new c;
57       c a_c_object;
58
59       delete cp;
60    }
61    catch( c::error &err )
62    {
63       err.destroy(); // деструктор вызывается, только если
64    }                 // объект создан оператором new
65 }
```

Заключение

Вот так-то. Множество правил, которые я считаю полезными и которые, надеюсь, будут полезны и для вас. Конечно, многие из представленных здесь правил дискуссионны. Пожалуйста, я готов с вами о них поспорить. Несомненно, я не считаю себя каким-то законодателем в стиле Си++ и сам нарушаю многие из этих правил при случае; но я искренне верю, что следование этим правилам сделает меня лучшим программистом, и надеюсь, что вы их тоже оцените.

Я закончу вопросом. Сколько времени потребуется программисту на Си++ для того, чтобы заменить электрическую лампочку? Ответ — нисколько, а вы, кажется, все еще мыслите процедурно. Правильно спроектированный класс `электрическая_лампа` должен наследовать метод замены от базового класса `лампа`. Просто создайте объект производного класса и пошлите ему сообщение `заменить_себя()`.

Об авторе

Ален Голуб — программист, консультант и преподаватель, специализирующийся на Си++, объектно-ориентированном проектировании и операционных системах Microsoft. Он проводит семинары по приглашению частных фирм повсюду на территории США и преподает в филиалах Калифорнийского университета, расположенных в Беркли и Санта-Круз. Он также работает программистом и консультантом по объектно-ориентированному проектированию, используя Си и Си++ в операционных средах Microsoft Windows, Windows-95, Windows NT и UNIX.

М-р Голуб регулярно пишет для различных компьютерных журналов, включая *"Microsoft Systems Journal"*, *"Windows Tech Journal"* и изредка *"BYTE"*. Его популярная колонка "Сундучок с Си", публиковавшаяся в *"Dr.Dobb's Journal"* с 1983 по 1987 годы, стала для многих людей первым введением в Си. В число его книг входят *"Compiler Design in C"*, *"C+C++"* и *"The C Companion"*. М-р Голуб сочиняет музыку и имеет лицензию частного пилота.

Вы можете связаться с ним через Интернет по адресу *allen@holub.com* или через его фирму Software Engineering Consultants, P.O.Box 5679, Berkeley, CA 94705 (телефон и факс: (510) 540-7954).