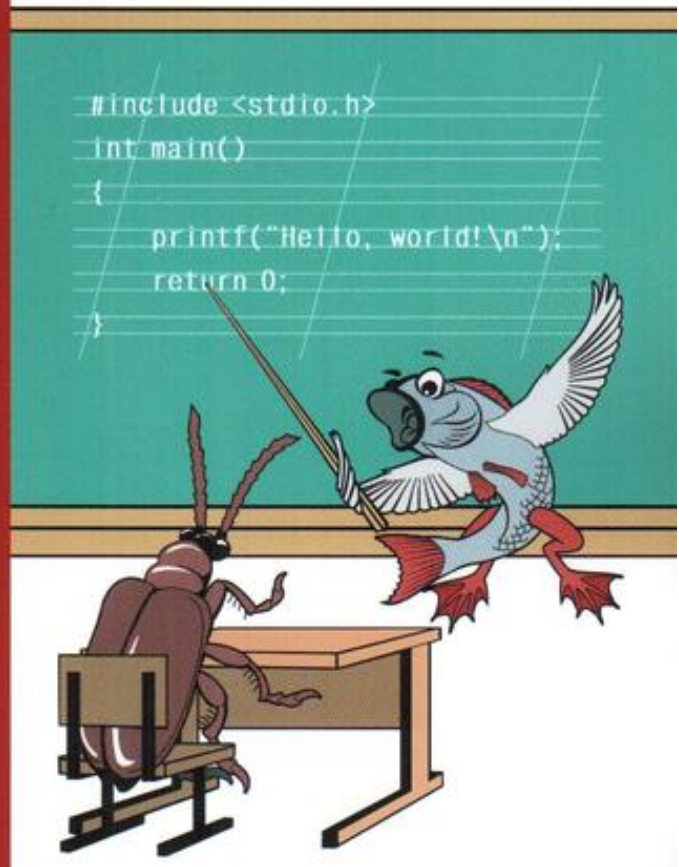
А. В. СТОЛЯРОВ



ОФОРМЛЕНИЕ ПРОГРАММНОГО КОДА

ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

Методическое пособие Андрея Викторовича Столярова «Оформление программного кода», опубликованное в издательстве МАКС Пресс в 2019 году, называемое далее «Произведением», защищено действующим российским и международным авторско-правовым законодательством. Все права на Произведение, предусмотренные законом, как имущественные, так и неимущественные, принадлежат его автору.

Настоящая Лицензия устанавливает способы использования электронной версии Произведения, право на которые предоставлено автором и правообладателем неограниченному кругу лиц, при условии безоговорочного принятия этими лицами всех условий данной Лицензии. Любое использование Произведения, не соответствующее условиям данной Лицензии, а равно и использование Произведения лицами, не согласными с условиями Лицензии, возможно только при наличии письменного разрешения автора и правообладателя, а при отсутствии такого разрешения является противозаконным и преследуется в рамках гражданского, административного и уголовного права.

Автор и правообладатель настоящим **разрешает** следующие виды использования данного файла, являющегося электронным представлением Произведения, без уведомления правообладателя и без выплаты авторского вознаграждения:

1. Воспроизведение Произведения (полностью или частично) на бумаге путём распечатки с помощью принтера в одном экземпляре для удовлетворения личных бытовых или учебных потребностей, без права передачи воспроизведённого экземпляра другим лицам;

2. Копирование и распространение данного файла в электронном виде, в том числе путём записи на физические носители и путём передачи по компьютерным сетям, с соблюдением следующих условий: (1) **все воспроизведённые и передаваемые любым лицам экземпляры файла являются точными копиями оригинального файла** в формате PDF, при копировании не производится никаких изъятий, сокращений, дополнений, искажений и любых других изменений, включая изменение формата представления файла; (2) **распространение и передача копий другим лицам производится исключительно *бесплатно*, то есть при передаче не взимается никакое вознаграждение ни в какой форме**, в том числе в форме просмотра рекламы, в форме платы за носитель или за сам акт копирования и передачи, даже если такая плата оказывается значительно меньше фактической стоимости или себестоимости носителя, акта копирования и т. п.

Любые другие способы распространения данного файла при отсутствии письменного разрешения автора запрещены. В частности, **запрещается**: внесение каких-либо изменений в данный файл, создание и распространение искаженных экземпляров, в том числе экземпляров, содержащих какую-либо часть произведения; распространение данного файла в Сети Интернет через веб-сайты, оказывающие платные услуги, через сайты коммерческих компаний и индивидуальных предпринимателей, а также **через сайты, содержащие рекламу любого рода**; продажа и обмен физических носителей, содержащих данный файл, даже если вознаграждение значительно меньше себестоимости носителя; включение данного файла в состав каких-либо информационных и иных продуктов; распространение данного файла в составе какой-либо платной услуги или в дополнение к такой услуге. С другой стороны, **разрешается** дарение (бесплатная передача) носителей, содержащих данный файл, бесплатная запись данного файла на носители, принадлежащие другим пользователям, распространение данного файла через бесплатные децентрализованные файлообменные P2P-сети и т. п. Ссылки на экземпляр файла, расположенный на официальном сайте автора, разрешены без ограничений.

А. В. Столяров *запрещает* Российскому авторскому обществу и любым другим организациям производить любого рода лицензирование любых его произведений и осуществлять в интересах автора какую бы то ни было иную связанную с авторскими правами деятельность без его письменного разрешения.

А. В. Столяров

ОФОРМЛЕНИЕ ПРОГРАММНОГО КОДА

издание второе

Москва – 2019

УДК 519.683+004.42
ББК 32.973.26-018.1
С81

Столяров А. В.
С81 Оформление программного кода. – 2-е изд., испр. и доп. – Москва: МАКС Пресс, 2019. – 116 с.
ISBN 978-5-317-06257-6

В пособии изложены основные принципы, применяющиеся для повышения читаемости текстов компьютерных программ и их доступности для анализа человеком; в частности, даются рекомендации по разбиению программ на модули и подсистемы, уделяется много внимания различным стилям расстановки структурных отступов и незначащих (декоративных) пробелов.

Для студентов программистских специальностей, преподавателей, программистов.

Ключевые слова: программирование, оформление кода, стиль кода, декомпозиция программ, модульное программирование.

УДК 519.683+004.42
ББК 32.973.26-018.1

Stolyarov, Andrey V.
Program code appearance: the guidelines. – MAKS Press, Moscow: 2019. – 116 p.
ISBN 978-5-317-06257-6

The book is devoted to basic principles used in computer programming to improve code readability, maintainability and clearness for persons other than the author. Special attention is paid to various styles of indentation and spacing; techniques of breaking a program down to modules and subsystems are discussed.

Intended for programmers, computer science and software engineering students, teachers.

Keywords: programming, coding style, coding convention, program decomposition, modules.

ISBN 978-5-317-06257-6

Оглавление

Предисловие

Начинающие программисты обычно полагают, что текст программы предназначен для компьютера; у опытных программистов на этот счёт иное мнение. Совсем не сложно написать программу так, чтобы её «понял» компилятор или интерпретатор, и при этом можно совершенно не задумываться о том, удачно ли выбраны имена переменных и подпрограмм, правильно ли программа разбита на строки, служат ли своей цели структурные отступы, — можно писать текст как попало, компилятору более-менее всё равно. Но такой подход годится лишь в случае, если программа, которую вы пишете, во-первых, настолько коротка, что вы её закончите в один приём, и, во-вторых, настолько бесполезна, что вы не только не станете её никому пересылать и даже показывать, но и сами никогда в жизни к ней не вернётесь. Вот только такие программы обычно не стоят того, чтобы вообще быть написанными.

Если же программа, которую вы вознамерились создать, претендует на то, чтобы быть кому-то (хотя бы даже вам самим) полезной, или даже не претендует, но вы собираетесь путём её написания чему-то научиться или что-то попробовать, то в подавляющем большинстве случаев программа будет, с одной стороны, достаточно сложна, чтобы написать её в один приём не получилось, и, с другой стороны, достаточно любопытна, чтобы у вас позже возникло желание вновь открыть и прочитать её текст. Так или иначе, оказывается чрезвычайно обидно тратить время на то, чтобы *разобраться в собственном коде.* Не следует переоценивать свои возможности по запоминанию внутреннего устройства написанных программ; реальность такова, что удержать в голове подробности реализации даже небольшой программы, занимающей 100–200 строк, оказывается человеку не под силу. Этот принцип срабатывает не только при возвращении к когда-то ранее написанной программе, но и непосредственно в процессе её написания: нет ничего проще (и обиднее), чем безнадёжно запутаться в тексте своей собственной программы, не успев толком ничего написать.

Дело резко осложняется, когда читать программу приходится кому-то кроме её автора. Разбираться в чужой программе — занятие сложное само по себе, даже если программа оформлена идеально с точки зрения понятности. Если же её автор не слишком утруждал

себя грамотным оформлением, то попытка понять что-то в такой программе превращается в сущий ад.

Так или иначе, следует с самого начала осознать один очень простой факт: **текст программы предназначен прежде всего для прочтения человеком, и лишь во вторую очередь — для обработки компьютером**. Практически все существующие языки программирования предоставляют автору исходного текста определённую свободу по части оформления кода, и эта свобода достаточна, чтобы сделать самую сложную программу понятной любому программисту, знакомому с используемым языком, но достаточна она и для того, чтобы сделать очень простую программу непонятной её собственному автору. Между прочим, существуют *международные соревнования*, кто напишет непонятнее. Наиболее известное из таких соревнований — Obfuscated C Code Contest; вот программа, победившая в одной из номинаций в 2011 году:

```
main(_,l)char**l;{6*putchar(--_%20?_+_/21&56>_?strchr(1[l],
_^"pt‘u}rxf~c{wk~zyHHOJ]QULGQ[Z"[_/2])?111:46:32:10)^_&&main(2+_,l);}
```

(автор — Такето Конно; исходно эта программа была написана в одну строчку). Заинтересованный читатель найдёт описание этой программы на странице `http://www.ioccc.org/2011/konno/hint.html`.

Ясно, что в большинстве случаев цель, преследуемая программистами, прямо противоположна. Можно в ряде случаев не заботиться о других читателях вашего кода, но уж себя-то, любимого, обижать не стоит. Как уже говорилось, если не уделять должного внимания оформлению, легко запутаться в собственной программе, не успев ещё ничего сделать; добавим к этому, что программирование — занятие чрезвычайно утомительное, требующее предельного напряжения интеллектуальных возможностей, так что мы не можем себе позволить пренебрежение имеющимися способами снижения нагрузки на мозг; как показывает многолетний опыт программистов всего мира, грамотное оформление программы снижает утомляемость даже не в разы, а *на порядок*. Имея дело с качественно оформленным кодом, вы можете проработать подряд 10–12 часов, не утратив способности к конструктивной деятельности, тогда как при работе с кодом, написанным «как попало», вы уже через каких-нибудь сорок минут или час почувствуете, что ваши мозги, как часто говорят, «вскипели» и работать дальше отказываются.

Хотелось бы отметить ещё один важнейший принцип. **Оформление программы должно быть правильным на всех этапах её написания, всегда, в любой момент с самого начала и до самого конца.** К сожалению, автору этих строк регулярно приходится видеть студентов, которые расстановку структурных отступов «оставляют на потом», как нечто не столь важное: пусть сначала программа заработает, а потом уж займёмся её «украшательством».

Так вот, это попросту *глупо*. «Потом», то есть когда программа уже написана, отступы, вообще говоря, *не нужны*, они играют свою роль не тогда, когда программа готова и осталось лишь сдать её преподавателю, а как раз тогда, когда программа ещё не написана: отступы позволяют видеть общую структуру кода, экономя интеллектуальные силы для более важных вещей, чем запоминание и удержание в голове структуры программы и её основных реперных точек, и это не говоря уже о том, что при грамотной расстановке отступов гораздо труднее становится сделать ошибку, забыв где-то открывающую или закрывающую скобку. Помните, что первым читателем вашей программы будете вы сами, а трудностей при её написании вам хватит и без того, чтобы заставлять самого себя разбираться в своём же коде.

Глава 1

Общие правила и принципы

1.1. Средства и цели

Сказанное в предисловии можно свести к следующим двум принципам, которые никогда не следует забывать в процессе написания программного кода.

1. **Программа в первую очередь предназначена для прочтения человеком, и лишь во вторую — для обработки компьютером.**

2. **Обеспечение правильного оформления — задача первоочередная; код, оформленный безграмотно, можно считать ненаписанным, даже если он успешно проходит трансляцию и, быть может, как-то работает.**

Завершая на этом разговор о *важности* грамотного оформления кода, мы перейдём к обсуждению того, *каким же* должно быть это оформление. Вводную главу мы посвятим формулировке и разъяснению универсальных правил, которые не зависят от применяемого языка программирования, а в последующих главах сделаем ряд уточнений, необходимых для конкретных языков.

Синтаксис большинства языков программирования достаточно гибок; мы можем разорвать строку в любом месте, где по смыслу положено ставить пробел, мы можем вместо одного пробела поставить их десять или сто, мы можем в любом месте кода вставить пустую строку; мы можем написать много комментариев, можем написать их немного, можем вообще не снисходить до комментирования. Довольно широки наши возможности при выборе идентификаторов —

имён для переменных, подпрограмм, макросов и других сущностей; в одной и той же ситуации разные программисты могли бы применить одно из следующих имён: `line_counter`, `LineCounter`, `linecounter`, `lines`, `LINES`, `lcnt`, `i_lncnt`, или даже просто `c`. Впрочем, находятся и такие программисты, которые в той же ситуации обзовут свою переменную `cccc`, `x273` или вовсе `abrakadabra` (автор лично видел переменную с таким именем в качестве счётчика строк, так что это, увы, не преувеличение).

Свобода, как водится, оказывается классической палкой о двух концах. Если разным людям дать совершенно одинаковые пистолеты, один воспользуется оружием для самообороны, другой — для стрельбы по пивным бутылкам или по тарелочкам, но обязательно найдётся и третий, который отстрелит себе ногу. Точно так же обстоят дела и с гибкостью синтаксиса в программировании: можно использовать эту гибкость, чтобы превратить программу в литературное произведение или хотя бы просто сделать её чтение если не приятным, то по крайней мере не слишком утомительным занятием; но можно, наоборот, запутать всё так, что даже сам автор программы на следующий день не сможет понять, как весь этот сумбур в действительности устроен.

Итак, наша цель — грамотно распорядиться имеющейся свободой, благо это не так сложно, и к тому же в нашем распоряжении оказывается весь опыт, накопленный программистами за несколько десятилетий существования программирования как вида человеческой деятельности. Сразу оговоримся, что предположения о том, что-де это или то «никогда не понадобится», как правило, слишком вредны, чтобы их допускать. Правило «никогда не говори “никогда”» выполняется в программировании едва ли не сильнее, чем во всех других областях. Если вы сочтёте, что ваша программа «никогда» больше вам не понадобится и сотрёте её, то по закону вселенской подлости вы уже через неделю будете об этом сожалеть, потому что однажды написанный код придётся написать ещё раз, потратив драгоценное время. То же самое произойдёт, если вы не стёрли программу, а «всего лишь» написали её как попало: когда она вам понадобится снова (а что это произойдёт, вы можете не сомневаться, сколь бы бесполезной она ни казалась), вам, скорее всего, не удастся в ней разобраться и придётся переписывать её заново.

Итак, ни в коем случае не предполагайте, что программа, которую вы пишете, никогда вам не потребуется в будущем. Точно так же не следует предполагать, что программу «никогда» не потребуется показать другому человеку; не следует даже предполагать, что программу «никогда» не потребуется показать человеку, который не знает русского языка (и мы абсолютно серьёзны).

Приняв решение быть осторожнее со словом «никогда», мы приходим к неизбежному выводу, что круг читателей программы, сколь бы простой и бесполезной вы её ни считали, может оказаться гораздо шире, нежели можно было бы предположить, и программу следует сделать понятной для всех её будущих читателей, сколько бы их ни было; вряд ли, конечно, программу станет читать человек, не знающий языка программирования, на котором она написана (хотя бывает и такое), но никаких других осмысленных ограничений мы a priori налагать не можем.

В следующих параграфах мы перечислим основные правила, которые сделают текст программы удобным в работе.

1.2. Самопоясняющий код

Конечно, комментарии могут облегчить понимание программы, но большинство программистов сходится в том, что злоупотреблять комментариями не следует. Это вполне можно понять, ведь при чтении обильно комментированной программы приходится фактически выполнять двойную работу: вникать в сам текст и в комментарии к нему. Сказанное не означает, что комментарии вообще не следует писать — напротив, во многих случаях без них не обойтись; но если код можно написать настолько ясным, чтобы комментариев к нему не требовалось — то именно так, разумеется, и следует поступить. Код, при написании которого гибкость синтаксиса языка программирования используется как инструмент объяснения устройства кода и используемых идей, называют *самопоясняющим* или *самодокументирующим*.

Можно выделить три основных инструмента самодокументирования программы — это выбор осмысленных идентификаторов, грамотное использование структурных отступов и разбиение задач на подзадачи.

1.2.1. Выбор имён (идентификаторов)

Общее правило при выборе имён достаточно очевидно: **идентификаторы следует выбирать в соответствии с тем, для чего они используются**. Некоторые авторы утверждают, что идентификаторы всегда обязаны быть осмысленными и состоять из нескольких слов. На самом деле это не всегда так: если переменная исполняет сугубо локальную задачу и её применение ограничено несколькими строчками программы, имя такой переменной вполне может состоять из одной буквы. В частности, целочисленную переменную, играющую роль переменной цикла, чаще всего называют просто «`i`»,

и в этом нет ничего плохого. Но однобуквенные переменные уместны только тогда, когда из контекста однозначно (и без дополнительных усилий на анализ кода) понятно, что это такое и зачем оно нужно, ну и ещё, пожалуй, разве что в тех редких случаях, когда переменная содержит некую физическую величину, традиционно обозначаемую именно такой буквой — например, температуру вполне можно хранить в переменной `t`, а пространственные координаты — в переменных `x`, `y` и `z`. Указатель можно назвать `p` или `ptr`, строку — `str`, переменную для временного хранения какого-то значения — `tmp`; переменную, значение которой будет результатом вычисления функции, часто называют `result` или просто `res`, для сумматора вполне подойдёт лаконичное `sum`, и так далее.

Важно понимать, что подобные лаконичности подходят лишь для локальных идентификаторов, то есть таких, область видимости которых ограничена одной подпрограммой, одним классом в объектно-ориентированных языках, в крайнем случае одним небольшим модулем (хотя это уже на грани фола). Если же идентификатор виден во всей программе, он просто обязан быть длинным и более чем осмысленным — хотя бы для того, чтобы не возникало конфликтов с идентификаторами из других подсистем. Чтобы понять, о чём идёт речь, представьте себе программу, над которой работают два программиста, и один из них имеет дело с температурным датчиком, а другой — с часами, измеряющими время; обе величины традиционно обозначаются `t`, но если наши программисты воспользуются этим обстоятельством для именования глобально видимых объектов, то проблемы нам не миновать: программа, в которой есть две разные глобальные переменные с одним и тем же именем, не имеет шансов пройти этап линковки.

Более того, когда речь идёт о глобально видимых идентификаторах, сама по себе длина и многословность ещё не гарантирует отсутствия проблем. Допустим, нам потребовалось написать функцию, которая опрашивает датчик температуры и возвращает полученное значение; если мы назовём её `get_temperature`, то формально вроде бы всё будет в порядке, на самом же деле с очень хорошей вероятностью нам в другой подсистеме потребуется узнать температуру, ранее записанную в файл или просто запомненную где-то в памяти программы, и для такого действия тоже вполне подойдёт идентификатор `get_temperature`. К сожалению, не существует универсального рецепта, как избежать таких конфликтов, но кое-что посоветовать всё же можно: **выбирая имя для глобально видимого объекта, подумайте, не могло бы такое имя обозначать что-то другое**. В рассматриваемом примере для идентификатора `get_temperature` можно с ходу предложить две-три альтернативные роли, так что его следует признать неудачным. Правильнее бу-

дет выбрать, например, идентификатор `scan_temperature_sensor`, но лишь в том случае, если он используется для работы со *всеми* температурными датчиками, с которыми имеет дело ваша программа — например, если такой датчик заведомо единственный, либо если функция `scan_temperature_sensor` получает на вход номер или другой идентификатор датчика. Если же ваша функция предназначена для измерения, к примеру, температуры в салоне автомобиля, причём существует ещё и датчик, скажем, температуры охлаждающей жидкости в двигателе, то в имя функции следует добавить ещё одно слово, чтобы полученное имя идентифицировало происходящее однозначно, например: `scan_cabin_temperature_sensor`.

Отметим ещё один момент, связанный с глобальными идентификаторами. Если в языке отсутствуют обособленные пространства имён (такие как `namespace` в Си++), во избежание возможных конфликтов имён все глобально-видимые идентификаторы, относящиеся к одной подсистеме (например, библиотеке) обычно снабжают общим префиксом, обозначающим эту подсистему. Например, все видимые имена библиотеки GNU Readline начинаются с «`rl_`»: `rl_gets`, `rl_complete` и т. п.

Некоторое время назад была достаточно популярна так называемая венгерская нотация для идентификаторов переменных; эта нотация требует, чтобы любой идентификатор переменной начинался с информации о типе этой переменной. Так, целочисленные переменные требуется всегда начинать с буквы `i`, а параметр функции, представляющий собой константную строку, оформленную в соответствии с соглашениями Си (то есть с нулём на конце) и задающую имя файла, придётся назвать примерно так: `lpczsFileName`. Уродливое lpczs расшифровывается как Long Pointer to Constant Zero-terminated String.

В действительности венгерская нотация превращает тексты программ в неудобочитаемый шифр. К счастью, она постепенно потеряла популярность даже в мире Windows, а за его пределами и вовсе никогда не использовалась, если не считать нескольких проектов 1970-х годов, достаточно крупных для своего времени, но ныне представляющих разве что исторический интерес. Тем не менее, мы считаем уместным предостеречь читателя от использования венгерской нотации. Возможно, вам попадутся её сторонники, которые начнут весьма убедительно расписывать достоинства именно такого именования идентификаторов; не обращайте на них внимания.

1.2.2. Структурные отступы: общие принципы

Структура любого фрагмента программы должна быть видна с первого взгляда, для чего принято использовать ***структурные отступы***. Практически все современные языки программирования позволяют поместить в начале любой строки текста *произвольное количество пробельных символов* — пробелов или табу-

ляций, что позволяет оформить программу так, чтобы её структуру можно было «схватить» даже расфокусированным взглядом, не вчитываясь.

Любопытным исключением из этого утверждения оказывается язык Python, который как раз не допускает произвольного количества пробелов в начале строки — напротив, в нём на уровне синтаксиса имеется жёсткое *требование* к количеству таких пробелов, соответствующее принципам оформления структурных отступов. Иначе говоря, большинство языков программирования *допускают* соблюдение структурных отступов, тогда как Python такого соблюдения *требует*. Сторонники обучения программированию на этом языке обычно относят такое жёсткое требование к достоинствам Python в роли учебного пособия; к сожалению, автор неоднократно наблюдал противоположный эффект — переходя с Python на какой-нибудь другой язык, начинающие программисты облегчённо вздыхают и «забивают» на отступы, ведь «здесь можно и без этого».

Структура программы формируется по принципу вложения одного в другое — например, одних операторов в другие операторы; техника структурных отступов позволяет высветить структуру программы, попросту *сдвигая вправо* любые вложенные конструкции относительно того, во что они вложены. К примеру, на Паскале заголовок программы, секции описаний констант, типов, переменных и меток, подпрограммы (процедуры и функции), а также главная программа *не вложены ни во что*, так что их все следует писать, начиная с первой позиции строки, не оставляя перед ними никаких пробельных символов. С другой стороны, каждое описание переменной *вложено* в секцию описаний, а каждый оператор главной программы *вложен* в саму главную программу, и т. д.; поэтому их следует сдвинуть вправо. На языке Си всё несколько проще: ни во что не вложены директивы макропроцессора (`#include`, `#define` и т. п.), описания функций (NB: в языке Си нет вложенных функций), описания глобальных переменных (вообще-то лучше не использовать глобальные переменные, но если они всё же появились, то их описания, разумеется, ни во что не вложены), и, как правило, описания типов — точнее, те из них, которые приводятся вне тел функций. Всё это мы пишем, начиная с первой позиции строки. В то же время локальные описания переменных (а иногда и типов), а также любые операторы, естественно, вложены в функции и должны быть соответствующим образом сдвинуты.

Пусть теперь в программе встречается тот или иной *сложный* оператор, то есть такой, в который в качестве составной части могут входить другие операторы. Можно не сомневаться, что такие встретятся: только самые простенькие учебные программы обходятся без ветвлений и циклов, ну а в ветвлениях и циклах телами выступают, в свою очередь, операторы. В этой ситуации операторы, представля-

ющие собою тела других операторов, должны быть сдвинуты вправо относительно своих «объемлющих» конструкций.

Поясним сказанное на примере; для этого рассмотрим программу, отыскивающую корень линейного уравнения вида $ax + b = 0$. На Паскале такая программа может выглядеть, например, так:

```
program linear_equation;
var
    a, b: real;
begin
    writeln('This program solves a*x+b=0');
    write('Please type a and b: ');
    read(a, b);
    if a = 0 then begin
        if b = 0 then
            writeln('True for any x')
        else
            writeln('No roots');
    end
    else
        writeln('x = ', -b / a : 5:5);
end.
```

Обратите внимание, что с крайней левой позиции строки мы начали писать заголовок (строку со словом `program`), секцию описаний (слово `var`) и главную программу. Описания переменных вложены в секцию описаний и должны быть сдвинуты вправо; мы сдвинули их на четыре пробела. Операторы главной программы, начиная с первого `writeln`, вложены в неё и также должны быть сдвинуты, что мы и сделали. Наконец, тела обеих веток оператора `if` вложены в этот `if` и, соответственно, сдвинуты ещё правее (уже на восемь пробелов от начала строки). В одной из веток у нас оказался ещё один оператор `if`, в который, в свою очередь, вложены два `writeln`'а; их *ранг вложенности* составляет 3, и для их сдвига применяется уже 12 пробелов.

Следует обратить внимание на то, что знак, *закрывающий* сложную конструкцию (в данном случае это ключевое слово `end`), должен быть написан в точности с такой же позиции по горизонтали, с которой начинается сама вложенная конструкция. Первый `end` в нашей программе закрывает оператор `if`, так что мы разместили букву `e` в слове `end` точно под буквой `i` в слове `if`. Второй `end` закрывает главную программу и написан в крайней левой позиции строки — точно так же, как и открывающий главную программу `begin`.

А теперь посмотрим, как могла бы выглядеть вышеприведённая программа, если бы мы не соблюдали структурные отступы:

```
program linear_equation; var a, b: real; begin writeln(
'This program solves a*x+b=0'); write('Please type a and b: '
); read(a, b); if a = 0 then begin if b = 0 then writeln(
'True for any x') else writeln('No roots'); end else writeln(
'x = ', -b / a :5:5); end.
```

Подчеркнём, что мы не изменили в программе ни одного слова. Более того, компилятору *абсолютно всё равно*, какой из этих двух текстов обрабатывать — результат получится совершенно одинаковый. Находятся даже люди, утверждающие, что программа не стала ничуть сложнее для понимания. Такие люди, как правило, никогда не работали с программами более чем на сотню строк. Программу в неотформатированном виде, безусловно, тоже можно прочитать, но это занимает в несколько раз больше времени и отнимает существенно больше сил; просто на совсем коротеньких программах это не так заметно.

Аналогичная программа на Си будет примерно такой:

```
#include <stdio.h>

int main()
{
    double a, b;
    printf("This program solves a*x+b=0.\n");
    printf("Please type a and b: ");
    scanf("%lf %lf", &a, &b);
    if (a == 0) {
        if (b == 0)
            printf("True for any x.\n");
        else
            printf("No roots.\n");
    }
    else
        printf("x = %5.5lf\n", -b / a);
    return 0;
}
```

Принципы её оформления — абсолютно те же.

Подробный разговор об отступах у нас впереди, и вообще им будет уделено самое пристальное внимание, поскольку именно структурные отступы оказываются мощнейшим инструментом повышения читаемости текста программы. В частности, мы узнаем, что стили расстановки отступов бывают разными, но далеко не все они допустимы, и научимся отличать допустимое от недопустимого. Пока же отметим несколько наиболее фундаментальных принципов.

Начнём с того, что *размер структурных отступов бывает разным* и может зависеть от личных предпочтений программиста или (чаще) руководителя проекта. В вышеприведённых примерах мы использовали четыре пробела, что соответствует размеру отступа, установленному по умолчанию в редакторе GNU Emacs. Во многих проектах (в частности, в ядре ОС Linux) для структурного отступа используют символ табуляции, причём ровно один. Можно встретить размер отступа в два пробела — именно такие отступы приняты в коде программ, выпускаемых Фондом свободного программного обеспечения (FSF). Совсем редко используется три пробела; такой размер отступа иногда встречается в программах, написанных для Windows. Другие размеры отступа использовать не следует, и этому есть ряд причин. Одного пробела слишком мало для визуального выделения блоков, левый край текста при этом начинает восприниматься как нечто плавное и не служит своей цели. Количество пробелов, превышающее четыре, трудно вводить: если их больше пяти, их приходится считать при вводе, что сильно замедляет работу, но и пять пробелов оказывается вводить очень неудобно (если угодно, попробуйте сами и убедитесь). Если же использовать больше одной табуляции, то на экран ничего не поместится: третий уровень вложенности окажется при этом на 48-й позиции, оставляя всего 32 знакоместа на содержательную часть строки, а четвёртый уровень, для которого потребуется 64-я позиция, вообще будет невозможно использовать.

Второй принцип структурных отступов состоит в их постоянстве в рамках одной программы. Вы можете выбрать, какой размер из допустимых (два пробела, три пробела, четыре пробела, табуляция) вам больше нравится, но выбранный вами размер отступов должен использоваться *во всей вашей программе*. Если вдруг вы по каким-то причинам решите изменить первоначальное решение относительно размера отступов, то следующим действием должно быть соответствующее изменение размера отступов во всей уже написанной части программы.

Наконец, последний универсальный принцип состоит в том, что **любая структура должна заканчиваться строкой, написанной с той же позиции по горизонтали, с которой написана строка, где эта структура началась**. Иногда это правило (ошибочно!) понимается в том смысле, что соответствующие друг другу *скобки* (слова `begin` и `end` в Паскале, фигурные скобки в языке Си и т. д.) должны быть размещены в одной колонке, но это неверно. Открывающую скобку часто оставляют на одной строке с заголовком (оператора, подпрограммы, структуры и т. п.; именно так мы поступили в вышеприведённых примерах), при этом она может оказаться сколь угодно далеко справа, но *закрывающая* скобка при этом *обяза-*

на находиться в той же колонке, где начался заголовок. Вот пример недопустимого размещения закрывающей скобки под открывающей:

```
if a = 0 then begin
    if b = 0 then
        writeln('True for any x')
    else
        writeln('No roots')
             end    { ТАК НЕЛЬЗЯ!!! }
```

Сравните этот пример с приведённой выше программой. Отметим, что следующий вариант считается вполне допустимым:

```
if a = 0 then
begin
    if b = 0 then
        writeln('True for any x')
    else
        writeln('No roots')
end
```

От предыдущих примеров этот вариант отличается тем, что слово `begin` (а для языка Си — открывающая фигурная скобка) размещается на отдельной строке. Обычно в таких случаях скобки не сдвигаются относительно заголовка, но и это правило не является абсолютным: иногда скобки тоже сдвигают. Такой вариант применяется крайне редко, но, тем не менее, также относится к числу допустимых[1]:

```
if a = 0 then
    begin
        if b = 0 then
            writeln('True for any x')
        else
            writeln('No roots')
    end
```

Здесь мы по-прежнему использовали отступ в четыре пробела, но обычно при такой расстановке операторных скобок возникает острый дефицит места по горизонтали, так что чаще всего при этом используют два пробела, а не четыре.

[1] Для тех, кто уверен в недопустимости этого варианта, заметим, что именно так оформлены исходные тексты Glibc и вообще практически всех программ, создаваемых командой GNU. Можно согласиться с мнением Торвальдса, что этот стиль не слишком красив, но называть его недопустимым всё же не стоит, поскольку проблемы, возлагаемые на структурные отступы, он в целом решает.

Важно помнить, что какой бы из стилей (разумеется, из *допустимых*) вы ни выбрали, его следует придерживаться на протяжении всего текста программы.

Отметим ещё один важный момент. **Ни в коем случае нельзя допускать смешивания пробелов и табуляций при построении отступов**. Если вы используете для отступов табуляцию, используйте *только* её; если вы используете пробелы — никогда не заменяйте их табуляциями, сколько бы их ни пришлось вколотить. Дело тут в том, что соотношение табуляции и пробела, вообще говоря, зависит от используемой операционной среды, текстового редактора, программы просмотра и т. д.; далеко не всегда и не везде табуляция соответствует восьми позициям.

Дело осложняется ещё и тем, что некоторые текстовые редакторы, якобы ориентированные на программистов, зачем-то по собственной инициативе заменяют восемь пробелов символом табуляции; в качестве примера таких своевольных редакторов можно назвать встроенный редактор среды Turbo Pascal (конечно, эта среда к нынешнему времени мертвее иных египетских мумий, но, к сожалению, в некоторых учебных заведениях она всё ещё в ходу), а из редакторов под ОС Unix — почему-то очень популярный среди студентов встроенный редактор оболочки Midnight Commander. Обычно замену пробелов табуляциями можно отключить; если это так — обязательно отключите её, если нет — используйте другой редактор. Отметим, что пользоваться встроенными редакторами так называемых интегрированных сред разработки в любом случае не стоит — есть много гораздо более удобных текстовых редакторов, среди которых можно выбрать такой, который понравится лично вам.

1.2.3. Ещё о пробелах

Структурные отступы — не единственный случай, когда в программу ради лучшей читаемости вставляются пробельные символы, не влияющие на её смысл. Так, структурно обособленные части программы (например, отдельные процедуры и функции) рекомендуется отделить друг от друга пустыми строками. Символы-разделители, которые не требуют пробелов для отделения их от других лексем, тем не менее часто обрамляют пробелами с обеих сторон или с одной стороны.

К вопросу о том, когда следует и когда не следует вставлять в текст программы «лишние» пробелы, мы ещё вернёмся, а пока отметим один очень важный момент: **никогда не следует в конце строки оставлять пробелы, которых «не видно»**. Такие пробелы часто появляются в тексте программы «сами собой», непреднамеренно: например, если вы набрали строку кода с пробелом между

отдельными её частями, а затем решили эту строку разорвать в том месте, где стоит пробел, поставили курсор на первый символ после пробела (тот символ, который станет первым символом новой строки) и нажали Enter, то эта — казалось бы, привычная, очевидная и безобидная — последовательность действий как раз и приведёт к тому, что на предыдущей строке в конце останется «невидимый» пробел, ведь символ перевода строки вы вставили *после* него.

Причина крайней нежелательности таких пробелов довольно проста. Если рассмотреть две строчки кода, единственным различием между которыми окажется наличие/отсутствие пробелов в конце строки, то *формально* (т. е., например, с точки зрения систем контроля версий) это будут две разные строки, но если эти две строки предъявить для сравнения человеку, он не сможет увидеть, в чём заключается разница между ними. Во многих случаях, возникающих в программистской практике, человек, изучающий различия двух версий одного и того же программного текста, *не может себе позволить* оставить такую ситуацию без внимания, а это значит, что ему придётся потратить время (иногда значительное) на выяснение, в чём же тут дело. Выяснить, что в конце одной из двух строк присутствуют «невидимые» пробелы, зачастую бывает непросто — так, если программист изучает отчёт об изменениях в коде, сгенерированный в виде html-страницы, то «невидимые» пробелы он сможет заметить не раньше, чем загрузит обе версии в текстовый редактор — а ведь можно и не догадаться это сделать.

1.2.4. Разбиение задач на подзадачи

Вернёмся к принципам написания саморазъясняющего кода. Последним из трёх перечисленных нами принципов будет применённый к программированию классический девиз «разделяй и властвуй». **Программу следует разбить на подпрограммы (процедуры, функции и т. п.) так, чтобы каждую из них можно было охватить одним взглядом и понять, как она устроена, не копаясь во всём остальном коде.** Это называется ***декомпозицией***.

Начинающие программисты (обычно школьники или студенты младших курсов) часто делают одну весьма серьёзную ошибку: пренебрегая подпрограммами, пытаются реализовать всю задачу в виде одного большого массива кода — главной программы на Паскале или функции `main` на Си. Когда размер такой программы переваливает за какую-нибудь сотню строк, код становится совершенно не поддающимся навигации; попросту говоря, при работе с такой программой (например, при необходимости что-то в ней исправить) больше

времени тратится на поиск нужного фрагмента, нежели на сами исправления.

Сразу же отметим, что разбивать программу на отдельные части *как попало* бессмысленно. Наша цель — снизить сложность программы, повысить её читаемость и понятность; но нарезать программу на куски можно так, что она станет ещё более запутанной, чем была.

Важнейшее правило декомпозиции состоит в том, что **каждая подпрограмма должна решать ровно одну задачу**, причём вы должны для себя сформулировать, какую конкретно задачу будет решать эта подпрограмма. **Ответ на вопрос «что делает эта подпрограмма» должен состоять из *одной фразы***, имеющей одну грамматическую основу, то есть всевозможные сложносочинённые и сложноподчинённые предложения здесь не годятся.

Кроме того, **каждая выделенная подзадача должна быть такой, чтобы при работе над вызывающей подпрограммой можно было не помнить детали реализации вызываемой, и наоборот, при работе над вызываемой — никак не учитывать детали реализации вызывающей.** Если это правило — так называемое ***требование обособленности*** — не выполняется, такое разбиение на подпрограммы сделает чтение текста программы труднее, а не проще, ведь при анализе кода придётся постоянно «прыгать» между телами двух подпрограмм, поскольку одну без другой уже не понять.

При проектировании подпрограмм важно следить за получающимися у вас списками параметров. Нарушение требования обособленности очень часто проявляется возникновением параметров, смысл которых невозможно объяснить, не прибегая к объяснениям принципа работы вызывающей подпрограммы. С учётом этого можно уточнить сформулированное выше правило: **ответ на вопрос, что делает данная подпрограмма, должен состоять из одной простой фразы, и из этой фразы должно быть хотя бы в первом приближении понятно, какова семантика всех параметров подпрограммы**.

Формируя список параметров, обратите внимание на их количество. **Подпрограмму, имеющую не более пяти параметров, использовать легко; подпрограмму с шестью параметрами использовать несколько затруднительно; подпрограммы с семью и более параметрами усложняют, а не облегчают работу с кодом**. Это обусловлено особенностями человеческого мозга. Удержать в памяти последовательность из пяти объектов нам достаточно просто, последовательность из шести объектов может удержать в памяти не каждый, ну а если объектов семь или больше, то удержать их в памяти единой картинкой попросту невозможно, приходится соответствующую последовательность *зазубривать*,

а потом тратить время и силы, чтобы вспомнить зазубренное. Пока ваша подпрограмма имеет не больше пяти параметров, вы, как правило, легко удержите в памяти их семантику (если это не так — скорее всего, подпрограмма неудачно спроектирована), так что сделать вызов такой подпрограммы окажется для вас легко. Если же параметров больше, то написание каждого вызова этой подпрограммы превратится в мучительное и не всегда успешное ковыряние в памяти или, что более вероятно, в тексте программы; такие вещи неизбежно отвлекают программиста от текущей задачи, заставляя вспоминать несущественные детали кода, написанного ранее.

Пожалуй, самое простое из правил грамотной декомпозиции состоит в ограничении длины каждой обособленной части. В идеале каждая подпрограмма (включая «главную» программу или функцию) должна быть настолько короткой, чтобы одного беглого взгляда на неё было достаточно для понимания её общей структуры. Остаётся только понять, *сколько* означает это «быть настолько короткой». Опыт показывает, что идеальная подпрограмма не должна превышать в длину 25 строк.

Число 25 возникло здесь не случайно. Традиционный размер экрана алфавитно-цифрового терминала составляет 25x80 или 24x80 (24 или 25 строк по 80 знакомест в строке), и считается, что подпрограмма должна целиком умещаться на такой экран, чтобы для её анализа не приходилось прибегать к скроллингу. Вполне возможно, что лично вы предпочитаете работать с редакторами текстов, использующими графический режим, и на вашем экране умещается гораздо больше, чем 25 строк; это в действительности никак не меняет ситуацию, потому что, во-первых, воспринимать текст существенно длиннее, нежели на 25 строк, тяжело, даже если его удалось поместить на экран; во-вторых, многие программисты, даже используя графический режим, предпочитают работать с крупными шрифтами, так что на *их* экран больше 25 строк всё-таки не влезет.

Лимит в 25 строк, вообще говоря, не вполне жёсткий. Так, если в вашей подпрограмме встретился оператор выбора (`case` для Паскаля, `switch` для Си, `cond` для Лиспа и т. п.), то вполне допустимо чуть превысить указанный размер: скажем, подпрограмма в 50 строк обычно криминалом не считается, хотя и не приветствуется. С другой стороны, если подпрограмма подбирается к длине в 60, а то и 70 строк, то её необходимо немедленно, не оставляя этого на абстрактное «потом», разбить на подзадачи. Если же подпрограмма перевалила за сотню строк, вам следует переосмыслить своё отношение к оформлению кода, поскольку при правильном подходе такого никогда не произойдёт.

Чаще всего нарушение перечисленных правил происходит из-за *поздней декомпозиции*: автор программы пишет какую-то её часть, не задумываясь о выносе подзадач в отдельные подпрограммы, и спохватывается, когда уже сложно что-то исправить. Стремясь до-

биться *формального* соответствия требованиям, он в подпрограмме, длина которой превысила приличия, выбирает некий *фрагмент кода* и выносит его в отдельную подпрограмму. Обычно при этом тут же выясняется, что этот фрагмент использует две-три локальные переменные, а то и все пять, и их приходится передать в новую подпрограмму через параметры, допускающие модификацию переменных вызываемым — var-параметры Паскаля, указатели в Си, ссылочные параметры в Си++ и т. п.; собственно говоря, обилие таких параметров как раз и свидетельствует о том, что в подпрограмму вынесена *не подзадача, а просто кусок кода.* Разумеется, такая подпрограмма не соответствует ни правилу одной фразы, ни требованию обособленности, часто имеет слишком много параметров и в целом выглядит довольно нелепо. Такое «вынесение кусков» — это вообще не декомпозиция, это *самообман*, ведь код всей программы при этом становится не легче, а *труднее* читать: раньше намертво связанные между собой фрагменты кода хотя бы располагались рядом, а теперь, чтобы их понять, нужно попеременно смотреть в разные места программы.

Сам факт появления длинной подпрограммы в вашем тексте показывает, что вы то ли вообще не задумываетесь о декомпозиции, то ли мысленно отмахиваетесь от мыслей о ней, считая, что прямо сейчас есть более важные дела. Вот только все эти «важные дела» потом, скорее всего, придётся переделывать заново. Когда у вас всё-таки дойдут руки до декомпозиции, вы с хорошей вероятностью обнаружите, что делать это уже поздно и подпрограмму придётся переписать с нуля, поскольку все её части намертво завязаны друг на друга и разделяться не желают. Если это так — переписывайте, не откладывая. Это всё равно придётся сделать, но чем позже — тем больше будет потрачено сил.

Этого кошмара можно легко избежать, если всегда следовать одному простому правилу: **если вы видите, что, написав некую дополнительную подпрограмму, можете упростить тот фрагмент, над которым работаете, то сразу же, не задумываясь, напишите её, даже если это потребует заметного времени;** время, вложенное в подготовку инструментов, потом окупится. Следует подчеркнуть, что потенциальная сложность дополнительных подпрограмм не должна вас останавливать. К примеру, если вы работаете над процедурой, длина которой составляет 20–30, и видите возможность сократить её длину на три строчки ценой написания процедуры в десять строк — сделайте это. Совокупный объём вашей программы при этом вырастет, но *сложность* — снизится.

Кроме того, всегда помните, что возникновения бардака проще не допустить, нежели потом его разгребать. Не позволяйте чрезмерно громоздким конструкциям пролезать в ваш текст. Например, **опера-**

тор выбора, вложенный в другой оператор выбора, практически никогда не следует считать допустимым. Если в вашем операторе выбора реализация всех или некоторых альтернатив оказалась настолько сложной, следует выделить каждую альтернативу в отдельную подпрограмму, а сам оператор выбора тогда будет состоять из их вызовов. Конечно, вложенные операторы выбора — это лишь пример ситуации, которой следует избегать; скажем, пять вложенных друг в друга циклов будут смотреться ещё хуже, притом намного, а при правильной и своевременной декомпозиции у вас, скорее всего, и трёх вложенных циклов никогда не образуется.

1.3. Универсально-читаемый код

Создавая текст программы, следует учитывать, что в мире существуют самые разные операционные системы и среды, программисты используют несколько десятков (если не сотен) редакторов текстов на любой вкус, а также всевозможные визуализаторы кода, функции которых могут сильно различаться. Далеко не все программисты говорят по-русски[2]; кроме того, для символов кириллицы существуют различные кодировки. Наконец, от одного рабочего места к другому могут существенно различаться размеры экрана и используемых шрифтов.

С чтением правильно оформленной программы не должно возникнуть проблем, какова бы ни была используемая операционная среда. Этого можно добиться, вооружившись всего тремя простыми правилами: символы из набора ASCII доступны всегда, английский язык знают все, а экран не бывает меньше, чем 24x80 знакомест, но *больше* он быть не обязан.

1.3.1. Алфавит ASCII — гарантия универсальности текста

Если использовать в тексте программы только символы из набора ASCII, можно быть уверенным, что этот текст успешно прочитается на любом компьютере мира, в любой операционной системе, с помощью любой программы, предназначенной для работы с текстом, и т. д. Напомним, что в этот набор входят:

- заглавные и строчные буквы **латинского** алфавита без диакритических знаков: `ABCDEFGHIJKLMNOPQRSTUVWXYZ`, `abcdefghijklmnopqrstuvwxyz`;

[2] Как уже отмечалось ранее, предположения со словом «никогда» делать вредно; это относится и к предположению о том, что вашу программу «никогда» не станут читать программисты из других стран.

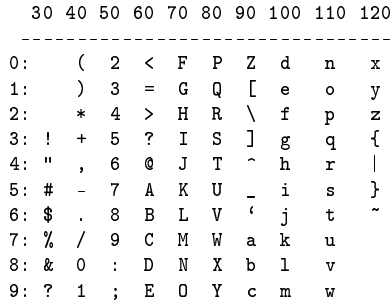


Рис. 1.1: Отображаемые символы ASCII

- арабские цифры: 0123456789;
- знаки арифметических действий, скобки и знаки препинания: .,;:'"`-?!@#$%^&()[]{}<>=+-*/~\|;
- знак подчёркивания _;
- пробельные символы — пробел, табуляция и перевод строки.

Никакие другие символы в этот набор не входят. В ASCII не нашлось места для символов национальных алфавитов, включая русскую кириллицу, а также для латинских букв с диакритическими знаками, таких как Š или Ä. Нет в этом наборе многих привычных нам типографских символов, таких как длинное тире, кавычки-ёлочки («») и т. п.

Символы, не входящие в набор ASCII, в тексте программ использовать нельзя — даже в комментариях, не говоря уже о строковых константах и тем более об идентификаторах. Большинство языков программирования не позволит использовать что попало в идентификаторах, но есть и такие трансляторы, которые считают символы, не входящие в ASCII-таблицу, допустимыми в идентификаторах — примером может служить большинство интерпретаторов Лиспа. Ну а на содержимое строковых констант и комментариев большинство трансляторов вообще не обращает никакого внимания, позволяя вставить туда практически что угодно. И тем не менее, попустительство трансляторов не должно сбивать нас с толку: текст программы обязан состоять из ASCII-символов, и только из них. Любой символ, не входящий в ASCII, может превратиться во что-то совершенно иное при переносе текста на другой компьютер, может просто не прочитаться и т. д.

Возникает естественный вопрос, как быть, если программа, которую вы пишете, должна общаться с пользователем по-русски. Ответ на этот вопрос мы дадим чуть позже.

1.3.2. Английский язык — не роскошь, а средство взаимопонимания

По сложившейся традиции программисты всего мира используют именно английский язык для общения между собой[3]. Как правило, можно предполагать, что любой человек, работающий с текстами компьютерных программ, поймёт хотя бы не очень сложный текст на английском языке, ведь именно на этом языке написана документация к разнообразным библиотекам, стандарты, описания сетевых протоколов, издано множество книг по программированию; конечно, многие книги и другие тексты переведены на русский (а равно и на французский, японский, венгерский, хинди и прочие национальные языки), но было бы неразумно ожидать, что *любой* нужный вам текст окажется доступен на русском — тогда как на английском доступна практически любая программистская информация.

Из этого вытекают три важных требования. Во-первых, **любые идентификаторы в программе должны состоять из английских слов или быть аббревиатурами английских слов**. Если вы забыли, как нужное вам слово перевести на английский, не поленитесь заглянуть в словарь. Подчеркнём, что слова должны быть именно английские — не немецкие, не французские, не латинские и тем более не русские «транслитом» (последнее вообще считается у профессионалов признаком крайне дурного тона). Во-вторых, **комментарии в программе должны быть написаны по-английски**; лучше вообще не писать комментарии, нежели пытаться писать их на языке, отличном от английского. И, наконец, **пользовательский интерфейс программы должен быть либо англоязычным, либо «международным» (то есть допускающим перевод на любой язык)**; этот момент мы подробно рассмотрим в следующем параграфе.

К настоящему моменту у некоторых читателей мог возникнуть закономерный вопрос — «а что делать, если я не знаю английского». Ответ будет тривиален, но он может вам не понравиться: в этом случае необходимо срочно начать интенсивное изучение английского, и никаких других вариантов тут не предложить. Программист, не умеющий более-менее грамотно писать по-английски (и тем более не понимающий английского), в современных условиях профессионально непригоден, сколь бы неприятно это ни звучало.

[3] Оставим в стороне вопрос о том, хорошо это или плохо, и ограничимся констатацией факта. Отметим, впрочем, что у врачей и фармацевтов всего мира есть традиция заполнять рецепты и некоторые другие медицинские документы на латыни, а, например, официальным языком Всемирного почтового союза является французский; так или иначе, существование единого профессионального языка общения оказывается во многом полезно.

1.3.3. О русскоязычном пользовательском интерфейсе

Пункт о языке пользовательского интерфейса нуждается в дополнительном комментарии, который послужит ответом на вопрос о том, как быть, если программа должна общаться с пользователем по-русски (или по-немецки, или по-китайски — это не важно).

Изоляционизм в программировании, к счастью, ушёл в далёкое прошлое, и в наши дни над одной программой могут работать программисты из нескольких десятков разных стран, а пользоваться одной и той же программой могут пользователи всего мира — во всяком случае, всех частей мира, где есть компьютеры. В такой обстановке постоянно возникает ситуация, когда программу необходимо «научить» общаться с конечным пользователем на таком языке, которого не знает никто из её авторов, причём эта ситуация в наше время представляет собой скорее правило, а не исключение. Адаптация программы к использованию другого языка оказывается достаточно простой, если её автор следовал определённым соглашениям; общая идея тут такова, что исходные строки, которые должен увидеть (или ввести) пользователь, прямо во время работы программы заменяются строками, написанными на другом языке, которые загружаются из внешнего файла (то есть *не являются частью программы*). В операционных системах семейства Unix обычно для этой цели используется библиотека `gettext`; в частности, при работе на языке Си, чтобы сделать возможной загрузку строк во время исполнения, все строки в программе обрамляются знаком подчёркивания и скобками — например, вместо

```
printf("Hello, world\n");
```

пишут

```
printf(_("Hello, world\n"));
```

Макрос с именем «`_`» разворачивается в вызов функции `gettext`, которая пытается найти файл с переводом сообщений на используемый язык интерфейса, а в нём, в свою очередь — перевод для строки `"Hello, world\n"`, причём сама эта строка используется как ключ для поиска. Если не удалось найти такой перевод (или сам файл с переводами), в качестве результата возвращается аргумент, то есть если `gettext` не знает, как перевести строку `"Hello, world\n"` на нужный язык, она так и оставит эту строку нетронутой.

Если вы по каким-то причинам не хотите использовать `gettext`, достаточно переопределить макрос `_`, чтобы он всегда возвращал свой аргумент; переделывать всю программу при этом не придётся.

Даже если вы не сочли нужным подготовить свою программу к использованию с библиотекой `gettext`, это может сделать за вас другой программист, просто заключив все строковые константы в макровызов `_()`, что достаточно просто — при известной ловкости это можно сделать одной командой в текстовом редакторе. Есть, однако, одно крайне важное условие, выполнение которого необходимо для превращения моноязычной программы в «международную»: все сообщения в её тексте должны быть английскими. Двойной

перевод системами интернационализации не предусмотрен, ну а переводчиков с *русского* на другие языки, будь то китайский или немецкий, найти гораздо сложнее, чем переводчиков с *английского* — особенно если учесть, что речь идёт не о профессиональных переводчиках, а, как правило, о программистах, которым пришло в голову перевести сообщения очередной программы на свой родной язык; английский знают программисты во всём мире, но вот найти нерусского программиста, при этом знающего русский, будет посложнее.

Как видим, программа, даже не адаптированная исходно под нужды многоязычности, вполне имеет шансы стать когда-нибудь «международной» — но только в том случае, если исходно все сообщения в её тексте английские. Из этого очевидным образом следует сформулированное выше утверждение: ваша программа должна быть либо «международной», либо англоязычной. Если вы не чувствуете себя в силах или не имеете желания учитывать возможный перевод интерфейса программы на другие языки, по крайней мере не лишайте *других* программистов возможности сделать это за вас. Если же русскоязычность является требованием, не поленитесь сделать всё *правильно* — то есть в соответствии с требованиями «международности».

1.3.4. Стандартный размер экрана

Необходимо учитывать существование такого понятия, как **стандартный размер алфавитно-цифрового экрана**, который **составляет 24 или 25 строк по 80 символов**.

Ширина текста в 80 символов стала своего рода традицией в области программирования. Происхождение числа 80 восходит ко временам перфокарт; перфокарты наиболее популярного формата, предложенного фирмой IBM, содержали 80 колонок для пробивания отверстий, причём при использовании перфокарт для представления текстовой информации каждая колонка задавала один символ. Одна перфокарта, таким образом, содержала строку текста до 80 символов длиной, и именно из таких строк состояли тексты компьютерных программ тех времён. Длина строки текста, равная 80 символам, ещё в начале 1990-х годов оставалась одним из стандартов для матричных принтеров. При появлении в начале 1970-х годов алфавитно-цифровых терминалов их ширина составила 80 знакомест, чтобы обеспечить «совместимость» двух принципиально различных способов ввода компьютерных программ. До сих пор многие компьютеры, оснащённые дисплеями, после включения питания начинают работу в текстовом режиме, и лишь после загрузки операционной системы переключаются в графический режим; ширина экрана в текстовом режиме в большинстве случаев составляет всё те же 80 знакомест.

Одним из традиционных способов управления компьютером остаётся командная строка (в особенности это верно для систем семейства Unix, но часто требуется и в мире Windows); чаще все-

го для этого используются графические программы, *эмулирующие* алфавитно-цифровой терминал. Ширина строки в таких программах составляет, как несложно догадаться, 80 символов, хотя это обычно легко исправить, просто изменив размеры окна.

Было бы неверно полагать, что число 80 здесь абсолютно случайно. Если ограничение на длину строк сделать существенно меньшим, писать программы станет неудобно, в особенности если речь идёт о структурированных языках, в которых необходимо использование структурных отступов. Так, в 40 символов на строку не уложились бы даже простенькие примеры программ, приведённые на стр. 14–15. С другой стороны, программы с существенно более длинными строками оказывается тяжело читать, даже если соответствующие строки помещаются на экране или листе бумаги. Причина здесь сугубо эргономическая и связана с необходимостью постоянно переводить взгляд влево-вправо.

Если вы возьмёте в руки любую книгу типографского происхождения, напечатанную в одну колонку — любой эпохи, на любом языке, лишь бы письменность, использованная в книге, была алфавитной — и сосчитаете в любой выбранной наугад строке на любой странице составляющие эту строку символы, включая знаки препинания и пробелы, вы получите результат от 65 до 75; меньше 65 бывает довольно редко, но всё-таки бывает, а вот больше 75 знаков в строке типографской книги найти будет намного сложнее. Дело тут в том, что такую книгу было бы *неудобно читать*. Именно такая, а не какая-то иная предельная длина типографской строки обусловлена особенностями нашего зрения — соотношением угла зрения, за пределами которого изображение становится слишком нечётким для восприятия букв, и углового размера букв, которые мы ещё можем разобрать, не вглядываясь.

Традиция ограничивать длину строк текста 80 символами насчитывает столь долгую историю[4] именно потому, что число 80 оказалось удачным компромиссом между малой вместительностью коротких строк и неудобочитаемостью длинных. Во времена перфокарт первые несколько позиций строки — от четырёх до шести — обычно использовали под номер строки, после него ставили пробел, отделяющий номера от самих строк, и на строку текста программы оставалось 73–75 знаковых колонок.

При современном размере дисплеев, их графическом разрешении и возможности сидеть к ним близко без вреда для здоровья многие программисты не видят ничего плохого в редактировании текста при ширине окна, существенно превышающей 80 знакомест. С точки

[4] 80-колоночные перфокарты были предложены IBM ещё в тридцатые годы XX века — задолго до появления первых ЭВМ.

зрения эргономики такое решение не вполне удачно; целесообразно либо сделать шрифт крупнее, чтобы глаза меньше уставали, либо использовать ширину экрана для размещения нескольких окон, в некоторых из которых работает сеанс редактирования — это сделает более удобной навигацию в вашем коде, ведь код сложных программ обычно состоит из множества файлов, и вносить изменения часто приходится одновременно в несколько из них. Отметим, что многие оконные текстовые редакторы, ориентированные на программирование, такие как `geany`, `gedit`, `kate` и т. п., штатно показывают на экране линию правой границы — как раз на уровне 80-го знакоместа.

Существует достаточно много программистов, предпочитающих не распахивать окно текстового редактора шире, чем на 80 знакомест в строке; более того, многие программисты пользуются редакторами текстов, работающими в эмуляторе терминала, такими, как vim или emacs; оба редактора имеют графические версии, но не всем программистам эти версии нравятся. Довольно часто в процессе эксплуатации программы возникает потребность просматривать и даже редактировать исходные тексты на *удалённой* машине, качество связи с которой (либо политика безопасности которой) может не позволять использование графики, и в этой ситуации окно алфавитно-цифрового терминала становится единственным доступным инструментом. Существуют программные средства, предназначенные для работы с исходными текстами программ (например, выявляющие различия между двумя версиями одного и того же исходного текста), которые реализованы в предположении, что строки исходного текста не превышают 80 символов в длину.

Часто листинг программы бывает нужно напечатать на бумаге. Наличие длинных строк в такой ситуации поставит вас перед неприятным выбором. Можно заставить длинные строки умещаться на бумаге в одну строчку — либо уменьшив размер шрифта, либо используя более широкий лист бумаги или «пейзажную» ориентацию — но при этом бóльшая часть площади листа бумаги останется пустой, а читать такой листинг будет труднее; если строки обрезать, попросту отбросив несколько правых позиций, есть риск упустить что-то важное; наконец, если заставить строки автоматически переноситься, читаемость полученного бумажного листинга будет хуже, нежели читаемость исходного текста при его отображении на экране, что уже совсем никуда не годится.

Вывод из всего вышесказанного напрашивается довольно очевидный: каким бы текстовым редактором вы ни пользовались, не следует допускать появления в программе строк, длина которых превосходит 80 символов. В действительности желательно всегда укладываться в 75 символов, что позволяет уместить в 80 символов не только строку исходного кода, но и её номер (четыре символа на номер,

одно знакоместо на пробел между номером и собственно строкой, оставшиеся 75 позиций на текст). Это позволит комфортно работать с вашим текстом, например, программисту, использующему редактор vim с включённой нумерацией строк; из такого исходного кода можно будет сформировать красивый и легко читаемый листинг с пронумерованными строками. Как уже говорилось, редакторы текстов, ориентированные на программирование, обычно поддерживают изображение правой границы, и по умолчанию отображают эту границу именно после 80-го знакоместа в строке; не пренебрегайте этим.

Некоторые руководства по стилю оформления кода допускают «в исключительных случаях» превышать предел длины строки. Например, стиль оформления, установленный для ядра ОС Linux, категорически запрещает разносить на несколько строк текстовые сообщения, и для этого случая говорится, что лучше будет, если строка исходного текста «вылезет» за установленную границу. Причина такого запрета достаточно проста. Ядро Linux — программа крайне обширная, и ориентироваться в её исходных текстах довольно трудно. Часто в процессе эксплуатации возникает потребность узнать, какой именно фрагмент исходного текста стал причиной появления того или иного сообщения в системном журнале, и проще всего найти соответствующее место простым текстовым поиском, который, разумеется, не сработает, если сообщение, которое мы пытаемся найти, разнесено на несколько текстовых констант, находящихся на разных строках исходника.

Тем не менее, превышение допустимой длины строк остаётся нежелательным. В том же самом руководстве по оформлению кода для ядра Linux на этот счёт имеются дополнительные ограничения — так, за правой границей экрана не должно быть «ничего существенного», чтобы человек, бегло просматривающий программу и не видящий текста справа от границы, не пропустил в результате какое-то важное её свойство. Чтобы определить, криминален ваш случай или нет, может потребоваться достаточно серьёзный опыт. Поэтому наилучшим вариантом будет всё же считать требование соблюдения 80-символьной границы жестким, то есть не допускающим исключений; как показывает практика, с этим всегда можно справиться, удачно разбив выражение, сократив текстовое сообщение, уменьшив уровень вложенности путём вынесения частей алгоритма во вспомогательные подпрограммы.

Кроме стандартной ширины экрана, следует обратить внимание также и на его высоту. Как уже говорилось выше, подпрограммы следует делать достаточно небольшими, чтобы они помещались на экран по высоте; остаётся вопрос, какой следует предполагать эту вот «высоту экрана». Традиционный ответ на этот вопрос — 25 строк, хотя имеются и вариации (например, 24 строки). Предполагать, что экран будет больше, не следует; впрочем, как уже говорилось, длина подпрограммы в некоторых случаях «имеет право» слегка превышать высоту экрана, но не намного.

1.4. Модульность

1.4.1. О роли подсистем и модулей

Пока исходный текст программы состоит из нескольких десятков строк, его проще всего хранить в одном файле. С увеличением объёма программы, однако, работать с одним файлом становится всё труднее и труднее, и тому можно назвать несколько причин. Во-первых, длинный файл элементарно тяжело перелистывать. Во-вторых, как правило, программист в каждый момент времени работает только с небольшим фрагментом исходного кода, старательно выкидывая из головы остальные части программы, чтобы не отвлекаться, и в этом плане было бы лучше, чтобы фрагменты, не находящиеся в работе в настоящий момент, располагались бы где-нибудь подальше, то есть так, чтобы не попадаться на глаза программисту даже случайно. В-третьих, если программа разбита на отдельные файлы, в ней оказывается гораздо проще найти нужное место, подобно тому, как проще найти нужную бумагу в шкафу с офисными папками, нежели в большом ящике, набитом сваленными в беспорядке бумажками. Наконец, часто бывает так, что один и тот же фрагмент кода используется в разных программах — а ведь его, скорее всего, приходится время от времени редактировать (например, исправлять ошибки), и тут уже совершенно очевидно, что гораздо проще исправить файл в одном месте и скопировать (файл целиком) во все остальные проекты, чем исправлять один и тот же фрагмент, который вставлен в разные файлы.

Практически любой язык программирования (или как минимум любая практически применимая его реализация, как в случае Паскаля) поддерживает включение содержимого одного файла в другой файл во время трансляции; в языке Си это делают с помощью директивы `#include`, в большинстве реализаций Паскаля — директивой `{$I }`, в Лиспе — вызовом псевдофункции `load`, в Прологе для того же самого предусмотрен встроенный предикат `consult`, и так далее. Разбиение текста программы на отдельные файлы, соединяемые транслятором, снимает часть проблем, но, к сожалению, не все, поскольку такой набор файлов остаётся, как говорят программисты, *одной единицей трансляции* — иначе говоря, мы можем их транслировать только все вместе, за один приём. Если речь идёт об интерпретируемом исполнении, то другого выхода у нас в любом случае нет, но при использовании компиляции время получения готового исполняемого файла после внесения правок в исходный текст может быть существенно сокращено, и такое сокращение часто оказывается необходимым для продолжения работы. Современные компиляторы работают довольно быстро, но объёмы наиболее серьёзных программ

таковы, что их полная перекомпиляция может занять несколько часов, а иногда и несколько суток. Если после внесения любого, даже самого незначительного изменения в программу нам, чтобы посмотреть, что получилось, придётся ждать сутки (да и пару часов — этого уже будет достаточно) — работать станет совершенно невозможно. Более того, программисты практически всегда используют так называемые ***библиотеки*** — комплекты готовых подпрограмм, которые изменяются очень редко, так что постоянно тратить время на их перекомпиляцию было бы несколько глупо. Наконец, проблемы создают постоянно возникающие конфликты имён: чем больше объём кода, тем больше в нём требуется различных глобальных идентификаторов (как минимум, имён подпрограмм), растёт вероятность случайных совпадений, а сделать с этим при трансляции в один приём почти ничего нельзя.

Все эти проблемы позволяет решить техника ***раздельной компиляции***. Суть её в том, что программа создаётся в виде множества обособленных частей, каждая из которых транслируется отдельно. Такие части называются ***единицами трансляции*** или ***модулями***. Чаще всего в роли модулей выступают отдельные файлы. Обычно в виде обособленной единицы трансляции оформляют набор логически связанных между собой подпрограмм; в модуль также помещают и всё необходимое для их работы — например, глобальные переменные, если такие есть, а также всевозможные константы и прочее. Каждый модуль транслируется отдельно; в результате трансляции каждого из них получается ***объектный файл***, обычно имеющий суффикс «`.o`». Затем с помощью редактора связей из набора объектных файлов получают исполняемый файл.

Очень важным свойством модуля является наличие у него собственного ***пространства видимости имён***: при создании модуля мы можем решить, какие из вводимых имён будут видны из других модулей, а какие нет; говорят, что модуль ***экспортирует*** часть вводимых в нём имён. Часто бывает так, что модуль вводит несколько десятков, а иногда и сотен идентификаторов, но все они оказываются нужны только в нём самом, а из всей остальной программы требуются обращения лишь к одной-двум подпрограммам, и именно их имена модуль экспортирует. Это практически снимает проблему конфликтов имён: в разных модулях могут появляться метки с одинаковыми именами, и это никак нам не мешает, если только они не экспортируются. Технически это означает, что при трансляции исходного текста модуля в объектный код все идентификаторы, кроме экспортируемых, исчезают.

1.4.2. Модуль как архитектурная единица

При распределении кода программы по модулям следует помнить несколько правил.

Прежде всего, **все возможности одного модуля должны быть логически связаны между собой**. Когда программа состоит из двух-трёх модулей, остаётся возможность помнить, как именно распределены по модулям части программы, даже если такое распределение не подчинено никакой логике. Ситуация резко меняется, когда число модулей достигает хотя бы десятка; между тем, программы, состоящие из *сотен* модулей, представляют собой довольно частое явление, и, больше того, можно легко найти программы, в состав которых входят тысячи и даже десятки тысяч модулей. Ориентироваться в таком массиве кода можно только в том случае, если реализация программы не просто раскидана по модулям, но в соответствии с некоторой логикой разделена на подсистемы, каждая из которых состоит из одного или нескольких модулей.

Чтобы проверить, правильно ли вы проводите разбивку на модули, задайте себе по поводу каждого модуля (а также по поводу каждой подсистемы, состоящей из нескольких модулей) простой вопрос: *«За что конкретно отвечает этот модуль (эта подсистема)?»* Ответ должен, как водится, состоять из одной фразы; «правило одной фразы» вообще достаточно универсально, когда речь идёт о декомпозиции чего бы то ни было на какие бы то ни было части. Если дать такой ответ не получается, то, скорее всего, ваш принцип разбивки на модули нуждается в коррекции. В частности, если модуль отвечает не за *одну* задачу, а за *две*, притом не связанные между собой, логично будет рассмотреть вопрос о разбивке этого модуля на два.

1.4.3. Ослабление сцепленности модулей

При реализации одних модулей постоянно приходится использовать возможности, реализованные в других модулях; говорят, что реализация одного модуля *зависит* от существования другого, или что модули ***сцеплены*** между собой. Опыт показывает, что **чем слабее сцепленность модулей, то есть их зависимость друг от друга, тем эти модули полезнее, универсальнее и легче поддаются модификации**.

Сцепленность модулей может быть разной; в частности, если один модуль использует возможности другого, но второй никак не зависит от первого, говорят об ***односторонней зависимости***, тогда как если каждый из двух модулей написан в предположении о существовании второго, приходится говорить о ***взаимной зависи-***

мости. Кроме того, если модуль только вызывает подпрограммы из другого модуля, говорят о ***сцепленности по вызовам***, если же модуль обращается к глобальным переменным другого модуля, говорят о ***сцепленности по переменным***. Отличают также ***сцепленность по данным***, когда одну и ту же структуру в памяти используют два и более модуля; вообще говоря, такая сцепленность может возникнуть и без сцепленности по переменным — например, если одна из подпрограмм, входящих в модуль, возвращает указатель на структуру данных, принадлежащую модулю.

Опыт показывает, что односторонняя зависимость всегда лучше, нежели зависимость взаимная, а сцепленность по вызовам всегда предпочтительнее сцепленности по переменным. Особенной осторожности требует сцепленность по данным, которая часто становится источником неприятных ошибок. В программе, идеальной с точки зрения разбиения на модули, все зависимости между модулями — односторонние, глобальных переменных нет вообще, а для каждой структуры данных, размещённой в памяти, можно указать её владельца (модуль, который отвечает, например, за своевременное уничтожение этой структуры), причём пользуется каждой структурой данных только её владелец.

Практика вносит некоторые коррективы в «идеальные» требования. Часто возникает необходимость во взаимозависимых модулях. Конечно, остаётся возможность слить такие модули в один, и в некоторых редких случаях именно так и следует поступить, но не всегда. К примеру, при реализации многопользовательской игры мы могли бы выделить в один модуль общение с пользователем, а в другой — поддержку связи с другими экземплярами нашей программы, которые обслуживают других игроков; практически неизбежно такие модули будут вынуждены обращаться друг к другу, но поскольку каждый из них отвечает за свою (чётко сформулированную!) подзадачу, объединять их в один модуль не нужно — ясность программы от этого нисколько не выиграет. Можно сказать, что взаимной зависимости модулей следует по возможности избегать, но всерьёз бояться её возникновения не стоит — это не криминал.

Иначе обстоит дело со сцепленностью по переменным и по данным. Без глобальных переменных можно обойтись *всегда*; при этом глобальные переменные затрудняют понимание работы программы, ведь функционирование той или иной подпрограммы начинает зависеть не только от поданных ей на вход параметров, но и от текущих значений глобальных переменных, а обнаружить такую зависимость можно только путём внимательного просмотра текста подпрограммы. Во время отладки мы можем обнаружить, что некая подпрограмма работает неправильно из-за «странного» значения глобальной переменной, причём может остаться совершенно непонятно, кто

и когда занёс в неё это значение. Говорят, что *глобальные переменные накапливают состояние*. Такое накапливание состояния способно затруднить отладку, даже если глобальные переменные локализованы в своих модулях, но в этом случае мы хотя бы знаем, где искать причины странного поведения программы: один модуль — это ещё не вся программа. Если же глобальная переменная видна во всей программе (***экспортируется*** из своего модуля), то, во-первых, любое её изменение потенциально может нарушить работу любой из подсистем программы, и, во-вторых, изменения в неё может внести кто угодно, то есть приходится быть готовыми искать по всей программе причину любого сбоя.

Сцепленность по переменным имеет и другой негативный эффект: такие модули сложнее модифицировать. Представьте себе, что какой-то из ваших модулей должен «помнить» координаты некоего объекта в пространстве. Допустим, при его создании вы решили хранить обычные ортогональные (декартовы) координаты. Уже в процессе эксплуатации программы может выясниться, что удобнее хранить не декартовы, а полярные координаты; если модули общаются между собой только путём вызова подпрограмм, такая модификация никаких проблем не составит, но вот если переменные, в которых хранятся координаты, доступны из других модулей и активно ими используются, то о модификации, скорее всего, придётся забыть — переписывание всей программы может оказаться чрезмерно сложным. Кроме того, часто возникает такая ситуация, когда значения нескольких переменных как-то друг с другом связаны, так что при изменении одной переменной должна также измениться и другая (другие); в таких случаях говорят, что необходимо обеспечить ***целостность состояния***. Сцепленность по глобальным переменным лишает модуль возможности гарантировать такую целостность.

Можно назвать ещё одну причину, по которой глобальных переменных следует по возможности избегать. Всегда (всегда!) существует вероятность того, что объект, который в настоящее время в вашей программе один, потребуется «размножить». Например, если вы реализуете игру и в вашей реализации имеется игровое поле, то вам весьма и весьма вероятно в будущем могут понадобиться *два* игровых поля. Если ваша программа работает с базой данных, то можно (и нужно) предположить, что рано или поздно потребуется открыть одновременно две или больше таких баз данных (например, для изменения формата представления данных). Ряд примеров можно продолжать бесконечно. Если теперь предположить, что информация, критичная для работы с вашей базой данных (или игровым полем, или любым другим объектом) хранится в глобальной переменной и все подпрограммы завязаны на использование этой переменной, то

совершить «метапереход» от одного экземпляра объекта к нескольким у вас не получится.

Хуже всего обстоят дела со сцепленностью по динамическим структурам данных. Один из модулей может посчитать структуру данных ненужной и удалить её, тогда как в других модулях сохранятся указатели на эту структуру данных, которые будут попрежнему использоваться. Возникающие при этом ошибки практически невозможно локализовать. Поэтому следует строго придерживаться «правила одного владельца»: у каждой создаваемой динамической структуры данных должен быть «владелец» (подсистема, модуль, структура данных, а в объектно-ориентированных языках программирования — соответственно, объект), и притом только один. За время своего существования динамическая структура данных может в случае крайней необходимости поменять владельца (например, одна подсистема может создать структуру данных, а использовать её будет другая подсистема), но правило существования и единственности владельца должно соблюдаться неукоснительно. Использовать структуру данных имеет право либо сам владелец, либо кто-то, кого владелец вызвал; в этом последнем случае вызванный не вправе предполагать, что структура данных просуществует дольше, чем до возврата управления владельцу, и не должен, соответственно, запоминать какие-либо указатели на эту структуру данных или на её части.

Понятие «владельца» динамической структуры данных обычно не поддерживается средствами языка программирования и, следовательно, существует лишь в голове программиста. Если отношение «владения» не вполне очевидно из текста программы, обязательно напишите соответствующие комментарии.

Общий подход к сцепленности модулей можно сформулировать следующими краткими правилами:

- **избегайте возникновения взаимных (двунаправленных) зависимостей между модулями, если это не сложно, но не считайте их криминалом;**
- **избегайте использования глобальных переменных, покуда это возможно; применяйте их только в случае, если такое применение способно сэкономить по меньшей мере несколько дней работы (экономию нескольких часов работы поводом для введения глобальных переменных лучше не считать);**
- **избегайте сцепленности по данным, а если это невозможно, то неукоснительно соблюдайте правило одного владельца.**

1.4.4. Выделение модулей во внешние библиотеки

Под ***библиотекой*** в программировании обычно понимают некий набор более-менее универсального программного кода, решающего определённую задачу или логически связанную между собой группу частных подзадач и допускающего включение в программы, в которых такие подзадачи возникают.

Часто разработка библиотеки с самого начала рассматривается как отдельный самостоятельный проект, то есть её создатели исходят из предположения, что избранные ими подзадачи достойны универсального решения и предложенное решение «обязательно кому-нибудь пригодится». К сожалению, бывает и так, что библиотечное решение, на создание которого потрачено много сил и времени, так никому и не пригождается. Но возможен качественно иной подход к созданию библиотек, который гарантирует, что силы не пропадут даром; подход состоит в том, что изначально код будущей библиотеки представляет собой часть обычной программы и решает такие подзадачи, которые потребовались (*уже* потребовались!) в ходе работы над этой программой. Решение о формировании отдельной библиотеки на основе имеющихся модулей принимается уже в процессе работы.

Коль скоро такое решение принято, из имеющегося набора исходных файлов выделяется определённый набор модулей, которые перемещаются в другую директорию и исключаются из основного проекта; в остальные файлы проекта по необходимости вносятся изменения, требующиеся, чтобы соответствующие возможности брались не из «своих» модулей, а из внешней библиотеки (конкретика зависит от используемого языка программирования, а в некоторых случаях — и от используемых инструментов, сред, системы сборки и т. п.). С этого момента основной проект и библиотека живут самостоятельными жизнями.

При принятии решения о создании отдельной библиотеки на основе имеющихся модулей необходимо руководствоваться следующими соображениями. Прежде всего, задайте себе вопрос «какие задачи будет решать новая библиотека». Как обычно в таких случаях, ответ должен состоять из одной фразы.

Второй вопрос может оказаться несколько сложнее: какие модули должны быть выделены в библиотеку и какими возможностями, *не вошедшими* в библиотеку, эти модули будут пользоваться. Категорически недопустима ситуация, при которой создаваемая библиотека будет в каком бы то ни было смысле зависеть от возможностей, которые остаются в основной программе. Кроме того, нежелательна ситуация, при которой библиотека использует возможности другой библиотеки — неважно, вашей собственной или внешней. Многие

программисты считают такую ситуацию допустимой, но на практике она приводит к конфликтам версий и другим феерическим «приключениям» при попытках воспользоваться библиотекой, так что в итоге кто-то вполне может предпочесть решить ту же задачу с нуля, нежели чем пользоваться вашей библиотекой. Так или иначе, зависимость одной библиотеки от другой теоретически допустима, но вот перекрёстные зависимости двух и более библиотек друг от друга недопустимы ни в коем случае — вы, скорее всего, просто не сможете слинковать такой проект. Если модули двух и более библиотек обнаруживают взаимозависимости, их нужно объединить в одну библиотеку; помните только, что правило одной фразы при ответе на вопрос «что делает эта библиотека» никто не отменял.

Третий и последний вопрос, на который надо ответить при выделении модулей в библиотеку — это вопрос о том, можете ли вы себе представить программу, совершенно не похожую на ту, что вы сейчас пишете, но в которой, тем не менее, могут пригодиться возможности вашей новой библиотеки.

Если ответы на все вопросы получены, вы можете смело объявить часть ваших модулей отдельной библиотекой и позволить им жить своей независимой жизнью. Если же удовлетворительно ответить на вышеперечисленные вопросы не удалось, то вполне возможно, что вам поможет небольшая переделка имеющегося кода, в особенности интерфейсов подпрограмм. Например, ваши подпрограммы могут принимать на вход такие типы данных, которые пришли из другой библиотеки или из такого модуля вашей программы, который вы не хотите включать в библиотеку. Конечно, в таком виде ваши подпрограммы для выделения в библиотеку не годятся, но может получиться так, что их очень легко переделать на работу со стандартными типами данных вместо специфических для вашей программы.

В целом выделение части кода в самостоятельную библиотеку выгодно с двух точек зрения: во-первых, такой код можно повторно использовать в других программах, и, во-вторых, сцепленность модулей при этом резко падает, снижая общую сложность работы с вашей программой. Помните только, что библиотека должна удовлетворять определённым свойствам, которые перечислены выше; без соблюдения этих простых правил создание отдельной библиотеки может привести только к дополнительным сложностям.

В заключение разговора о библиотеках дадим ещё одну рекомендацию. **Не торопитесь делать вашу библиотеку динамически связываемой**. В современных условиях динамическое связывание не даёт практически никакого ощутимого выигрыша, при этом требуя изрядного дополнительного расхода времени и сил программистов. Так или иначе, сделать свою библиотеку динамической вы всегда успеете.

1.5. Что такое coding style и какие они бывают

Внимательный читатель мог заметить, что практически все приведённые выше рекомендации так или иначе оставляют определённую свободу действий. В особенности это заметно, когда речь идёт о расстановке структурных отступов: в самом деле, надо ли использовать для отступов табуляцию, или же лучше будет отступать на четыре пробела? Или на два? Или на три? Следует ли сносить на следующую строку открывающую операторную скобку после заголовка сложного оператора? А после заголовка подпрограммы? А как быть, если заголовок не уместился в одну строку? Ну и, как говорится, так далее.

С уверенностью здесь можно сказать только одно: **в рамках одной программы стиль оформления должен выдерживаться строго один и тот же**, так что, если вас пригласили присоединиться к уже существующему проекту, вам придётся считать «правильным» тот стиль, который принят в этом проекте, сколь бы некрасивым он вам ни показался. Но вот вопрос о том, какого стиля придерживаться в программе, написание которой начинается с нуля, оказывается не так прост, как мы могли бы ожидать.

Сторонники разных стилей часто довольно нетерпимо относятся к другим стилям; так, Линус Торвальдс в документе, описывающем стиль для ядра Linux, называет «еретическим движением» сторонников размера отступов, отличного от табуляции, а руководства по оформлению кода GNU предлагает — в качестве символического жеста — распечатать на принтере, после чего сжечь не читая.

В такой обстановке важно уметь провести черту между требованиями, под которыми есть рациональная основа, и требованиями, продиктованными личными пристрастиями. Споры сторонников разных стилей часто называют «священными войнами» (holy wars), подчёркивая их схожесть с бессмысленной враждой представителей различных религий. По-видимому, можно сказать, что допустимых стилей оформления программного кода существует достаточно много, и выбор конкретного стиля может зависеть от ваших личных вкусовых предпочтений, либо от вкусовых предпочтений руководителя разработки. Если речь идёт о выборе одного из нескольких *допустимых* стилей оформления, не стоит быть категоричным и отстаивать «свой» стиль до победного конца; так, автор этих строк в своё время достаточно безболезненно перешел от использования в качестве отступа двух пробелов к использованию четырёх, а также неоднократно правил программы, использующие табуляцию, придерживаясь при этом, естественно, основного стиля программы.

Сказанное не означает, что *любой* стиль имеет право на существование. Это категорически не так. Важно понимать разницу между допустимыми особенностями стиля и особенностями недопустимыми. Интересно, что даже наличие значительного числа программистов, использующих тот или иной стиль, не делает его допустимым. В следующих главах мы намеренно постараемся при изложении каждого ключевого момента показать наряду с допустимыми примерами также и примеры ошибочного подхода к оформлению; как читатель уже мог заметить, такие примеры мы помечаем соответствующим знаком на полях.

1.6. Эстетика кода

К сожалению, мы вынуждены признать, что всё вышеизложенное хотя и способствует созданию правильных программ, но, увы, ровным счётом ничего не гарантирует, и сейчас мы попытаемся объяснить, почему.

Начнём с примера из области, никак с программированием не связанной. Существует довольно много людей, всерьёз и плотно имеющих дело с верёвками и узлами — это моряки (как профессионалы, так и яхтсмены-любители), альпинисты и спелеологи, арбористы, промышленные альпинисты и многие другие. В мире придумано много сотен разнообразных достаточно хитрых узлов, предназначенных для совершенно разных ситуаций; реально люди, занимающиеся каким-то видом деятельности, используют пять-шесть узлов, а знать «на всякий случай» могут в два-три раза больше; но и этим пяти или шести совершенно необходимым узлам новичков приходится подолгу учить. При этом, не сговариваясь, инструкторы по совершенно разным дисциплинам, говорящие на разных языках и живущие на разных континентах, сходятся в одном: каждый, кого когда-либо всерьёз учили завязывать узлы, слышал, что *правильно завязанный узел всегда выглядит красиво.* Это никак не зависит от типа узла, от применяемой верёвки, от цели, с которой завязывают конкретный узел: если узел смотрится красиво с сугубо эстетической точки зрения, то он, как правило, завязан правильно, и наоборот, узел, выглядящий кривым, косым или просто неуклюжим, наверняка построен с ошибками и может в самый неподходящий момент подвести: развязаться, порваться или наоборот — затянуться намертво, так, что его потом не удастся развязать и придётся резать верёвку.

Никто не возьмётся формализовать, что́ же конкретно скрывается за этим выражением — «узел, выглядящий красиво»; но принцип, тем не менее, великолепно работает.

Программирование — дисциплина многократно более сложная, нежели вязание узлов, но эстетический принцип действует и здесь: правильно организованный код *выглядит красиво*. К сожалению, далеко не все программисты способны это увидеть: в отличие от эстетики верёвочных узлов, эстетика программного кода не просто неформализуема, она ещё и *неуловима*. Отличить красивый узел от некрасивого может практически любой человек, не лишённый базовых эстетических навыков; но чтобы увидеть красоту программной архитектуры, нужно для начала научиться видеть сразу всю программу как единое целое, представлять себе связи между её частями, различать при этом разные виды таких связей — и это будет лишь самое начало на пути к постижению подлинного изящества одних программ и утилитарной вульгарности других.

Принципы оформления кода, изложенные в этой книжке, соотносятся с подлинной эстетикой программных архитектур примерно так, как соотносится умение не заваливать горизонт, нажимая на спуск фотокамеры, с умением создавать фотографические шедевры мирового уровня. Это касается и возможности объяснить происходящее новичкам: что горизонт должен быть горизонтальным, объяснить легко, несложно рассказать и базовые законы построения композиции и сочетания цветов, но попробуйте объяснить, какие конкретно особенности превращают в шедевр, например, фотографии Гельмута Ньютона, да к тому же ещё и не все — ведь творческие неудачи бывали и у него. Это относится и к возможности *научить*: выдрессировать очередного начинающего фотографа на выдерживание горизонта в нужной позиции — вопрос пяти минут, а попробуйте составить программу обучения так, чтобы на выходе из нескольких сот учеников получился хотя бы один настоящий фотохудожник.

Точно так же обстоят дела с программированием: научить базовым принципам оформления кода не так просто, но можно, эта задача заведомо имеет решение; а вот хорошо чувствовать эстетику (или, напротив, неряшливость) создаваемой программы научить практически невозможно, этому каждый вынужден учиться сам. Едва ли не единственным «источником мудрости» здесь служит собственный негативный опыт, когда программист возвращается к когда-то давно написанному коду, чтобы изменить и усовершенствовать его, и обнаруживает, что код оказывает ему достойнейшее сопротивление, столь жёсткое, что программу оказывается проще переписать заново. При этом раз за разом подспудно формируется понимание, как надо и как не надо действовать, чтобы потом — через неделю, месяц, год, десять лет — не сожалеть о собственной безалаберности и стремлении написать «как-нибудь, лишь бы работало». Написание программы как попало действительно позволяет сэкономить немного времени здесь и сейчас, но в будущем способно аукнуться потерей

времени, превышающего достигнутую экономию зачастую в сотни раз, и хорошо если не в тысячи. Проблема тут в том, чтобы знать (или скорее чувствовать), *как надо*; судя по всему, «опыт, сын ошибок трудных» тут никаким обучением заменить нельзя.

К сожалению, противоположный вариант — отбить любые зачатки эстетического чувства в отношении построения программ — ни малейшей трудности не представляет, и многие педагоги делают со своими учениками именно это, не ведая при этом, что творят. Научить плохому гораздо проще, чем научить хорошему.

Окончательно вытравить эстетическое чутьё позволяют так называемые олимпиады по программированию, где для серьёзных достижений, помимо прочего, оказывается совершенно необходима банальная скорость вколачивания кода. При этом код, написанный во время соревнований, не имеет никакой самостоятельной ценности, никто и никогда не возвращается к нему ни через неделю, ни через годы; можно сказать, что на олимпиадах не существует никакого «потом» — того самого «потом», которое выступает практически единственным средством развития программистского вкуса. Напротив, любые «заморочки» из этой области на олимпиаде только мешают, снижая скорость и затягивая время решения задач. В итоге бывшему олимпиаднику увидеть эстетические достоинства программы столь же непросто, как завзятому курильщику оценить какой-нибудь тонкий аромат.

Пожалуй, здесь даже невозможно посоветовать, что делать и как быть. Разве что одну вещь сказать всё же можно: если одни программы кажутся вам красивыми, а другие — некрасивыми, постарайтесь развить этот аспект своего восприятия. Во многих случаях вы окажетесь на верном пути. Учтите только, что здесь совсем рядом пролегает путь совершенно ложный: бесцельные трюки, позёрство и нерациональное усложнение программного текста на ровном месте может приводить автора кода в восторг от собственной крутизны, но с искомой эстетикой ничего общего не имеет. Здесь действует довольно простой принцип: **если очевидно, как переписать фрагмент программы, чтобы он стал понятнее стороннему читателю, то именно так его и следует переписать**. Бессмысленное усложнение не имеет ничего общего с хорошим вкусом, только с самолюбованием. Любителям «извернуться как следует» мы можем адресовать хорошо известный англоязычный девиз **Keep It Simple, Stupid!**[5], который русскоговорящие программисты часто называют «принципом поцелуя», буквально переводя аббревиатуру «K.I.S.S.» на русский как одно слово.

[5]Приблизительно перевести это на русский можно фразой «Не усложняй, глупец!», но смысловых оттенков такой перевод не передаёт.

Глава 2

Процедурный код: Паскаль, Си, Си++

Языки программирования Паскаль, Си и Си++ имеют между собой много общего. При обсуждении оформления кода нас прежде всего интересуют следующие особенности, присущие всем этим языкам.

1. Перечисленные языки относятся к классу императивно-процедурных; это означает, во-первых, что алгоритм на этих языках записывается в виде последовательности действий, и, во-вторых, что во всех этих языках присутствуют *подпрограммы*, то есть обособленные алгоритмы, исполняющие то или иное (вообще говоря, сколь угодно сложное) действие, которое благодаря их существованию можно рассматривать как один шаг алгоритма.

2. Во всех перечисленных языках основной единицей записи алгоритма является *оператор*[1], причём операторы бывают простые и сложные; под сложным мы понимаем такой оператор, *в состав которого* входят другие операторы (один или больше).

[1] Исходный английский термин — «statement» — было бы правильнее перевести как «утверждение», тем более что словом «оператор» в математике обозначается *отображение*, т. е. некая функция, а не действие. История не сохранила имени того недальновидного переводчика, которому мы обязаны таким значением термина «оператор» в русском языке, а равно и всей той неразберихой, которая в итоге из этого получилась, в особенности в связи с появлением ключевого слова `operator` в Си++. Так или иначе, слово «оператор» в русскоязычной программистской лексике устоялось накрепко, но очень полезно помнить, что этот термин является переводом слова «statement», тогда как англоязычное «operator» правильнее переводить как «операция», хотя это и не передаёт точного смысла.

Вложенные операторы в свою очередь могут быть как простыми, так и сложными.

3. Все перечисленные языки включают в себя операторы ветвления (с необязательной `else`-частью), цикла с предусловием, цикла с постусловием и параметрического цикла, причём в большинстве случаев синтаксис предусматривает *ровно один оператор* в роли *тела* цикла или ветвления (единственное исключение здесь — паскалевский `repeat/until`).

4. Для составления тела цикла или ветвления из нескольких операторов используется так называемый ***составной оператор***, образуемый из последовательности операторов путём заключения её в ***операторные скобки***, в качестве которых в Паскале используются ключевые слова `begin` и `end`, а в Си и Си++ — фигурные скобки `{` и `}`.

5. Синтаксическая конструкция составного оператора также используется в качестве тела подпрограммы, а в Паскале — ещё и в качестве тела «главной программы».

Все перечисленные сходства Паскаля, Си и Си++ позволяют сформулировать правила оформления кода, общие для этих языков, что мы и попытаемся сделать в этой главе. Разумеется, при всех сходствах между рассматриваемыми языками имеются также довольно существенные различия, которым мы посвятим соответствующие параграфы в конце главы.

2.1. Заголовок и тело

Можно выделить две основные ситуации, в которых речь идёт о *заголовке* и *теле*: это, во-первых, определение подпрограммы (функции или паскалевской процедуры), и, во-вторых, любой сложный оператор. Пока мы для простоты картины будем считать, что заголовок у нас умещается на одну строку; что делать, если заголовок получился слишком длинным, мы обсудим в § 2.7.3.

Пожалуй, самый простой случай ситуации «заголовок-тело» мы имеем для оператора `while`; заметим, синтаксис этого оператора для Паскаля и Си отличается разве что тем, что Паскаль предполагает после условия слово `do`, а Си требует заключить условие в скобки, но в остальном эти языки предоставляют совершенно одинаковые операторы цикла с предусловием. Мы начинаем с рассмотрения именно этого оператора, как самого простого, но даже на его примере можно проиллюстрировать изрядное количество степеней свободы выбора стиля.

2.1.1. Оператор `while` и основные стили отступов

Начнём с тривиального случая, когда тело оператора `while` состоит из одного *простого* оператора:

```
while p^.data <> x do
    p := p^.next;
```

```
while (p->data != x)
    p = p->next;
```

Казалось бы, здесь с оформлением всё понятно; но и этот тривиальный случай оставляет возможность сделать достаточно характерную ошибку в оформлении текста. Даже в коде, написанном опытными программистами, можно встретить фрагмент, подобный вышеприведённому, написанный в одну строчку:

```
while (p->data != x) p = p->next;
```

Этот фрагмент нельзя назвать абсолютно недопустимым, но так лучше всё же не писать, и вот почему. Если в программе в одних случаях тело сложного оператора оставляют на одной строчке с заголовком, а в других сносят на следующую, то чтобы *увидеть тело при беглом просмотре*, придётся приложить дополнительное (пусть незначительное, но всё-таки ненулевое) усилие: выяснить, есть там что-то после заголовка или нет. Конечно, на это уйдёт доля секунды, не больше, но именно из таких вот долей секунды и складывается дополнительное утомление при чтении программ. Одного этого соображения уже достаточно, но есть и ещё одна причина. *При пошаговом выполнении программы в отладчике, как правило, минимальной единицей программного кода считается строка*, и это значит, что цикл, записанный в одну строчку, вы в отладчике сможете пройти только как один шаг. Фактически в некоторых случаях вы даже не сможете понять, выполнился ли цикл хотя бы один раз или нет! То же самое касается и условного оператора: чтобы понять во время отладки, выполнялось ли его тело или только вычислялось условное выражение, придётся задействовать какие-нибудь сторонние признаки, поскольку видно этого не будет. Вывод получается простой и однозначный: **никогда не оставляйте тело сложного оператора на одной строчке с заголовком**. В конце концов, запас строк во Вселенной ничем не ограничен.

Рассмотрим теперь более сложный пример, когда в теле цикла нужно выполнить два действия (или больше), так что требуется применение составного оператора. В большинстве случаев профессиональные программисты пишут примерно так:

```
while p <> nil do begin
    s := s + p^.data;
    p := p^.next
end;
```

```
while (p) {
    s += p->data;
    p = p->next;
}
```

Обратите внимание, что открывающая операторная скобка оставлена на одной строке с заголовком, тогда как закрывающая стоит в той же колонке, в которой начинается заголовок, то есть закрывающую операторную скобку мы в обоих случаях написали точно под буквой `w` слова `while`. Как мы уже упоминали во введении, открывающую операторную скобку вполне допустимо снести на следующую строчку:

```
while p <> nil do
begin
    s := s + p^.data;
    p := p^.next
end;
```

```
while (p)
{
    s += p->data;
    p = p->next;
}
```

Заметим, именно такой стиль чаще всего встречается в учебниках по Паскалю, но вот большинство программистов, пишущих на Си и Си++, такой стиль не любят.

В обоих приведённых выше случаях составной оператор не сдвигается относительно заголовка, сдвигу подвергается только его содержимое. Крайне редко можно встретить стиль, при котором сам составной оператор тоже сдвигается:

```
while p <> nil do
    begin
        s := s + p^.data;
        p := p^.next
    end;
```

```
while (p)
    {
        s += p->data;
        p = p->next;
    }
```

При этом фактическое наполнение тела цикла оказывается сдвинуто на удвоенный отступ, если же тело состоит из одного простого оператора, его по-прежнему сдвигают на один отступ. С формальной точки зрения это можно назвать логичным, поскольку составной оператор — это тоже оператор; но на практике такой двойной отступ никаких преимуществ не даёт, съедая при этом лишнее место по горизонтали. Тем не менее, как сказано выше[2], этот стиль допустим.

Рассмотрим теперь несколько вариантов **недопустимого** стиля. Мы уже упоминали, что в программах новичков часто встречается примерно такое:

```
while p <> nil do begin
    s := s + p^.data;
    p := p^.next
                end;
```

```
while (p) {
    s += p->data;
    p = p->next;
        }
```

Закрывающая операторная скобка при этом оказывается сдвинута на непонятное количество позиций, зависящее, например, от того, из скольки символов состоит запись условного выражения. На то, чтобы пресловутую скобку найти, уходит лишняя доля секунды —

[2] См. сноску на стр. 17.

с ходу её в тексте не видно, она «сливается с окружающим пейзажем». Свою основную задачу — сделать *очевидным* конец сложного оператора даже для расфокусированного взгляда — такое положение закрывающей операторной скобки не решает.

Автор пособия неоднократно встречал в студенческих работах и вот такой, извините за выражение, кошмар:

```
while p <> nil do begin
                       s := s + p^.data;
                       p := p^.next
                  end;
```

При таком «стиле» места на экране не хватит даже на три уровня вложенности, но это не самое важное; гораздо хуже то, что размер сдвига оказывается не постоянным, как это должно быть, а зависящим от ширины заголовка. Читать такую программу совершенно невозможно.

Категорически недопустим также и следующий вариант:

```
while p <> nil do begin
    s := s + p^.data;
    p := p^.next end;
```

```
while (p) {
    s += p->data;
    p = p->next; }
```

а равно и любые вариации на его тему; **для закрывающей операторной скобки обязательно должна быть отведена отдельная строка**, иначе, опять-таки, конец сложного оператора будет сложно увидеть. С другой стороны, **после открывающей операторной скобки в строке не должно быть ничего**, за исключением разве что комментариев, так что следующий пример тоже недопустим:

```
while p <> nil do
begin s := s + p^.data;
    p := p^.next
end;
```

```
while (p)
{   s += p->data;
    p = p->next;
}
```

Если три вышеприведённых примера взяты автором из программ, написанных студентами и другими новичками, то следующий «стиль» был, как это ни странно, обнаружен в исходном тексте программы, написанной профессиональным программистом и, более того, достаточно широко используемой. Тем не менее, он тоже недопустим:

```
while p <> nil do begin
    s := s + p^.data;
    p := p^.next
    end;
```

```
while (p) {
    s += p->data;
    p = p->next;
    }
```

Как уже, несомненно, догадался читатель, здесь закрывающая операторная скобка опять сливается с окружающим текстом и не даёт

возможности увидеть конец сложного оператора расфокусированным взглядом.

Возвращаясь к допустимым вариантам сочетания заголовка оператора `while` и составного оператора, используемого в качестве тела, мы можем заметить, что принципиально различных вариантов существует всего три:

1. открывающая операторная скобка пишется на одной строчке с заголовком, закрывающая размещается точно под началом заголовка, содержимое составного оператора сдвигается на один отступ относительно начала заголовка;

2. открывающая операторная скобка сносится на следующую строку после заголовка и ставится точно под началом заголовка, закрывающая пишется точно под ней (и под началом заголовка), содержимое составного оператора сдвигается на один отступ;

3. открывающая операторная скобка сносится на следующую строку после заголовка и сдвигается относительно заголовка на один отступ, закрывающая пишется точно под ней, содержимое составного оператора сдвигается ещё на один отступ (то есть на два отступа относительно заголовка и на один — относительно операторной скобки).

Выше мы привели примеры кода для каждого из этих вариантов. Как уже говорилось, вы можете выбрать для себя любой из них (хотя последний из перечисленных мы бы всё же не рекомендовали), важно лишь придерживаться одного стиля на протяжении всей программы. Отметим, что выбор одного из этих стилей влияет не только на оформление оператора `while`, но и на все остальные сложные операторы, телом которых служит составной оператор.

Отметим ещё один немаловажный момент. Исходно составной оператор предназначался для объединения нескольких операторов в один, но существует целый ряд случаев, в которых *рекомендуется* (а в некоторых вариантах стилей — даже *требуется*) обрамление составным оператором *одного* оператора, или даже использование составного оператора, не содержащего внутри ни одного оператора. В частности, **если телом сложного оператора является, в свою очередь, сложный оператор, в особенности `if` с веткой `else`, то в большинстве случаев его желательно «обернуть» в операторные скобки, даже если он один**; читаемость программы может при этом возрасти. Подчеркнём, что в большинстве случаев это именно рекомендация, а не требование, и, более того, вы можете в некоторых случаях следовать ей, а в некоторых — нет, в том числе и в рамках одной программы.

Что касается пустых операторных скобок, то это специфическая разновидность пустого оператора, которая хорошо заметна в коде, и это свойство можно использовать для повышения читаемости вашего текста. Как известно, в Паскале и в Си можно сделать пустой оператор, просто поставив символ «;» (точку с запятой), и когда, например, нужно пометить меткой конец составного оператора (для чего требуется пустой оператор), то именно так и поступают. Но **если телом сложного оператора (обычно цикла) должен стать пустой оператор, то лучше использовать пустые операторные скобки, нежели точку с запятой**. Дело в том, что точку с запятой очень легко не заметить, в особенности при беглом просмотре программы, и тогда вам может показаться, что оператор, следующий непосредственно за «пустотелым» циклом, является его телом, а на то, чтобы установить действительное положение вещей, уйдёт несколько лишних секунд умственного напряжения. Например, вместо

```
while wait(nil) <> -1 do
    ;
```

```
while (wait(NULL) != -1)
    ;
```

лучше будет написать

```
while wait(nil) <> -1 do
begin
end;
```

```
while (wait(NULL) != -1)
{}
```

Отметим, что пустое тело цикла часто оставляют на одной строке с заголовком. Это допустимо, хотя мы бы рекомендовали так не делать, особенно если это точка с запятой, а не пустые скобки. Если вы возьмёте себе за правило никогда не оставлять тело цикла в строке заголовка, даже когда оно пустое, случай «точка с запятой в конце заголовка» будет в вашей программе недопустимым, и это довольно удачно, поскольку в большинстве случаев такая точка с запятой оказывается поставлена *по ошибке*, а поскольку заметить её трудно, это приводит к неприятным ошибкам в работе программы. Заведомая недопустимость такой ситуации позволяет, увидев заголовок с точкой с запятой на конце (поверьте, хотя бы раз вы эту ошибку сделаете), немедленно и без раздумий приступить к исправлению ошибки.

2.1.2. Оператор `if` с веткой `else`

Как Паскаль, так и Си/Си++ допускают использование оператора ветвления как в полном варианте — с веткой `else`, так и в сокращённом — без неё. Если ветка `else` отсутствует, то оператор `if` оформляется абсолютно так же, как и оператор `while`, который мы подробно обсудили в предыдущем параграфе. Никакой свободы

нам не оставляет также и случай, когда обе ветки присутствуют, но состоят из одного простого оператора, то есть не требуют использования операторных скобок:

```
if p <> nil then
    res := p^.data;
else
    res := 0
```

```
if (p)
    res = p->data;
else
    res = 0;
```

Нужно только не забывать, что тело всегда следует размещать на отдельной строчке, и в случае оператора `if` это касается обеих ветвей. Так, следующие варианты недопустимы:

```
if p <> nil then res := p^.data
else res := 0;
```

```
if (p) res = p->data;
else res = 0;
```

Рассмотрим теперь случай, когда *обе ветви оператора `if` используют составной оператор*. В этой ситуации нужно прежде всего вспомнить, какой из трёх вариантов размещения открывающей операторной скобки, перечисленных в конце предыдущего параграфа, мы выбрали. Если наш выбор пал на второй или третий вариант (в обоих случаях открывающая операторная скобка сносится на следующую строку), то всё оказывается достаточно просто. Если мы решили не сдвигать скобки составного оператора, а сдвигать только его тело, `if` придётся оформить так:

```
if p <> nil then
begin
    flag := true;
    x := p^.data
end
else
begin
    new(p);
    p^.data := x
end
```

```
if (p)
{
    flag = 1;
    x = p->data;
}
else
{
    p = malloc(sizeof(*p));
    p->data = x;
}
```

Если же вы, невзирая на все наши старания, избрали вариант со сдвигом операторных скобок, то `if` будет выглядеть так:

```
if p <> nil then
    begin
        flag := true;
        x := p^.data
    end
else
    begin
        new(p);
        p^.data := x
    end
```

```
if (p)
    {
        flag = 1;
        x = p->data;
    }
else
    {
        p = malloc(sizeof(*p));
        p->data = x;
    }
```

В обоих случаях `else` вместе со скобками занимает три строки, что несколько многовато для разделителя. Именно поэтому эти варианты (второй и третий) не слишком популярны у профессионалов, хотя, несомненно, допустимы. Что касается варианта, при котором открывающая операторная скобка оставляется на одной строке с заголовком, то в этом случае для ветки `else` также остаётся два варианта, а именно: поместить слово `else` на одной строке с закрывающей операторной скобкой или же писать `else` с новой строки. Первый вариант выглядит так:

```
if p <> nil then begin
    flag := true;
    x = p^.data
end else begin
    new(p);
    p^.data := x
end
```

```
if (p) {
    flag = 1;
    x = p->data;
} else {
    p = malloc(sizeof(*p));
    p->data = x;
}
```

Надо отметить, что именно этот вариант наиболее популярен у профессиональных программистов, но и второй вариант вполне допустим (более того, можно найти аргументы в его поддержку):

```
if p <> nil then begin
    flag := true;
    x := p^.data
end
else begin
    new(p);
    p^.data := x
end
```

```
if (p) {
    flag = 1;
    x = p->data;
}
else {
    p = malloc(sizeof(*p));
    p->data = x;
}
```

Рассмотрим теперь случай, когда *только одна* ветка конструкции `if-else` требует использования составного оператора, тогда как вторая состоит из одного *простого* оператора. Если принято решение сносить открывающую операторную скобку на следующую строку (не важно, со сдвигом или без сдвига), то этот случай не является особым и не требует отдельного рассмотрения. В принципе можно было бы вообще не рассматривать этот случай в качестве специального, но, как читатель может без труда убедиться самостоятельно, результаты могут получиться несколько неэстетичные. В связи с этим можно дать следующую рекомендацию: **если одна из ветвей конструкции `if-else` представляет собой составной оператор, то для второй ветви следует также использовать составной оператор, даже если она состоит из одного простого оператора**. Следование этой рекомендации, вообще говоря, факультативно, хотя правила, принятые в конкретной группе разработчиков, могут превратить эту рекомендацию в требование, как

это сделано в уже неоднократно упомянутом выше сообществе разработчиков ядра ОС Linux.

2.1.3. Если заголовок слишком длинный

Всё сказанное выше относительно заголовка оператора и его тела сформулировано в предположении, что заголовок сам по себе (и с учётом его отступа) умещается в одну строку. Но что делать, если условное выражение в заголовке оператора `if` или цикла получилось таким длинным, что сакраментальных 80 знакомест не хватает?

Первое, что необходимо сделать в такой ситуации — подумать, нельзя ли это выражение подсократить. Во многих случаях длинные условные выражения возникают в силу недостатка опыта программиста; так, автор часто видел в студенческих программах проверку принадлежности символа к тому или иному множеству (например, множеству знаков препинания) через последовательность явно прописанных сравнений с каждым из элементов множества, что-то вроде:

```
if (a == ',' || a == '.' || a == ';' || a == ':' || a ...
```

Разумеется, в такой ситуации никакой проблемы с недостатком места в строке на самом деле нет, есть лишь проблема недостаточности воображения. Программист, миновавший стадию новичка, в такой ситуации опишет *функцию*, проверяющую на принадлежность заданного символа предопределённому множеству, а заголовок `if`'а будет содержать вызов этой функции:

```
if (is_punctuation(a)) {
```

Более опытный программист воспользуется готовой функцией `ispunct` из стандартной библиотеки, а ещё более опытный, возможно, заявит, что стандартная функция `ispunct` чрезмерно сложна, поскольку зависит от настроек локали в окружении, и вернётся к варианту со своей собственной функцией. В любом случае никакой проблемы с длиной заголовка тут нет.

К сожалению, не всегда все проблемы решаются так просто. Многострочные заголовки, как бы мы ни пытались их побороть, всё равно иногда в программе возникают. Однозначного ответа, как с ними поступить, к сожалению, нет; мы рассмотрим один из возможных вариантов, который нам представляется наиболее практичным и отвечающим поставленной задачи читаемости программы.

Итак, **если заголовок сложного оператора приходится разнести на несколько строк, то:**

1. **разбейте выражение в заголовке на несколько строк; предпочтительно разрывать строку по «операции верхнего уровня», это обычно логическая связка «и» либо «или»;**

2. **каждую последующую строку заголовка сдвиньте относительно первой строки заголовка на обычный размер отступа;**

3. **вне зависимости от количества простых операторов в теле обязательно возьмите тело вашего оператора в операторные скобки, то есть сделайте его составным оператором;**

4. **вне зависимости от используемого стиля снесите открывающую операторную скобку на следующую строку, чтобы она послужила зрительным разделителем между строками заголовка и строками тела вашего оператора.**

Всё вместе будет выглядеть примерно так:

```
while (TheCollection^.KnownSet^.First = nil) and
    (TheCollection^.ToParse^.First <> nil) and
    (TheCollection^.ToParse^.First^.s = ' ') do
begin
    SkipSpace(TheCollection)
end;
```

```
while (!the_collection->known_set->first &&
    the_collection->to_parse->first &&
    the_collection->to_parse->first->s == ' ')
{
    skip_space(the_collection);
}
```

Такой вариант нормально сработает, если вы не сдвигаете составной оператор относительно заголовка; если же вы предпочли именно этот («третий») стиль оформления, можно посоветовать снести на следующую строку последнюю лексему заголовка (круглую скобку для Си/Си++, ключевое слово `do` для Паскаля), примерно так:

```
while (TheCollection^.KnownSet^.First = nil) and
    (TheCollection^.ToParse^.First <> nil) and
    (TheCollection^.ToParse^.First^.s = ' ')
do
```

```
    begin
        SkipSpace(TheCollection)
    end;
```

```
while (!the_collection->known_set->first &&
    the_collection->to_parse->first &&
    the_collection->to_parse->first->s == ' '
)
    {
        skip_space(the_collection);
    }
```

Роль зрительного разделителя в этом случае играет как раз этот вот завершающий символ заголовка.

Если вы используете стиль, при котором открывающая операторная скобка оставляется на одной строке с заголовком, вы можете воспользоваться ещё одним вариантом форматирования: как в предыдущем примере, снесите последнюю лексему заголовка на отдельную строку, и на этой же строке оставьте открывающую операторную скобку:

```
while (TheCollection^.KnownSet^.First = nil) and
    (TheCollection^.ToParse^.First <> nil) and
    (TheCollection^.ToParse^.First^.s = ' ')
do begin
    SkipSpace(TheCollection)
end;
```

```
while (!the_collection->known_set->first &&
    the_collection->to_parse->first &&
    the_collection->to_parse->first->s == ' '
) {
    skip_space(the_collection);
}
```

Такой стиль нравится не всем, но вы и не обязаны ему следовать; даже если везде вы оставляете операторную скобку на строке заголовка, для случая многострочного заголовка вы вполне можете сделать исключение и оформлять его так, как показано в первом примере этого параграфа.

2.1.4. Заголовок и тело подпрограммы

При всей кажущейся схожести ситуаций заголовок подпрограммы (функции или паскалевской процедуры) представляет собой

явление совершенно иное, нежели заголовок сложного оператора. В Паскале в качестве тела используется не просто составной оператор, а так называемый *блок*, включающий в себя секцию локальных описаний. В Си и Си++ блоком является любой составной оператор, ведь описать локальную переменную можно в начале любого такого оператора, но при этом в качестве тела функции используется именно блок с операторными скобками, заменить его одним простым оператором нельзя. Отметим, что ранние версии Си[3] также предусматривали информацию, помещаемую между заголовком и телом функции — именно там размещалась информация о типах формальных параметров. В Си++ снова появилась ситуация, когда некую информацию приходится поместить между заголовком и телом — это список инициализаторов в конструкторе.

В отличие от сложного оператора, подпрограмма — это не фрагмент реализации того или иного алгоритма, а *структурная единица программы как целого*, и для подпрограмм применяются несколько иные соглашения об оформлении. Прежде всего, вне зависимости от избранного стиля для сложных операторов **операторная скобка, с которой начинается внешний блок подпрограммы, всегда располагается на отдельной строке**. Для Паскаля это требование очевидно, достаточно вспомнить о секциях локальных описаний. Для случая чистого Си, возможно, это требование покажется не столь очевидным, но объяснение можно найти и здесь. Дело в том, что, в отличие от заголовка сложного оператора, **заголовок подпрограммы имеет самостоятельное значение**, то есть часто встречается отдельно от тела подпрограммы — при этом он служит *объявлением* соответствующей подпрограммы, в отличие от *описания*, в котором присутствует тело. В Паскале тело заменяют директивой «`forward;`», тогда как в Си объявление функции просто завершают точкой с запятой (опять-таки, вместо тела). Разнесение заголовка и тела на разные строки позволяет подчеркнуть факт самостоятельности заголовка.

Второе правило состоит в том, что (опять-таки вне зависимости от стиля, используемого для сложных операторов) **операторные скобки внешнего блока подпрограммы никогда не сдвигают относительно заголовка**. Например, если подпрограмма ни во что

[3]Имеется в виду так называемый «K&R C», то есть синтаксис, изначально предложенный Керниганом и Ритчи. Все компиляторы до сих пор имеют режим поддержки этого варианта синтаксиса, несмотря на то, что далеко не все программисты знают о его существовании.

не вложена[4], то операторные скобки, обрамляющие её блок, **всегда** размещаются в первой (крайней левой) позиции строки.

При этом остаются в действии общие правила: после открывающей операторной скобки в строке больше ничего не пишут, а закрывающую операторную скобку размещают точно под началом закрываемой ею структуры, то есть в данном случае — точно под открывающей операторной скобкой.

Таким образом, следующие варианты оказываются неправильными:

```
function cube(a: real): real;
    begin
        cube := a * a * a
    end;
```

```
float cube(float a)
    {
        return a * a * a;
    }
```

```
function cube(a: real): real; begin
    cube := a * a * a
end;
```

```
float cube(float a) {
    return a * a * a;
}
```

```
function cube(a: real): real;
begin cube := a * a * a end;
```

```
float cube(float a)
{ return a * a * a; }
```

Правильно будет написать так:

```
function cube(a: real): real;
begin
    cube := a * a * a
end;
```

```
float cube(float a)
{
    return a * a * a;
}
```

2.2. Особенности оформления операторов выбора

Оба рассматриваемых нами языка программирования — и Паскаль, и Си — поддерживают *операторы выбора*, позволяющие выполнить один из нескольких имеющихся альтернативных наборов операторов. В Паскале это оператор `case`, в Си — оператор `switch-case`. Вне зависимости от того, какого из трёх основных стилей вы придерживаетесь, оператор выбора требует ответить ещё на один вопрос: *будете ли вы сдвигать метки*. Вариантов ответа на этот вопрос ровно два: положительный и отрицательный, и означает это, что **метки, обозначающие начало очередной альтернативы в операторе выбора, можно либо оставлять в той колонке, где начинается заголовок оператора выбора, либо сдвинуть относительно последнего на размер отступа**. Как обычно в таких

[4] В языке Си это верно для любой функции, поскольку вложенных функций в Си нет; в Паскале предусмотрены вложенные подпрограммы, а в Си++ подпрограмма может быть вложена в класс или структуру в качестве *метода*.

случаях, нужно зафиксировать свой ответ на этот вопрос и оформлять операторы выбора единообразно в тексте всей программы, то есть вы можете выбрать любой из двух вариантов, но только один раз — на всю программу.

Поясним сказанное на примере. Язык Паскаль не предполагает в составе оператора выбора использование слова `begin` (хотя предполагает слово `end`), поэтому оформление заголовка и тела оператора выбора, вообще говоря, не зависит от того, оставляем ли мы открывающую операторную скобку на одной строке с заголовком сложного оператора, сносим ли мы её на следующую строку и снабжаем ли мы её своим собственным отступом; с другой стороны, каждая ветка оператора выбора предполагает использование только одного оператора, так что в большинстве случаев здесь приходится использовать составные операторы, и их оформление уже зависит от избранного стиля. Если мы оставляем операторную скобку на одной строке с заголовком, то в зависимости от нашего решения по поводу меток мы можем написать так:

```
case Operation of
    '+': begin
        writeln('Addition');
        c := a + b;
    end;
    '-': begin
        writeln('Subtraction');
        c := a - b;
    end;
    else begin
        writeln('Error');
        c := 0;
    end
end
```

```
case Operation of
'+': begin
    writeln('Addition');
    c := a + b;
end;
'-': begin
    writeln('Subtraction');
    c := a - b;
end;
else begin
    writeln('Error');
    c := 0;
end
end
```

Хотя оба варианта являются допустимыми, первый выглядит существенно привлекательнее и понятнее, так что *если мы пишем на Паскале и оставляем операторную скобку на одной строке с заголовком, будет лучше принять решение в пользу сдвигания меток.*

Если мы условились сносить операторную скобку на следующую строку, но не сдвигать её, вышеприведённый фрагмент кода должен выглядеть так:

```
case Operation of
    '+':
    begin
        writeln('Addition');
        c := a + b;
    end;
    '-':
    begin
        writeln('Subtraction');
        c := a - b;
    end;
    else
    begin
        writeln('Error');
        c := 0;
    end
end
```

```
case Operation of
'+':
begin
    writeln('Addition');
    c := a + b;
end;
'-':
begin
    writeln('Subtraction');
    c := a - b;
end;
else
begin
    writeln('Error');
    c := 0;
end
end
```

Для этого случая мы также порекомендуем сдвигать метки, но окончательное решение оставим за читателем.

Наконец, если мы не только сносим операторную скобку, но и сдвигаем её, то выглядеть это будет приблизительно так:

```
case Operation of
    '+':
        begin
            writeln('Addition');
            c := a + b;
        end;
    '-':
        begin
            writeln('Subtraction');
            c := a - b;
        end;
    else
        begin
            writeln('Error');
            c := 0;
        end
end
```

```
case Operation of
'+':
    begin
        writeln('Addition');
        c := a + b;
    end;
'-':
    begin
        writeln('Subtraction');
        c := a - b;
    end;
else
    begin
        writeln('Error');
        c := 0;
    end
end
```

Для этого случая наша рекомендация будет противоположной: если операторные скобки имеют свой собственный сдвиг, то метки лучше не сдвигать, результат будет эстетичнее.

В языке Си оператор выбора (`switch`) устроен несколько иначе. Во-первых, в синтаксисе оператора `switch` предусмотрена обязательная открывающая фигурная скобка, в результате чего заголовок `switch` становится зависимым от избранного основного стиля. Во-вторых, в отличие от Паскаля, в операторе выбора языка Си мы име-

ем дело не с *множеством ветвей*, а с одним линейным фрагментом кода, имеющим несколько точек входа (по числу меток `case`). Каждая «ветка» может содержать произвольное количество операторов, так что составные операторы в качестве ветвей оператора `switch` применяются очень редко[5]. Уместно напомнить, что если ветви оператора выбора являются взаимоисключающими, то каждую из них нужно заканчивать оператором `break;`, чтобы не допустить выполнения ветви, следующей непосредственно за данной.

Как и в случае Паскаля, при работе на Си нам придётся принять решение о том, сдвигать или не сдвигать метки на свой отступ. Если мы оставляем открывающую операторную скобку на одной строке с заголовком оператора, результат (со сдвигом метки и без такового) будет выглядеть так:

```
switch (operation) {
    case '+':
        printf("Addition\n");
        c = a + b;
        break;
    case '-':
        printf("Subtraction\n");
        c = a - b;
        break;
    default:
        printf("Error\n");
        c = 0;
}
```

```
switch (operation) {
case '+':
    printf("Addition\n");
    c = a + b;
    break;
case '-':
    printf("Subtraction\n");
    c = a - b;
    break;
default:
    printf("Error\n");
    c = 0;
}
```

Если мы решили сносить открывающую операторную скобку на следующую строку, но не сдвигать её, вышеприведённый фрагмент нужно будет оформить почти так же, отличаться будет только положение скобки:

```
switch (operation)
{
    case '+':
        printf("Addition\n");
/* ... */
```

```
switch (operation)
{
case '+':
    printf("Addition\n");
/* ... */
```

Наконец, если мы всё же решили сдвигать составной оператор на дополнительный отступ, для оператора `switch` получим примерно следующее:

[5] Единственная разумная причина для их применения в ветвях оператора `switch` — это необходимость описания локальных переменных. В этом случае используйте обычные правила для размещения операторных скобок, считая `case`-метку особым случаем заголовка сложного оператора.

```
switch (operation)
    {
        case '+':
            printf("Addition\n");
            c = a + b;
            break;
        case '-':
            printf("Subtraction\n");
            c = a - b;
            break;
        default:
            printf("Error\n");
            c = 0;
    }
```

```
switch (operation)
    {
    case '+':
        printf("Addition\n");
        c = a + b;
        break;
    case '-':
        printf("Subtraction\n");
        c = a - b;
        break;
    default:
        printf("Error\n");
        c = 0;
    }
```

Однозначно рекомендовать тот или иной ответ на вопрос о сдвиге меток здесь сложно. Отметим только, что обычно стиль оформления меток оставляют общим как для `case`-меток, так и для обычных меток, используемых в операторе `goto`, а для случая языка Си++ это же решение распространяют на ключевые слова `public`, `private` и `protected` в заголовках классов; как мы увидим позже, их лучше не сдвигать, и именно по этой причине программисты, работающие на Си++, обычно не сдвигают и `case`'ы. Впрочем, это тоже не догма: вы вполне можете считать `case` отдельным случаем, для которого используется специальный стиль.

Обратите внимание, что в вышеприведённых примерах почти никогда не встречается никакой текст на одной строчке с меткой; единственное исключение — открывающая операторная скобка для стиля, при котором она оставляется на одной строчке с заголовком. Это не случайно. Метка сама по себе может занимать разное количество знакомест, и если использовать строку, где написана метка, для размещения оператора, то это приведёт либо к тому, что операторы одного ранга вложенности окажутся начинающимися с разных позиций, либо размер отступа внутри оператора выбора окажется отличным от такового во всей остальной программе. И то, и другое недопустимо. Приведём несколько примеров подобного безобразия.

```
switch (mode) {
    case mode_fully_active: m++;
                            finalize_request();
                            break;
    case mode_stop: m = 0;
                    drop_request();
                    break;
    default: m = -1;
             drop_request();
}
```

В этом примере операторы разных ветвей одного оператора выбора (то есть имеющие заведомо одинаковый ранг вложенности) сдвинуты на разное количество позиций.

```
switch (mode) {
case mode_fully_active: m++;
                        finalize_request();
                        break;
case mode_stop:         m = 0;
                        drop_request();
                        break;
default:                m = -1;
                        drop_request();
}
```

В этом примере операторы одного ранга сдвинуты на одинаковое количество пробелов относительно объемлющего оператора, но при этом само это количество пробелов заведомо не соответствует размеру сдвига, т. к. определяется длиной идентификатора `mode_fully_active`.

```
switch (mode) {
    case mode_fully_active: m++;
        finalize_request();
        break;
    case mode_stop: m = 0;
        drop_request();
        break;
    default: m = -1;
        drop_request();
}
```

Здесь правильно располагаются все операторы, кроме тех, что оказались на одной строке с меткой (операторы `m++;`, `m = 0;` и `m = -1;`); последние расположены в позициях, определяемых длиной идентификаторов в метках, и при чтении программы эти операторы очень легко не заметить.

Итак, общее правило состоит в том, что **на одной строке с меткой в операторе выбора может стоять разве что открывающая операторная скобка, и более ничего**. Однако из этого правила есть исключение. **Если каждая из ветвей оператора выбора состоит из одного действия (например, вызова подпрограммы), можно записать каждую ветвь на одной строке (метку вместе с действием)**. В этом случае для достижения лучшего эстетического эффекта рекомендуется тела ветвей выбора

расположить, начиная с одной и той же позиции, хотя это и не обязательно. Например:

```
case State of
    StHome:   HandleHome();
    StString: HandleString();
    StEscape: HandleEscape();
    StNum:    HandleNumber();
    else      HandleError()
end
```

```
switch (state) {
    case st_home:   handle_home();   break;
    case st_string: handle_string(); break;
    case st_escape: handle_escape(); break;
    case st_num:    handle_number(); break;
    default:        handle_error();
}
```

2.3. Последовательность взаимоисключающих `if`'ов

Оператор выбора в обоих рассматриваемых языках имеет очень важное ограничение: условием перехода на одну из меток является *равенство* селектирующего выражения одной из *констант*, причём как константы, так и выражение обязаны иметь порядковый тип. Часто требуется сделать выбор одной из нескольких возможных ветвей работы, основываясь на более сложных условиях или на выражении выбора, имеющем непорядковый тип (например, выбор по значению *строки*). Такой выбор приходится реализовывать с помощью длинной цепочки операторов ветвления: `if` // `else if` // `else if` // ... // `else`. Если *формально* следовать правилам, изложенным в § 2.1.2, тела ветвей такой конструкции выбора придётся сдвигать всё дальше и дальше вправо, примерно так:

```
if cmd = "Save" then begin
    writeln('Saving...');
    SaveFile;
end else
    if cmd = "Load" then begin
        writeln('Loading...');
        LoadFile;
    end else
        if cmd = "Quit" then begin
            writeln('Good bye...');
```

```
            QuitProgram;
        end else begin
            writeln('Unknown command');
        end
```

```
if (0 == strcmp(cmd, "Save")) {
    printf("Saving...\n");
    save_file();
} else
    if (0 == strcmp(cmd, "Load")) {
        printf("Loading...\n");
        load_file();
    } else
        if (0 == strcmp(cmd, "Quit")) {
            printf("Good bye...\n");
            quit_program();
        } else {
            printf("Unknown command\n");
        }
```

Несложно видеть, что при таком подходе окажется достаточно семивосьми ветвей, чтобы горизонтальное пространство экрана кончилось; а ведь их может потребоваться гораздо больше. Однако, что гораздо важнее, такой (формально абсолютно правильный) стиль форматирования вводит читателя программы в заблуждение относительно соотношения между ветвями конструкции. Ясно, что эти ветви *имеют одинаковый ранг вложенности*[6]. Но при этом сдвинуты они на *разные* позиции!

Пояснить возникающую проблему можно и другим способом. Такая цепочка `if`'ов представляет собой *обобщение оператора выбора* и служит тем же целям, что и оператор выбора; разница лишь в выразительной мощности. Но ветви оператора выбора пишутся на одном уровне вложенности. Вполне логично считать, что и ветви такой конструкции из `if`'ов должны располагаться на одном уровне отступа.

Достигается это рассмотрением стоящих рядом ключевых слов `else` и `if` как единого целого. Вне зависимости от избранного стиля вы можете написать `else if` на одной строке через пробел, либо разнести их на разные строки, начинающиеся в одной и той же позиции. Если вы не сносите открывающую операторную скобку на отдельную строку, то вышеприведённые фрагменты вы можете оформить вот так (`if` каждый раз с новой строки):

[6] Если вы сомневаетесь в этом, попробуйте поменять ветви местами. Очевидно, что работа программы при этом никак не изменится. А раз это так — значит, предположение о том, что, например, первая из ветвей «главнее» второй, а вторая «главнее» третьей, оказывается неверно.

```
if cmd = "Save" then begin
    writeln('Saving...');
    SaveFile;
end else
if cmd = "Load" then begin
    writeln('Loading...');
    LoadFile;
end else
if cmd = "Quit" then begin
    writeln('Good bye...');
    QuitProgram;
end else begin
    writeln('Unknown command');
end
```

```
if (0 == strcmp(cmd, "Save")) {
    printf("Saving...\n");
    save_file();
} else
if (0 == strcmp(cmd, "Load")) {
    printf("Loading...\n");
    load_file();
} else
if (0 == strcmp(cmd, "Quit")) {
    printf("Good bye...\n");
    quit_program();
} else {
    printf("Unknown command\n");
}
```

либо вот так (`if` на одной строке с предыдущим `else`; это тоже допустимо):

```
if cmd = "Save" then begin
    writeln('Saving...');
    SaveFile;
end else if cmd = "Load" then begin
    writeln('Loading...');
    LoadFile;
end else if cmd = "Quit" then begin
    writeln('Good bye...');
    QuitProgram;
end else begin
    writeln('Unknown command');
end
```

```
if (0 == strcmp(cmd, "Save")) {
    printf("Saving...\n");
    save_file();
} else if (0 == strcmp(cmd, "Load")) {
    printf("Loading...\n");
    load_file();
} else if (0 == strcmp(cmd, "Quit")) {
    printf("Good bye...\n");
    quit_program();
} else {
    printf("Unknown command\n");
}
```

Адаптировать сказанное к случаям, когда открывающая скобка сносится на следующую строку, мы предлагаем читателю самостоятельно; отметим, что вариант с `else if` на одной строке при этом выглядит несколько странно, но всё равно остаётся допустимым.

Подчеркнём, что всё сказанное в этом параграфе относится только к случаю, когда *каждая ветка `else`, кроме самой последней, состоит ровно из одного оператора `if`*. Если это не так, следует применять обычные правила форматирования оператора ветвления.

2.4. Особые случаи и досрочный выход вместо `else`

Ещё одна «хитрая» ситуация с использованием ветвления возникает при ***обработке особых случаев***. Особые случаи возникают обычно в начале подпрограммы. Прежде чем выполнять свою основную задачу, процедура или функция вынуждена проверить корректность параметров, «допустимость» состояния среды вычисления, возможно, попытаться открыть какой-нибудь файл и убедиться,

что он открылся, и т. п., и если что-то не так, подпрограмма завершается, так и не дойдя до своего основного кода.

Характерная ошибка начинающих в такой ситуации — применение полного ветвления, то есть оператора ветвления с веткой `else`: проверяем особый случай, в основную ветку («`then`») помещаем её обработку, обычно довольно короткую, а весь оставшийся текст подпрограммы засовываем в составной оператор после `else`. Если встретился ещё один особый случай — повторяем то же самое и для него, и так далее. В итоге главный фрагмент кода, ради которого в основном и написана подпрограмма, оказывается вложен глубже всего, вся конструкция отвлекает внимание от основной идеи на мелкие детали.

Частично остроту проблемы может снизить применение цепочки `else-if`'ов, оформленной так, как мы обсуждали в предыдущем параграфе, но это не всегда возможно. Часто, прежде чем констатировать особый случай, приходится выполнять одно или несколько подготовительных действий — например, открыть файл или что-то предварительно посчитать; при этом очередной оператор `if-then-else` становится *не единственным* в теле `else` от предыдущей проверки, делая применение особого оформления невозможным.

Намного правильнее будет совсем другое решение. При обнаружении очередного особого случая следует его обработать и *выйти из подпрограммы*, применив для этого `exit` в Паскале или `return` в Си. Никакой `else` после этого не нужен: если мы получили управление после неполного `if`, тело которого заканчивается досрочным возвратом управления, то это заведомо означает, что условие такого `if` оказалось ложным и тело его не выполнялось.

Пусть, к примеру, нам нужно написать подпрограмму, решающую квадратное уравнение. Коэффициенты она получит через параметры; поскольку вычислить и вернуть нужно два числа, а не одно, возврат лучше осуществить через параметры (параметры-переменные в Паскале, параметры адресного типа в Си). Поскольку решение квадратного уравнения возможно не для любых коэффициентов, при решении задачи на Си будет логично возвращать из функции логическое значение: `1` она вернёт, если всё в порядке, ну а если по тем или иным причинам решить уравнение не получается, будет возвращаться значение `0`. Традициям Паскаля такое использование функций несколько противоречит, поэтому при решении на Паскале правильнее будет ввести ещё один параметр-переменную, имеющий логический тип.

Специальных случаев при решении квадратного уравнения получается два. Во-первых, коэффициент при второй степени может оказаться равен нулю, в этом случае уравнение не является квадратным и решать его как квадратное нельзя. Во-вторых, дискриминант мо-

жет оказаться отрицательным[7]. Если применять полное ветвление, решение на Паскале будет выглядеть примерно так:

```
procedure quadratic(a, b, c: real;
                    var ok: boolean; var x1, x2: real);
var
    d: real;
begin
    if a = 0 then
        ok := false
    else
    begin
        d := b*b - 4*a*c;
        if d < 0 then
            ok := false
        else
        begin
            d := sqrt(d);
            x1 := (-b - d) / (2*a);
            x2 := (-b + d) / (2*a);
            ok := true
        end
    end
end;
```

Аналогичное решение на Си будет примерно таким:

```
int quadratic(double a, double b, double c,
              double *x1, double *x2);
{
    double d;
    if (a == 0) {
        return 0;
    } else {
        d = b*b - 4*a*c;
        if (d < 0) {
            return 0;
        } else {
            d = sqrt(d);
            *x1 = (-b - d) / (2*a);
            *x2 = (-b + d) / (2*a);
            return 1;
        }
    }
}
```

[7] Рассматривать решение в комплексной плоскости мы не будем; специальный случай для нас сейчас интереснее.

Самое интересное — собственно решение квадратного уравнения — у нас оказалось «закопано» на третий уровень вложенности, да и в целом управляющая структура в получившихся подпрограммах выглядит довольно страшно, хотя здесь всего-то два специальных случая.

Интересно, что решение на Си мы можем переписать, просто отказавшись от `else` и больше ничего не меняя, поскольку в Си оператор `return` «един в двух лицах» — он и задаёт возвращаемое значение, и завершает функцию. Получится намного короче и намного яснее:

```
int quadratic(double a, double b, double c,
              double *x1, double *x2);
{
    double d;
    if (a == 0)
        return 0;
    d = b*b - 4*a*c;
    if (d < 0)
        return 0;
    d = sqrt(d);
    *x1 = (-b - d) / (2*a);
    *x2 = (-b + d) / (2*a);
    return 1;
}
```

На Паскале так коротко написать не выйдет, поскольку нужен оператор `exit` для выхода из процедуры, он пишется, естественно, на отдельной строке и плюс к тому, появляясь везде вместе с присваиванием, задающим значение флага `ok`, требует составного оператора. Но всё равно получится яснее, чем было:

```
procedure quadratic(a, b, c: real;
                    var ok: boolean; var x1, x2: real);
var
    d: real;
begin
    if a = 0 then
    begin
        ok := false;
        exit
    end;
    d := b*b - 4*a*c;
    if d < 0 then
    begin
        ok := false;
        exit
```

```
        end;
        d := sqrt(d);
        x1 := (-b - d) / (2*a);
        x2 := (-b + d) / (2*a);
        ok := true
    end;
```

На практике встречаются подпрограммы с существенно бóльшим количеством специальных случаев — их может оказаться пять, десять, сколько угодно; при попытке написать такую подпрограмму без досрочных выходов нам попросту не хватит ширины экрана для структурных отступов, не говоря уже о том, что общий случай, который очевидно «главнее» всех отдельно рассматриваемых специальных, окажется вытеснен управляющими структурами на периферию поля зрения.

2.5. Метки и оператор `goto`

В литературе часто можно встретить утверждение о том, что оператор `goto` якобы «нельзя использовать», потому что он запутывает программу. В большинстве случаев это действительно так, но существуют две (не одна, не три, а именно две) ситуации, в которых применение `goto` не только допустимо, но и желательно.

Первая из двух ситуаций очень простая: выход из многократно вложенных управляющих конструкций, например циклов. С выходом из *одного* цикла справятся специально предназначенные для этого операторы `break` и `continue` (в Си они присутствовали с самого начала, в стандарт Паскаля эти операторы не входили, но все современные реализации Паскаля их поддерживают), но что делать, если «выпрыгнуть» нужно, скажем, из трёх циклов, вложенных друг в друга? Конечно, обойтись без `goto`, строго говоря, можно и здесь: в условия циклов вставить проверки какого-нибудь специального флажка, в самом внутреннем цикле этот флажок взвести и сделать `break;`, и тогда все циклы завершатся. В большинстве случаев пресловутый флажок придётся проверять не только в условиях циклов, но и некоторые части тел этих циклов обрамлять `if`'ами, проверяющими всё тот же флажок. Было бы по меньшей мере странно утверждать, что все эти нагромождения окажутся более ясными, нежели один оператор `goto` — конечно, при условии, что имя метки выбрано удачно и соответствует ситуации, в которой на неё делается переход.

Вторую ситуацию описать несколько сложнее. Представьте себе, что вы пишете подпрограмму, которая в начале своей работы забирает себе некий ресурс (выделяет динамическую память, открывает файл и т. п.), а по завершении работы должна этот ресурс освободить

(соответственно освободить память, закрыть файл). Такая схема работы встречается достаточно часто; по-английски высвобождение захваченных ресурсов перед окончанием работы называется *cleanup*, что может быть приблизительно переведено словом *очистка*. Необходимость произвести очистку перед выходом из подпрограммы не представляет никаких проблем, если точка выхода у нас одна; проблемы начинаются, если где-нибудь в середине кода подпрограммы возникает потребность досрочного её завершения (в языке Си для этого используется оператор `return`, в Паскале — псевдопроцедура `exit`).

Пусть, к примеру, в начале нашей подпрограммы производятся три операции по выделению памяти, в конце — три соответствующие операции по её освобождению, и нам потребовалось в двух местах кода подпрограммы сделать досрочное завершение. Попытка обойтись без `goto` приведёт к тому, что в этих двух местах непосредственно перед завершением (т. е. перед `return` или `exit`) придётся продублировать код всех трёх операций по высвобождению памяти. Как известно, дублирование кода до добра обычно не доводит: если мы теперь изменим начало подпрограммы, добавив или убрав операции по захвату ресурса, велика вероятность того, что из получившихся *трёх* фрагментов, осуществляющих очистку (один в конце подпрограммы, два перед точками досрочного завершения) мы исправим только два, а про третий забудем. Поэтому в такой ситуации обычно поступают иначе: перед операциями очистки, находящимися в конце подпрограммы, ставят метку (как правило, её называют `quit` или `cleanup`), а вместо `return` или `exit` делают `goto` на эту метку.

Следует обратить внимание, что в обоих допустимых случаях — и при выходе из вложенных циклов, и при переходе на очистку — переход делается «вниз» по коду, т. е. вперёд по последовательности выполнения (то есть метка стоит в тексте программы ниже оператора `goto`) и «наружу» из управляющих конструкций. Если у вас возникло желание сделать `goto` *назад* — это означает, что вы создаёте цикл, а для циклов есть специальные операторы. Если же вам захотелось «впрыгнуть» внутрь управляющей конструкции, то, следовательно, у вас что-то пошло совсем не так, как надо, и нужно срочно понять причины возникновения таких странных желаний; отметим, что Паскаль такого просто не позволит, а Си такое хотя и допускает, но это не значит, что так действительно можно делать.

Теперь, когда мы знаем, что `goto` использовать в некоторых случаях не только можно, но и нужно, и, более того, мы точно знаем, какие это случаи (повторим ещё раз, этих случаев ровно два, и третьего нет) — остаётся ответить на вопрос, как всё это оформлять. Точнее, сам оператор `goto` никаких проблем с оформлением не вызывает, это обыкновенный оператор, оформляемый по обычным пра-

вилам; но вот то, как следует ставить *метку*, может стать предметом жаркой дискуссии.

Напомним, что меткой всегда помечается *оператор*, то есть пометить меткой пустое место нельзя. Вопросов, на которые нужно дать (зафиксировать) ответ, оказывается два: (1) сдвигать ли метку относительно объемлющей конструкции (отметим, что с аналогичной дилеммой мы уже встречались при обсуждении операторов выбора в § 2.2) и (2) размещать ли помеченный оператор на той же строке, где метка, или на отдельной строке.

Наиболее популярен вариант, при котором метка не сдвигается, то есть пишется в той же горизонтальной позиции, в которой размещены начало и конец объемлющей управляющей структуры, а помеченный оператор сносится на следующую строку, например:

```
procedure GoodProc;
    label quit;
    var p, q: ^SomethingBig;
begin
    new(p);
    new(q);
    { ... }

quit:
    dispose(q);
    dispose(p)
end;
```

```
void good_func()
{
    something_big *p, *q;
    p = malloc(sizeof(*p));
    q = malloc(sizeof(*q));
    /* ... */

quit:
    free(q);
    free(p);
}
```

Метка здесь оказалась в крайней левой позиции исключительно потому, что именно там размещается объемлющая структура (в данном случае это подпрограмма). Метка может встретиться и не на верхнем уровне, например:

```
if cond = 1 then begin
    while f <> 0 do begin
        while g <> 0 do begin
            while h <> 0 do begin
                if t = 0 then
                    goto eject;
            end
        end
    eject:
        writeln("go on...");
    end
end
```

```
if (cond() == 1) {
    while (f() != 0) {
        while (g() != 0) {
            while (h() != 0) {
                if (t() == 0)
                    goto eject;
            }
        }
    eject:
        printf("go on...\n");
    }
}
```

Приведённый вариант наиболее популярен, но это не единственный *допустимый* вариант. Довольно часто оператор, помеченный меткой, пишут в той же строке, что и метка, примерно так:

```
    { ... }                          /* ... */
quit: dispose(q);                quit: free(q);
    dispose(p);                      free(p);
end;                             }
```

Этот вариант неплохо смотрится, если метка — вместе с двоеточием и пробелом после него — занимает по горизонтали меньше места, чем выбранный размер отступа, что позволяет выдержать горизонтальное выравнивание для операторов:

```
    { ... }                          /* ... */
q:  dispose(q);                  q:  free(q);
    dispose(p);                      free(p);
end;                             }
```

Поскольку имя метки из одной буквы само по себе смотрится не слишком приятно, обычно такой стиль применяют в сочетании с использованием табуляции в качестве размера отступа; максимальная длина имени метки при этом составляет 6 символов.

Некоторые программисты предпочитают рассматривать метку просто как часть помеченного оператора, не выделяя её как особую сущность; оператор сдвигают как обычно, но на этот раз уже вместе с меткой. Концовка вышеприведённых подпрограмм в таком стиле будет выглядеть так:

```
    { ... }                          /* ... */
    quit: dispose(q);                quit: free(q);
    dispose(p);                      free(p);
end;                             }
```

Иногда метку сдвигают, но помеченный оператор сносят на следующую строку, примерно так:

```
    { ... }                          /* ... */
    quit:                            quit:
    dispose(q);                      free(q);
    dispose(p);                      free(p);
end;                             }
```

Основной недостаток таких решений — метка «сливается с окружающим пейзажем», перестаёт быть заметна в качестве особой точки в структуре кода; возьмём на себя смелость рекомендовать воздержаться от такого стиля, но, тем не менее, сохраним решение за читателем.

Особо следует рассмотреть ситуацию, когда метка нужна в самом конце составного оператора. Поскольку пометить меткой можно только оператор, в этом случае приходится вставить в программу пустой оператор, обычно это точка с запятой. Здесь у вас есть два варианта: можно оставить точку с запятой после метки на одной с ней строчке, а можно снести на следующую со сдвигом:

```
    { ... }
quit: ;
end;
```

```
    { ... }
quit:
    ;
end;
```

2.6. Управляющая логика

Логические (условные) выражения, в особенности появляющиеся в заголовках операторов `if`, `while` и т. п., оказываются источником целого семейства стилистических ошибок, которые чаще всего встречаются в коде новичков. Самая, если так можно выразиться, *одиозная* из них выглядит примерно так:

```
if flag = true then
    flag := false
else
    flag := true;
```

```
if (flag == 1)
    flag = 0;
else
    flag = 1;
```

Очевидно, что автор такого фрагмента задался целью сменить значение логической переменной на противоположное, но получившийся код, формально абсолютно правильный, способен вызвать у опытного программиста приступ гомерического хохота. В самом деле, ведь оператор

```
flag := not flag;
```

```
flag = !flag;
```

делает *то же самое*, при этом он впятеро короче и, что самое забавное, *понятнее*! Применение такой конструкции говорит о том, что в восприятии автора программы логические выражения — как вычисляемая сущность — начисто отсутствуют. Обычно такой программист подсознательно глубочайшим образом убеждён, что «вот это вот, которое в заголовке `if` или `while` — это где что-нибудь с чем-нибудь сравнивают»; воспринимать логическое выражение как частный случай выражения арифметического, а операции сравнения ставить в один ряд со всеми прочими операциями мозг автора подобного кода категорически отказывается. Вынос сложного условия в функцию, возвращающую логическое значение, попросту не придёт ему в голову; зачастую студенты с такой особенностью мышления не пишут булевы функции даже после явного указания преподавателя, потому что просто не понимают, чего от них хотят.

Возвращаясь к нашему примеру, отметим, что сам факт применения `if` — не единственный его «косяк». Обратите внимание, что в заголовке `if` *логическую величину сравнивают с логической же константой*. Аналогичную ерунду часто можно наблюдать, когда автор программы, преодолев сопротивление собственного подсознания, всё-таки вынес некое сложное условие в функцию, возвращающую

логическую величину, или использует такую функцию из библиотеки. Выглядит это так:

```
if check(x) = true then
```

```
if (check(x) == 1)
```

Бывает, впрочем, и наоборот:

```
if check(x) = false then
```

```
if (check(x) == 0)
```

Пожалуй, использование логической функции в исполнении такого программиста заслуживает аплодисментов, ведь частично ему удалось преодолеть собственные предубеждения; но, как видим, подспудное «в заголовке `if` всегда стоит сравнение» всё ещё никуда не делось. Разумеется, в первом случае следует написать просто

```
if check(x) then
```

```
if (check(x))
```

а во втором применить отрицание:

```
if not check(x) then
```

```
if (!check(x))
```

— заодно попытавшись убедить себя, что сравнение одной величины с другой — это лишь частный случай логических (если угодно, условных) выражений, важный, но не единственный. Вообще стоит обратить внимание на достаточно очевидный факт: **сравнение логической величины с истиной тождественно исходной величине, а сравнение с ложью тождественно её отрицанию**. Как следствие, **логическую величину вообще не следует никогда сравнивать с константами**.

Раз уж мы упомянули вынос сложного условия в функцию, отметим ещё один замечательный пример того, как делать не надо:

```
function check(a, b, c: integer)
                       : boolean;
begin
   if (a < 10) and (b > c) then
       check := true;
   else
       check := false;
end;
```

```
int check(int a, int b, int c)
{
    if (a < 10 && b > c)
        return 0;
    else
        return 1;
}
```

Разумеется, вместо этого кошмара нужно написать просто

```
function check(a, b, c: integer)
                       : boolean;
begin
    check := (a < 10) and (b > c);
end;
```

```
int check(int a, int b, int c)
{
    return a < 10 && b > c;
}
```

Повторим ещё раз, что к подобным ошибкам приводит дефект восприятия, формирующийся у начинающих из-за обилия сравнений в заголовках ветвлений и циклов при полном отсутствии примеров противоположных. Конечно, этот дефект, если он всё-таки возник, нужно исправлять. Иногда помогают следующие правила:

Условное выражение — это тоже выражение
Логическая величина — это тоже величина
Булева переменная — это тоже переменная

К сожалению, правила, оформленные в виде лозунгов, несколько опасны: их можно просто зазубрить без понимания, и тогда толку от них никакого не будет. Лучше всего, естественно, перекос в рассматриваемых примерах ликвидировать массовым приведением примеров противоположных, в данном случае — таких, где в заголовках операторов `if` и `while` нет ничего похожего на операции сравнения.

Отметим ещё одну стилистическую ошибку, которую, в отличие предыдущих, часто делают не только новички, но и опытные программисты:

```
if not condition then
   { ... }
else
   { ... }
```

```
if (!condition)
   /* ... */
else
   /* ... */
```

Здесь мы видим *отрицание в условии*, а затем — ветви `then` и `else`. Если бы ветви `else` не было, отрицание условия не вызывало бы никаких возражений, но когда она есть, возникает резонный вопрос: *простите, а кто мешает вам поменять местами ветви?*

2.7. Как разбить длинную строку

Выше мы уже сталкивались с проблемой слишком длинных строк, точнее — с её частным случаем, когда слишком длинным оказывается заголовок сложного оператора (см. § 2.1.3). Рассмотрим некоторые другие случаи, когда ширины экрана катастрофически не хватает.

Хотелось бы сразу заметить, что лучший способ справиться с ситуациями, описанными в этом параграфе — это попросту *не допускать* их. Часто руководства по стилю оформления кода пишутся в предположении, что программист всегда может избежать нежелательной ситуации, и там попросту не говорится о том, что́ же нужно делать, если ситуация всё-таки сложилась; такое «умолчание» приводит к тому, что программисты начинают выходить из положения как попало, причём часто даже разными способами в рамках одной программы. Во избежание этого мы приведём несколько примеров

того, как *возможно* действовать, если длинная строка никак не хочет становиться короче.

2.7.1. Слишком длинное выражение в присваивании

Итак, вы пишете программу и очередная строка, содержащая обыкновенное присваивание, не желает помещаться ни в 75, ни даже в 79 символов по ширине. Первое, что мы посоветуем сделать в этой ситуации — это **попытаться разбить строку по знаку присваивания**. Если слева от присваивания стоит что-то достаточно длинное, такой вариант вполне может помочь, например:

```
MyArray[f(x)].ThePtr^.MyField :=
    StrangeFunction(p, q, r) + AnotherStrangeFunction(z);
```

Обратите внимание, что выражение, стоящее справа от присваивания, мы не просто снесли на следующую строку, но и сдвинули вправо на размер отступа. Если после этого выражение всё ещё не помещается на экран, можно начать разбивать и его, и делать это лучше всего по знакам операций наименьшего приоритета, например:

```
MyArray[f(x)].ThePtr^.MyField :=
    StrangeFunction(p, q, r) * SillyCoeff +
    AnotherStrangeFunction(z) / SecondSillyCoeff +
    JustAVariable;
```

Может получиться так, что даже после этого экран всё ещё остаётся слишком узким для вашего выражения. Тогда можно попытаться начать разбивать на несколько строк подвыражения, входящие в ваше выражение; их части нужно, в свою очередь, сдвинуть ещё на один отступ, чтобы выражение более-менее легко читалось (насколько вообще возможно для такого «монструозного» выражения говорить о лёгкости прочтения):

```
MyArray[f(x)].ThePtr^.MyField :=
    StrangeFunction(p, q, r) +
    AnotherStrangeFunction(z) *
        FunctionWhichReturnsCoeff(z) *
        AnotherSillyFunction(z) +
    JustAVariable;
```

Если выражение состоит из большого количества подвыражений «верхнего уровня» (например, слагаемых), которые сами по себе не очень длинные, вполне допустимо оставлять несколько таких подвыражений в одной строке:

```
MyArray[f(x)].ThePtr^.MyField :=
    a + b + c + d + e + f + g + h + i + j + k + l + m +
    n + o + p + q + r + s + t + u + v + w + x + y + z;
```

(конечно, если в реальности вам пришлось вот так вот сложить 26 переменных, то это повод задуматься, почему вы не используете массив; здесь мы приводим сумму простых переменных исключительно для иллюстрации, в реальной жизни вместо переменных у вас будет что-то более сложное).

Отдельного обсуждения заслуживает ситуация, когда слева от присваивания стоит простое имя переменной или даже выражение, но короткое, так что разбивка строки по знаку присваивания не даёт никакого (или почти никакого) выигрыша. Естественно, выражение справа от присваивания по-прежнему лучше всего разбивать по операциям низшего приоритета; вопрос лишь в том, с какой позиции начинать каждую следующую строку. Ответов на этот вопрос ровно два: каждую следующую строку можно либо сдвигать на один отступ, как в примерах выше, либо размещать её начало точно под началом выражения в первой строке нашего присваивания (ровно после самого знака присваивания). Сравните, вот пример первого варианта:

```
MyArray[n] := StrangeFunction(p, q, r) * SillyCoeff +
    AnotherStrangeFunction(z) / AnotherCoeff +
    JustAVariable;
```

А вот так тот же код будет выглядеть, если выбрать второй вариант:

```
MyArray[n] := StrangeFunction(p, q, r) * SillyCoeff +
              AnotherStrangeFunction(z) / AnotherCoeff +
              JustAVariable;
```

Оба варианта допустимы, но обладают существенными недостатками. Первый вариант заметно проигрывает второму в ясности, но при этом второй вариант требует для второй и последующих строк нестандартного размера отступа, который оказывается зависящим от длины выражения слева от присваивания. В частности, этот (второй) вариант совершенно не годится, если вы используете в качестве отступа табуляцию, потому что добиться такого выравнивания можно только пробелами, а смешивать пробелы и табуляции не следует ни в коем случае.

Если недостатки обоих вариантов кажутся вам существенными, вы можете взять себе за правило *всегда* переводить строку после знака присваивания, если весь оператор целиком не поместился на одну строку. Такой вариант (рассмотренный в начале параграфа) свободен от обоих недостатков, но требует использования лишней строки;

впрочем, как уже говорилось, запас строк во Вселенной неограничен. Выглядеть это будет так:

```
MyArray[n] :=
    StrangeFunction(p, q, r) * SillyCoeff +
    AnotherStrangeFunction(z) / AnotherCoeff +
    JustAVariable;
```

2.7.2. Слишком длинный вызов подпрограммы

Если при обращении к процедуре или функции у вас не помещаются в одну строку параметры, то строку, естественно, придётся разорвать, и обычно это делают после очередной запятой, отделяющей параметры друг от друга. Как и в случае с разбросанным по нескольким строкам выражением, возникает вопрос, в какой позиции начать вторую и последующие строки, и вариантов тоже ровно два: сдвинуть их на размер отступа, либо сдвинуть их так, чтобы все параметры оказались записаны «в столбик». Первый вариант выглядит примерно так:

```
VeryGoodProcedure("This is the first parameter",
    "Another parameter", YetAnotherParameter,
    More + Parameters * ToCome);
```

Второй вариант для приведённого примера будет выглядеть так:

```
VeryGoodProcedure("This is the first parameter",
                  "Another parameter",
                  YetAnotherParameter,
                  More + Parameters * ToCome);
```

Этот вариант, как и аналогичный вариант форматирования выражений, не годится при использовании табуляции в качестве размера отступа: добиться такого выравнивания можно только пробелами, а смешивать пробелы и табуляции не следует.

Если оба варианта вам по тем или иным причинам не понравились, мы можем предложить ещё один вариант, который используется очень редко, хотя и выглядит вполне логичным: рассматривать имя подпрограммы и круглые скобки в качестве объёмлющей конструкции, а параметры — в качестве вложенных конструкций. В этом случае наш пример будет выглядеть так:

```
VeryGoodProcedure(
    "This is the first parameter",
    "Another parameter", YetAnotherParameter,
    More + Parameters * ToCome
);
```

2.7.3. Слишком длинный заголовок подпрограммы

В ситуации, когда на одну строчку не влез *заголовок подпрограммы* (процедуры или функции), следует прежде всего внимательно рассмотреть возможности его сокращения, при этом допуская, в числе прочего, вариант с изменением разбивки кода на подпрограммы. Как уже говорилось (см. § 1.2.4), подпрограммами с шестью и более параметрами очень тяжело пользоваться, так что, если причиной «распухания» заголовка оказалось большое количество параметров, следует подумать, нельзя ли так изменить вашу архитектуру, чтобы это количество сократить, возможно, ценой появления бóльшего количества подпрограмм.

Второй момент, на который следует обратить внимание — это имена (идентификаторы) параметров. Поскольку эти имена *локальны для вашей подпрограммы*, их вполне можно сделать короткими, вплоть до двух-трёх букв. Конечно, при этом мы можем лишить имена самопоясняющей силы, но заголовок подпрограммы в любом случае обычно снабжают комментарием (хотя бы коротким), и соответствующие пояснения о смысле каждого параметра можно вынести в этот комментарий.

К сожалению, нередки ситуации, когда даже после всех этих ухищрений заголовок по-прежнему не помещается в 79 знакомест. Скорее всего, придётся разнести на разные строки список параметров, но прежде чем это делать, стоит попытаться убрать на отдельные строки начало и конец заголовка. Так, в Паскале заголовок подпрограммы начинается со слова `procedure` или `function`; это слово можно написать на отдельной строке (следующая строка при этом *не сдвигается*!). Кроме того, тип возвращаемого значения функции, указанный в конце заголовка, также можно снести на отдельную строку вместе с двоеточием, но эту строку следует сдвинуть так, чтобы тип возвращаемого значения оказался где-то под концом списка параметров (даже если вы используете табуляции). Дело в том, что читатель вашей программы именно там (где-то справа) ожидает увидеть тип возвращаемого значения, и на то, чтобы отыскать его на следующей строчке слева, а не справа, читателю придётся потратить лишние усилия. Всё вместе может выглядеть примерно так:

```
procedure
VeryGoodProcedure(fpar: integer; spar: MyBestRecordPtr; str: string);
begin
    {...}
end;

function
VeryGoodFunction(fpar: integer; spar: MyBestRecordPtr; str: string)
```

```
                                                    : ExcellentRecordPtr;
begin
    {...}
end;
```

Для языка Си можно посоветовать указать на отдельной строке тип возвращаемого значения, а также всевозможные «хитрые» атрибуты, как то `static`, `inline` и тому подобное. В Си++ встречаются атрибуты функций, записываемые *после* списка параметров: `const` для константных методов и атрибут `throw()`, которые также можно разместить на отдельной строке, сдвинув максимально вправо. Наконец, если функция возвращает значение сложного типа, часть описания этого типа может также оказаться в конце заголовка[8]. Поскольку тип возвращаемого значения в Си может занимать довольно много места, такое разбиение иногда позволяет справиться с чрезмерно длинным заголовком:

```
static const char * const *
make_commandline_vector(command_database_item *lst, int cmdid, int t)
{
    /* ... */
}
```

Если все вышеприведённые способы не помогли и заголовок попрежнему не помещается по ширине в 79 знакомест, вам остаётся только один вариант — разбить на части список параметров. Естественно, переводы строк вставляются между описаниями отдельных параметров. Для языка Си в этом плане всё очевидно, строку разрывают сразу после запятой, стоящей после очередного имени параметра, тогда как для Паскаля есть ещё одна особенность: если несколько параметров имеют один тип и перечислены через запятую, желательно оставить их на одной строке, а разрывы строк размещать после точки с запятой, стоящей после имени типа. В любом случае остаётся вопрос относительно горизонтального размещения (сдвига) второй и последующих строк. Как и для рассмотренных выше случаев длинного выражения и длинного вызова подпрограммы, здесь есть три варианта: (1) начать список параметров на одной строке с именем подпрограммы, а последующие строки сдвинуть на размер отступа; (2) начать список на одной строке с именем подпрограммы, а последующие строки сдвинуть так, чтобы все описания параметров начинались в одной и той же позиции (этот случай не

[8] Отметим, что так всё же лучше не писать; если воспользоваться директивой `typedef` для описания типа возвращаемого значения, ясность программы резко возрастёт. Учтите, что, например, описание функции, которая возвращает указатель на массив указателей на функции, принимающие на вход указатель на функцию, заставит надолго задуматься большинство программистов, даже очень опытных, какие бы сказки они там ни рассказывали на эту тему.

годится, когда для форматирования используется табуляция), и (3) рассматривая имя подпрограммы и открывающую скобку как заголовок сложной структуры, снести описание первого параметра на следующую строку, сдвинув его на размер отступа, остальные параметры разместить под ним, а круглую скобку, закрывающую список параметров, разместить на отдельной строке в первой позиции (под началом заголовка).

2.7.4. Длинная строковая константа (литерал)

Случай, когда в строке программы не помещается текстовая константа, заслуживает особого обсуждения. Конечно, самое *худшее*, что можно сделать — это «заэкранировать» символ перевода строки, продолжив строковый литерал в начале следующей строчки кода. **Не делайте так никогда**:

```
        printf("This is a string which unfortunately is \
too long to fit on a single code line\n");
```

Язык Си++, а с некоторых пор и чистый Си[9] для таких случаев поддерживают конкатенацию строковых литералов: две строковые константы, встреченные в тексте программы подряд (разделённые только пробельными символами, включая, естественно, перевод строки), компилятор «сольёт» в одну строку, что позволяет написать так:

```
printf("This is a string which unfortunately is "
       "too long to fit on a single code line\n");
```

В Паскале такой возможности нет, но есть операция сложения строк, что позволяет сделать так:

```
writeln('This is a string which unfortunately is ' +
        'too long to fit on a single code line');
```

Однако бывает и так, что в языке нет ни одной из этих возможностей; например, когда-то давно чистый Си не поддерживал конкатенацию строковых литералов, а в некоторых (очень редких) случаях к коду предъявляется требование совместимости с очень старыми версиями компиляторов. Кроме того, единое текстовое сообщение, выдаваемое как одна строка, т. е. не содержащее символов перевода строки среди выдаваемого текста, вообще лучше не разносить на разные строки кода (см. замечание на стр. 30). Остаётся попробовать ещё два способа борьбы с длиной строкового литерала.

[9] Эта возможность была узаконена в ANSI C, то есть в самом первом из официальных стандартов этого языка. Надо сказать, что этот стандарт стал для Си первым и последним, который хоть как-то можно терпеть.

Во-первых, как это ни банально, стоит подумать, нельзя ли сократить содержащуюся в строке фразу без потери смысла. Как известно, краткость — сестра таланта. Например, для рассматриваемого примера возможен такой вариант:

```
printf("String too long to fit on a line\n");
```

Смысл английской фразы мы оставили прежним, но теперь она вопреки собственному смыслу вполне нормально помещается в строке кода.

Во-вторых, если сокращать ничего не хочется, можно заметить, что некоторые строковые константы уместились бы в строке кода, если бы содержащий их оператор начинался в крайней левой позиции, т. е. если бы не структурный отступ. В такой ситуации справиться с упрямой константой совсем легко — достаточно дать ей имя. Например, в языке Си можно воспользоваться макросом:

```
#define THE_LONG_STRING \
    "This string could be too long if it was placed in the code"
/* ... */
                    printf("%s\n", THE_LONG_STRING);
```

Конечно, такое макроопределение следует поместить вне управляющих конструкций, чтобы не пришлось его сдвигать; можно вынести его в начало файла или же (что в большинстве случаев предпочтительно) разместить его непосредственно перед описанием функции, в которой он используется.

К сожалению, бывают ситуации, в которых не помогает ни один из перечисленных способов. Тогда остаётся лишь последовать правилам из Linux Kernel Coding Style Guide и оставить в коде строку, длина которой превышает 80 символов. Следите только, чтобы это превышение не выходило за грань разумного. Так, если получившаяся строка кода «вылезла» за 100 символов, и при этом вам кажется, что ни одним из вышеперечисленных способов побороть зловредную константу нельзя, то это вам, скорее всего, только кажется; автор этих строк ни разу за всю свою практику не видел ситуации, в которой строковую константу нельзя было бы уместить в обычные 80 символов, не говоря уже о ста.

2.8. Разделители и пробелы

Знаки, которые в тексте программы выделяются в отдельные лексемы независимо от наличия или отсутствия вокруг них пробельных символов, называются *разделителями*. Обычно это знаки арифметических операций, скобки и знаки препинания, такие как запятая,

точка с запятой и двоеточие. Например, операции `+` и `-` являются разделителями как в Паскале, так и в Си, т. е. в обоих языках можно написать `a + b`, а можно и `a+b`, смысл от этого не изменится. В то же время операция `and` в Паскале разделителем не является: `a and b` — это не то же самое, что `aandb`; между тем её аналог из языка Си — операция `&&` — является разделителем, т. е. можно написать `a&&b` без ущерба для смысла. Можно указать и такие языки, в которых арифметические операции разделителями не являются — это, например, Пролог и Лисп, в которых символы `+`, `-` и прочие могут входить в идентификаторы.

Несмотря на то, что пробельные символы вокруг разделителей не обязательны, в ряде случаев их добавление может сделать текст программы более эстетичным и читаемым — *но не всегда*. При этом, как водится, невозможно предложить единый универсальный свод правил по поводу расстановки таких пробелов; существуют различные подходы к этому, имеющие свои достоинства и недостатки.

С уверенностью можно сказать, что **для знаков препинания — запятых, точек с запятой и двоеточий** — лучше всего подходит одно простое правило: **перед ними пробелы не ставятся, а после них — наоборот, ставятся** (это может быть как собственно пробел, так и перевод строки).

Некоторые программисты ставят пробелы с внутренней стороны скобок (круглых, квадратных, фигурных и угловых), примерно так:

```
MyProcedure( a, b[ idx + 5 ], c );
```

Мы не рекомендуем так делать, хотя это и допустимо. Единственное исключение — **фигурные скобки языка Си**, которые воспринимаются скорее не как скобки, а как структурирующие символы; их **рекомендуется выделять пробелами с обеих сторон**. Во всех остальных случаях пробелы с внутренней стороны скобок ставить не рекомендуется; лучше написать так:

```
MyProcedure(a, b[idx + 5], c);
```

При обращении к процедурам и функциям **пробел между именем вызываемой подпрограммы и открывающей скобкой обычно не ставят**, так же как и пробел между именем массива и открывающей квадратной скобкой операции индексирования. Интересно, что для языка Си часто рекомендуют ставить пробел в заголовке сложного оператора между его названием (ключевым словом `if`, `while`, `for`, `switch`) и открывающей скобкой, обрамляющей условное выражение — чтобы отличить эти случаи от случая вызова функции. Сравните:

```
for (i = 0; f(i) < g(); i++) {
    if (!cond(i)) {
        proceed(i);
        break;
    }
}
```

В этом фрагменте мы поставили пробелы перед открывающими скобками в заголовках операторов `for` и `if`, но не поставили пробелов перед скобками после имён функций `f`, `g`, `cond` и `proceed`. Следование этому правилу не обязательно, можно обойтись без пробелов после названий операторов, но именно такая рекомендация встречается во многих руководствах по стилю оформления кода.

Возвращаясь к пробелам внутри скобок, добавим, что иногда можно встретить вариант стиля, при котором такие пробелы ставятся в заголовках операторов, но не в вызовах функций:

```
for( i = 0; f(i) < g(); i++ ) {
    if( !cond(i) ) {
        proceed(i);
        break;
    }
}
```

Можно придумать и другие варианты допустимых стилей. Выбор, как обычно, за вами.

Несколько особняком стоит вопрос о том, какие из *арифметических операций* следует выделять пробелами, и как (с одной стороны или с обеих). Одна из самых популярных и при этом внятных рекомендаций звучит так: **символы бинарных операций выделяются пробелами с обеих сторон, символы унарных операций пробелами не выделяются**. При этом следует учитывать, что операции выборки поля из сложной переменной (точка в Паскале и Си, «стрелка» в Си) бинарными не являются, поскольку справа у них не операнд, а название поля, которое не может быть значением выражения. Подчеркнём, что такой стиль является наиболее популярным, но никоим образом не единственным; возможно следование совсем другим правилам, например — в любом выражении выделять пробелами бинарные операции наименьшего приоритета (то есть операции самого «верхнего» уровня), а остальные пробелами не выделять, и так далее.

2.9. Особенности Паскаля

Оформление кода на любом языке программирования имеет свои особенности, обусловленные именно этим языком. Выше мы уже

останавливались на целом ряде моментов, различающихся для Паскаля и Си; этот параграф мы полностью посвятим Паскалю.

2.9.1. Регистр букв, ключевые слова и имена

Одна из особенностей Паскаля — его принципиальная нечувствительность к регистру букв: один и тот же идентификатор можно написать как `myname`, `MYNAME`, `MyName`, `mYnAmE` и так далее. То же самое касается и ключевых слов: `begin`, `Begin`, `BEGIN`, `bEgIn`... компилятор всё стерпит.

Разумеется, такая толерантность компилятора — совершенно не повод писать как попало. Наиболее распространено мнение, что **ключевые слова следует писать на нижнем регистре (т. е. маленькими буквами)** и не придумывать на эту тему ничего лишнего. Из этого правила можно сделать одно вполне логичное исключение: написать большими буквами те `BEGIN` и `END`, которые обрамляют *главную программу*; можно, впрочем, этого и не делать.

Иногда можно встретить программы на Паскале, в которых ключевые слова написаны с большой буквы: `Begin`, `End`, `If` и т. д.; попадаются также программы, в которых с большой буквы написаны только названия управляющих операторов (`If`, `While`, `For`, `Repeat`), а все прочие, включая `begin` и `end`, пишутся на нижнем регистре. Всё это допустимо, хотя и экзотично; необходимо только чётко сформулировать для себя правила, в каких случаях как мы пишем ключевые слова, и неукоснительно следовать им во всей программе.

Совсем редко можно встретить текст, где все ключевые слова воспроизведены заглавными буквами. Как показывает практика, такой текст читается тяжелее; следовательно, так писать не надо. Ну и, естественно, не надо прибегать к изыскам вроде `BeGiN` или, скажем, `FunctioN` — такое тоже встречается в программах, но относится к области бессмысленного (и где-то даже *вызывающего*) позёрства.

Что касается выбора имён для идентификаторов, то в Паскале сложились определённые традиции на эту тему. Если имя переменной состоит из одной буквы или представляет собой короткую аббревиатуру (что вполне допустимо для локальных переменных, см. § 1.2.1), имя такой переменной обычно пишут на нижнем регистре: `i`, `j`, `t`, `tmp`, `res`, `cnt` и т. п. Если же имя переменной (а равно имя типа, процедуры или функции, константы, метки и так далее) состоит из одного или нескольких *слов*, то эти слова записывают слитно, при этом каждое слово начинают с большой буквы: `Counter`, `Summa`, `StrangeGlobalVariable`, `ListPtr`, `UserListItem`, `ExcellentFunction`, `KillThemAll`, `ProduceSomeCompleteMess` и тому подобное.

Впрочем, не менее популярен также и вариант, при котором заглавные буквы используются только в идентификаторах, состоящих из двух и более слов, а короткие идентификаторы, в том числе состоящие из одного слова, пишут на нижнем регистре: `i`, `cnt`, `counter`, `dogs`, но при этом `MainCounter`, `AngryDogs`, `GoParty` и т. п.

2.9.2. Вложенные подпрограммы

Паскаль — один из немногих языков, в которых возможны подпрограммы внутри подпрограмм. Бездумное использование этой возможности приводит к тому, что в коде становится трудно «найти концы»: подпрограммы верхнего уровня начинают разбухать, вбирая в себя всё новые и новые вложенные подпрограммы, итогом чего вполне может стать расстояние в несколько экранов кода между заголовком подпрограммы и последним её `end`'ом. Распухнуть такая «матрёшка» может не только в высоту, но и в ширину: часто возникает соблазн внутри вложенной процедуры или функции предусмотреть ещё одну вложенную подпрограмму, и так далее. Естественно, каждую такую «более вложенную» подпрограмму приходится сдвигать всё дальше и дальше вправо, оставляя меньше резерва на вложенность управляющих конструкций.

В литературе встречается рекомендация *вовсе не использовать вложенные подпрограммы*, но это всё же слишком категоричный вариант. Вложенные подпрограммы сослужат вам хорошую службу, если вы будете придерживаться нескольких простых правил:

1. уровень вложенности подпрограмм должен быть только один; не следует вкладывать новые подпрограммы внутрь уже имеющихся *вложенных* подпрограмм;
2. вложенные подпрограммы должны быть небольшими: обычно их тело не превышает трёх-четырёх строк;
3. общая длина подпрограммы внешнего уровня не должна превышать 50–70 строк, включая все вложенные подпрограммы.

Если у вас возникло желание написать подпрограмму, в которую вложено большое количество локальных подпрограмм, причём некоторые из них имеют значительный размер — скорее всего, это значит, что вам следует рассмотреть выделение всего этого кода в отдельный модуль.

2.9.3. Как управиться с секциями описаний

Стандарт Паскаля требует жёсткого порядка следования секций описаний, но, к счастью, существующие реализации этому требо-

ванию не следуют, позволяя располагать секции описаний в произвольном порядке и создавать больше одной секции любого типа — в частности, не обязательно ограничиваться только одной секцией описания переменных.

Всё это даёт возможность отличать настоящие глобальные переменные от переменных, которые нужны в основной программе, но к которым при этом не нужен доступ из подпрограмм. Например, если в главной программе присутствует цикл и для него нужна целочисленная *переменная цикла*, то описать такую переменную за неимением лучшего придётся в секции описаний переменных, относящейся ко всей программе, однако совершенно ясно, что никакого отношения к глобальным переменным она не имеет. В связи с этим будет лучше, если **всё, что нужно в главной программе и только в ней — переменные, метки, иногда типы, которые не используются нигде, кроме главной программы — следует описать непосредственно перед словом `begin`, обозначающим начало головной программы, т. е. после всех описаний подпрограмм**. Если вам нужны настоящие глобальные переменные — то есть такие, доступ к которым осуществляется из более чем одной подпрограммы, либо из главной программы и подпрограммы — то для их описаний следует создать ещё одну секцию `var`, на этот раз *перед* описаниями подпрограмм.

Заметим, что переход между частями программы по меткам невозможен, так что **метки, используемые в подпрограммах, должны в них же и описываться**, тогда как метки, описываемые в «глобальной» секции описания меток, должны предназначаться для главной программы. Следовательно, **если в главной программе используются метки, секция их описаний должна находиться непосредственно перед началом главной программы**.

2.10. Особенности языка Си

2.10.1. Соглашения об именах

В отличие от Паскаля, язык Си чувствителен к регистру букв, так что никакой свободы в написании ключевых слов здесь не обнаруживается: например, `if` — это название оператора, тогда как `If`, `iF` и `IF` — это обыкновенные идентификаторы, притом различные.

Как и для Паскаля, для языка Си можно указать определённые традиции, сложившиеся вокруг именования объектов программы, и эти традиции оказываются совершенно иными. Идентификаторы, состоящие из нескольких слов, набира-

ют обычно полностью на нижнем регистре, разделяя слова знаком подчёркивания: `counter`, `summa`, `strange_global_variable`, `list_ptr`, `user_list_item`, `excellent_function`, `kill_them_all`, `produce_some_complete_mess` и тому подобное. Локальные переменные обычно обозначают короткими именами, такими как `i`, `ch`, `tmp`, `str` и т. п.

Как известно, макропроцессор языка Си отрабатывает после лексического анализа, но до анализа синтаксического (и, естественно, семантического). Как следствие, идентификаторы макросов (макроимена) в языке Си не подчиняются областям видимости, что делает их *опасными*. Поэтому программисты, работающие на Си, обычно следуют традиции, по которой идентификаторы макросов набираются целиком на верхнем регистре (большими буквами). Естественно, в имена макросов могут входить также символ подчёркивания и, при необходимости, цифры:

```
#define VERY_DANGEROUS_MACRO 756
```

Идентификаторы, сочетающие в себе буквы обоих регистров, при работе на языке Си обычно не используются. Если вы увидели программу на Си, в которой имеются идентификаторы «смешанного регистра», такие как `MixedCase`, `isItGood`, `CamelIsAnAnimal`, `exGF` и прочее в таком духе — скорее всего, автор такой программы редко пишет на Си, являясь поклонником какого-нибудь другого языка. Отметим, что сказанное верно лишь для чистого Си; традиции языка Си++ совершенно иные, и к ним мы ещё вернёмся.

Иногда можно встретить, в том числе в системных библиотеках и заголовочных файлах, применение «смешанного регистра» для именования структурных типов, а в некоторых случаях — и для других целей. Было бы неправильно утверждать, что это *недопустимо*, коль скоро ясности программ такой стиль не вредит. Тем не менее, мы не рекомендуем использовать такие отступления от традиционного именования.

Несколько особняком стоят идентификаторы, имена которых начинаются с подчёркивания. Традиционно считается, что эти имена *зарезервированы за системой программирования*, то есть, например, разработчики системных заголовочных файлов могут использовать такие имена для своих внутренних нужд, не внося их в документацию. В связи с этим не следует пользоваться именами, начинающимися с подчёркивания, в своих программах: такие имена (например, `_counter`, `_error_code`, `__LINES_NUM` и т. п.) могут вступить в конфликт с заголовочными файлами, поставляемыми вместе с компилятором. Следует учитывать, что, даже если одно конкретное имя ни к каким конфликтам не привело, это не является основанием для дальнейшего его использования: одним из основных достоинств языка Си

является *переносимость программ*, а использование потенциально «конфликтных» имён эту переносимость, естественно, снижает, ведь если в вашей конкретной версии библиотечных заголовочников соответствующего имени нет, то это не значит, что такое же имя не появится при работе с другим компилятором, на другой платформе, да и просто в одной из будущих версий вашей системы программирования. Итак, **не начинайте имена ваших идентификаторов с символа подчёркивания**, оставьте такое именование системным заголовочным файлам.

Отдельного рассмотрения заслуживают идентификаторы элементов перечислимого типа (`enum`). На первый взгляд в них нет ровным счётом ничего особенного, это обыкновенные идентификаторы. Отметим, что и *на второй взгляд* они тоже являются обычными идентификаторами, так что, если вы используете `enum`'ы по назначению — для обозначения одной ситуации из предопределённого множества — то идентификаторы можно (и, пожалуй, нужно) писать, как обычно, на нижнем регистре с использованием подчёркивания. Есть, однако, один способ использования `enum`'ов, вносящий в происходящее определённые коррективы: в какой-то момент программисты сообразили, что перечислимый тип (а точнее, вводимые им идентификаторы-значения) можно использовать в качестве обычных целочисленных констант. Например, вместо приведённого выше `VERY_DANGEROUS_MACRO` можно было бы ввести совершенно ничем не опасную константу:

```
enum { absolutely_safe_constant = 756 };
```

Конечно, `enum` здесь используется не по назначению, но для языка Си такая ситуация вполне типична, можно привести и другие примеры хаков подобного рода. Так или иначе, идентификатор, подчиняющийся областям видимости, в любом случае лучше, нежели такой идентификатор, который ведёт себя как слон в посудной лавке (а именно таковы макросы в Си), поэтому метод введения констант как идентификаторов для перечислимых типов получил широкое распространение. Здесь всё ещё как будто бы нет повода для беспокойства, но он появится, как только мы сделаем следующий — вполне логичный — шаг: решим переделать таким способом все уже имеющиеся константы, которые были ранее объявлены с помощью `#define` и в соответствии с вышеописанными соглашениями названы именами, состоящими из заглавных букв. Возможно, *некоторые* из них мы даже переименуем. Но рано или поздно мы столкнёмся со случаем, когда переименование константы недопустимо: например, константа является частью интерфейса популярной библиотеки, включена в документацию и используется в паре десятков (а то и

в паре десятков тысяч) программ. Итак, мы обнаружили константу, которую нельзя переименовать; что же теперь, так и оставить её в виде макроса? Чаще всего, что характерно, именно так и поступают, в современных программах на Си константы, введённые `#define`'ом, всё ещё встречаются чаще, нежели `enum`'овые. Но, если подумать, валидных причин для такого решения попросту нет: в самом деле, ну *чем хуже* станет код, если очередной макрос заменить на безопасную константу с тем же именем? Результатом становится массовое использование в качестве имён констант перечислимого типа идентификаторов, более привычных в роли макроимён, то есть состоящих из заглавных латинских букв:

```
enum { VERY_DANGEROUS_MACRO = 756 };
```

Здесь наша константа уже и не dangerous, и не macro, но если переименовать её мы не смогли, то что же теперь поделать?

К сожалению, при этом несколько нарушается единообразие имён идентификаторов: одни константы именуются на нижнем регистре, а другие на верхнем. Поэтому в некоторых стилях оформления кода на Си присутствует требование именовать с использованием букв верхнего регистра *любые* константы, в том числе и такие, которые никогда не были макросами:

```
enum { ABSOLUTELY_SAFE_CONSTANT = 756 };
```

Как это часто бывает, окончательного ответа на вопрос, правильно это или нет, мы дать не сможем. Если вас пригласили в существующий проект, следуйте имеющимся соглашениям; если вы начинаете проект с нуля, подумайте сами, как вам будет удобнее именовать константы.

Исторически такому именованию констант перечислимого типа поспособствовали Керниган и Ритчи. Из текста их книги очень заметно, что многие нововведения, проникшие в язык из расширений, самовольно введённых авторами различных компиляторов, им не понравились. Видимо, это касается и перечислимых типов. Изначально в Си был только один способ введения именованных констант — директива `#define`, а сами константы могли быть только макросами и более ничем; естественно, их писали на верхнем регистре. Снизойдя в последних изданиях книги до описания перечислимого типа, Керниган и Ритчи тем не менее продолжили все константы, включая введённые `enum`'ом, называть именами верхнего регистра. Никакой логики в этом нет, константы как таковые не отличаются от других сущностей ничем столь принципиальным, чтобы их зачем-то всегда выделять из текста программы и делать «заметными».

2.10.2. Описания и инициализаторы

В языке Си часто встречаются описания типов и переменных, имеющие достаточно сложную синтаксическую структуру.

Начнём с классической рекомендации: **не пренебрегайте использованием директивы `typedef` для особенно сложных описаний!** Так, профиль системного вызова `signal` с использованием `typedef` выглядит достаточно просто:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

То же самое можно написать и без `typedef`, и выглядеть это будет вот так:

```
void (*signal(int signum, void (*handler)(int)))(int);
```

Если кто-нибудь скажет вам, что это описание-де вполне понятно любому вменяемому программисту, не верьте: даже самые опытные программисты вынуждены *разбирать* такие описания, хотя некоторые (не все!) делают это довольно быстро. Отметим, что вызов `signal` представляет собой случай довольно простой; в качестве упражнения попробуйте написать функцию, которая принимает *на вход одним из параметров* функцию, имеющую такой же профиль, как у `signal`.

С описанием типа, предполагающим использование фигурных скобок (`struct`, `union`, `enum`) всё довольно просто: единственная степень свободы — положение открывающей фигурной скобки (её, как водится, можно оставить на одной строке с именем структуры, а можно снести на следующую строку). Чего точно не следует делать — это сдвигать фигурные скобки относительно начала описания типа, то есть вот так писать можно:

```
struct item {
    const char *str;
    item *next;
};
```

```
struct item
{
    const char *str;
    item *next;
};
```

а вот так уже не стоит:

```
struct item
    {
        const char *str;
        item *next;
    };
```

Ситуация здесь подобна ситуации с началом и концом тела функции: всё тело целиком не сдвигают даже при использовании стиля, предполагающего сдвиг составного оператора относительно заголовка сложного оператора.

Отметим, что не стоит также и «вытягивать» описание структуры в одну строку:

```
struct item { const char *str; item *next; };
```

А вот описание перечислимого типа «вытянуть» можно, и смотреться это будет вполне логично, но только в случае, если оно умещается в одну строку и не содержит явно заданных значений констант:

```
enum state { home, whitespace, stringconst, ident, end };
```

Если в одну строку описание не поместилось, или если имеется потребность в явном задании целочисленных значений[10], то лучше будет каждую константу описывать на отдельной строке. Обратите внимание, что для лучшей читаемости знаки равенства рекомендуется расположить в одну колонку (используйте пробелы, если пользуетесь ими для структурных отступов; если в качестве отступа у вас табуляция, используйте её же для выравнивания знаков равенства):

```
enum state {
    st_home,
    st_whitespace,
    st_stringconst,
    st_ident,
    st_end
};
```

```
enum state {
    st_home         = 0,
    st_whitespace   = 32,
    st_stringconst  = 12,
    st_ident        = 77,
    st_end          = -1
};
```

Как обычно, положение открывающей фигурной скобки определяется избранным стилем. Если вы сносите её в описаниях структур, сделайте то же самое и при описании `enum`'ов:

```
enum state
{
    st_home,
```

```
enum state
{
    st_home         = 0,
```

Аналогично обстоят дела с длинными инициализаторами (например, при описании инициализированных массивов). Если инициализатор полностью умещается в одну строку, лучше его так и записать:

```
const int some_primes[] = { 11, 17, 37, 67, 131, 257, 521 };
```

Если инициализатор в одну строку не поместился, или если к значениям требуются комментарии, или по каким-либо другим причинам, можно записать каждый элемент инициализатора на отдельной строке:

[10] Кроме случая описания анонимного псевдотипа с целью введения *одной* константы; этот случай рассмотрен в предыдущем параграфе.

```
const unsigned long hash_sizes[] = {
    11,     /* > 8 */
    17,     /* > 16 */
    37,     /* > 32 */
    67,     /* > 64 */
    131,    /* > 128 */
    257,    /* > 256 */
    521,    /* > 512 */
    1031,   /* > 1024 */
    2053    /* > 2048 */
};
```

Более того, нет ничего страшного в разбиении инициализатора на несколько строк из соображений равномерности:

```
const unsigned long hash_sizes[] = {
    11,       17,       37,       67,       131,     257,
    521,      1031,     2053,     4099,     8209,    16411,
    32771,    65537,    131101,   262147,   524309,  1048583,
    2097169,  4194319,  8388617,  16777259, 33554467
};
```

Положение открывающей фигурной скобки в этих случаях также зависит от избранного стиля, причём, как ни странно, часто её сдвигают, то есть следующий вариант не только допустим, но и довольно популярен:

```
const unsigned long hash_sizes[] =
    {
        11,     /* > 8 */
        17,     /* > 16 */
        37,     /* > 32 */
        67,     /* > 64 */
        131,    /* > 128 */
        257,    /* > 256 */
        521     /* > 512 */
        1031,   /* > 1024 */
        2053    /* > 2048 */
    };
```

При инициализации двумерного массива обычно каждый элемент (то есть одномерный массив) стараются записать в одну строку, например:

```
const int change_level_table[5][5] = {
    { 4, 4, 2, 1, 1 },
    { 3, 4, 3, 1, 1 },
    { 1, 3, 4, 3, 1 },
```

```
    { 1, 1, 3, 4, 3 },
    { 1, 1, 2, 4, 4 }
};
```

Если же в одну строчку кода каждая строка вашей матрицы не помещается (случай сам по себе довольно редкий), придётся их равномерно разбить на несколько строчек.

2.10.3. Оператор постусловия

В языке Си ключевое слово `while` используется в двух ролях: в заголовке цикла с предусловием и в эпилоге цикла с постусловием (`do-while`). В обоих случаях сразу после слова `while` следует условное выражение в круглых скобках; конечно, в случае `do-while` следом за этим выражением идёт точка с запятой, но для случая цикла с пустым телом такая же точка с запятой может быть и в случае обычного `while`. Итак, глядя на конец цикла `do-while`, можно (и очень легко) перепутать его с циклом `while`, имеющим пустое тело.

Решение этой проблемы довольно очевидно для стиля расстановки фигурных скобок, при котором открывающая скобка не сносится на следующую строку. Прежде всего отметим, что **использование фигурных скобок в цикле `do-while` строго обязательно вне всякой зависимости от количества операторов в этом теле**. Некоторые программисты даже пребывают в уверенности, что это требование языка Си; на самом деле это не так, телом `do-while` может быть любой оператор, не только составной; но эта возможность Си никогда не используется. Проблема со смыслом `while` теперь решается тем, что слово `while` попросту остаётся на той же строке, что и предшествующая ему закрывающая скобка, примерно так, как мы для этого же стиля поступали со словом `else`:

```
do {
    get_event(&event);
    res = handle_event(&event);
} while (res != EV_QUIT_NOW);
```

Такое же решение мы можем применить и для стиля, при котором открывающая фигурная скобка сносится на следующую строку, но не сдвигается:

```
do
{
    get_event(&event);
    res = handle_event(&event);
} while (res != EV_QUIT_NOW);
```

Конечно, такое решение не слишком изящно, поскольку вынуждает нас вводить исключение из общего правила, однако так всё же лучше, чем путать конец цикла с началом нового цикла. Но вот что делать при использовании стиля, где скобки не только сносятся, но и сдвигаются, оказывается непонятно. Ричард Столлман в GNU Coding Style Guide предлагает оформлять `do-while` так:

```
do
  {
    get_event(&event);
    res = handle_event(&event);
  }
while (res != EV_QUIT_NOW);
```

При всём уважении к Столлману эта идея крайне неудачна. Если цикл занимает хотя бы десяток строк, при взгляде на слово `while` мы совершенно однозначно не заметим соответствующее ему `do`, чему способствует в немалой степени то обстоятельство, что циклы с предусловием встречаются гораздо чаще, нежели циклы с постусловием. После этого мы примем `while` за заголовок цикла, дальше, возможно, не заметим точку с запятой, а возможно, наоборот, заметим, решим, что это ошибка, «споткнёмся», после чего, просматривая фрагмент уже более внимательным взглядом, найдём, наконец, проклятое `do`, при этом потратив не меньше секунды на размышления в неправильном направлении и ещё две-три секунды, чтобы вернуться к тем мыслям, из которых нас выбил пресловутый `while`. Можно определённо сказать, что такие `while`'ы обходятся читателю программы слишком дорого. Поэтому, сколь бы противно сие ни выглядело, мы возьмём на себя смелость рекомендовать что-то вроде следующего:

```
do
  {
    get_event(&event);
    res = handle_event(&event);
  } while (res != EV_QUIT_NOW);
```

Отметим, что одной этой сложности может быть вполне достаточно, чтобы отказаться от такого стиля расстановки операторных скобок, по крайней мере, для языка Си; для Паскаля этот аргумент не действует, там такой проблемы нет.

2.10.4. О модулях и слове `static`

Язык Си можно назвать модульным только с большой натяжкой; разделить программу на Си на отдельные единицы трансляции, конечно, можно, но в самом языке средства поддержки модульности

практически отсутствуют. Применяемая техника заголовочных файлов с защитой от повторного включения не является частью языка Си, это скорее один из удачных хаков. Разделение по модулям ещё не означает локализации имён; по умолчанию все нелокальные имена в модулях доступны из других модулей: даже если не включить, например, некую глобальную переменную в заголовочник, на уровне редактора связей она всё равно будет видна, и где-то в другом месте вполне может оказаться директива `extern`. С функциями ситуация ещё хуже: видя вызов функции с незнакомым именем, компилятор выдаёт лишь предупреждение, продолжая компиляцию. Это означает, что, не вынеся заголовок функции в заголовочный файл, мы на самом деле ничего не добились.

Как обычно, если существует трудность, то находится и способ её устранения. Программисты обычно следуют определённой традиции в отношении идентификаторов, описанных в глобальной области видимости (функций и глобальных переменных):

- **для каждого модуля, кроме модуля, содержащего функцию `main`, следует предусмотреть одноимённый заголовочный файл, причём этот файл в обязательном порядке подключается к модулю;**

- **для каждой функции, а равно для каждой нелокальной переменной, есть лишь две возможности: либо её заголовок (для переменных — объявление со словом `extern`) должен быть вынесен в заголовочный файл, либо описание такой переменной или функции должно начинаться со слова `static`.**

Единственным исключением из этого правила является функция `main`; заголовочного файла для её модуля обычно нет, а объявлять её как `static` нельзя. Впрочем, зачастую программисты предпочитают это исключение обойти, включив в начало головного модуля прототип функции `main` и уже потом описав её саму. Это позволяет, например, при использовании компилятора `gcc` применить ключик `-Wstrict-prototypes`, заставляющий компилятор выдавать предупреждение о каждой функции, которая не объявлена как `static`, но при этом не имеет прототипа.

2.10.5. Характерные ошибки в оформлении функции

Достаточно часто можно видеть программы, в которых описания локальных переменных в функциях, а также заключительный опе-

ратор `return` почему-то не сдвинуты на размер отступа, а написаны с крайней левой позиции экрана, примерно так:

```
int main()
{
int i;
const char *hello = "Hello";
const char *goodbye = "Good Bye";
    for (i = 0; i < 10; i++) {
        printf("%s\n", hello);
    }
    for (i = 0; i < 10; i++) {
        printf("%s\n", goodbye);
    }
return 0;
}
```

Такой стиль, разумеется, недопустим. Прежде всего, оператор `return` — это обычный оператор, встречаться он может не только в самом конце функции, но и в её середине, будучи при этом вложенным в циклы и ветвления; нет совершенно никаких оснований считать, что случай возврата значения в последней строчке функции чем-то принципиальным отличается от других ситуаций, когда `return` встречается в коде.

Что касается переменных, то, конечно, их описания *операторами* не являются[11], но при таком стиле форматирования фигурная скобка, открывающая функцию, начинает сливаться с окружающим кодом; совершенно неясно, какого преимущества хотят добиться сторонники такого странного форматирования.

2.11. Особенности Си++

2.11.1. Соглашения об именах

Первоначально в Си++ действовали те же соглашения об именах, которые привычны программистам на чистом Си: макросы именовались большими буквами, всё остальное — маленькими. Следы этого времени всё ещё сохранились в именовании типов и прочих сущностей стандартной библиотеки: все стандартные классы, шаблоны и т. п. поименованы с использованием букв нижнего регистра. Со временем, однако, традиции несколько изменились. Те сущности, которые не затрагивают отличий Си++ от чистого Си — обычные переменные, функции, которые не являются членами классов и т. п. —

[11] Речь идёт о чистом Си; в языке Си++ описание переменной является оператором.

по-прежнему называют, как и в чистом Си, идентификаторами, состоящими из маленьких букв и подчёркиваний, в некоторых случаях добавляют цифры, ну а макросы — ровно по тем же причинам, что и в Си — именуют полностью большими буквами, чтобы их было хорошо видно. Надо сказать, что макросы в Си++ сильно не в почёте; единственное их использование, от которого невозможно отойти — это защита от повторного включения заголовочных файлов, и вообще условная компиляция в широком смысле, во всех же остальных случаях можно (и нужно!) использовать шаблоны, константы, inline-функции и прочие возможности, не завязанные на макропроцессор. Даже для обозначения нулевого указателя рекомендуется использовать арифметический `0`, а не макрос `NULL`, как в чистом Си.

Что же касается имён, связанных с нововведениями Си++ — классами и их методами (функциями-членами), то для них чаще всего используют «смешанный регистр», то есть в имени сочетаются тем или иным способом буквы верхнего и нижнего регистра. Так, в известной книге Гради Буча в именах классов каждое слово начинается с большой буквы, а дальше идут маленькие, например `MyGoodClass`, `StrangeThing`, `Complex` и т. п., тогда как для именования методов (функций-членов) используется чуть более сложная нотация: имя метода должно состоять не менее чем из двух слов, причём первое слово пишут полностью маленькими буквами, а все остальные — каждое с большой буквы, примерно так: `closeFile`, `isReady`, `doSomethingUseful`, `gameScore`, `getXCoord`, `setColor`... Поля классов и структур (члены-данные) по-прежнему называют маленькими буквами с использованием подчёркивания для разделения слов, тогда как в именах классов и методов подчёркивание не используется вообще.

Естественно, такой стиль — не единственный возможный, к тому же можно с ходу назвать определённые недостатки, связанные с ним: конструкторы и деструкторы класса тоже являются методами, но называться они обязаны так же, как и класс, в результате получается, что не все методы называются единообразно. Кроме того, не всегда удобно нижнее ограничение на количество слов в имени: часто возникает желание назвать метод *одним словом*, но сделать этого нельзя, поскольку хотя бы одна заглавная буква в имени должна присутствовать. Поэтому часто можно встретить программы на Си++, где соглашение «каждое слово с большой буквы» используется как для классов, так и для методов.

2.11.2. Класс или структура?

Как известно, формально классы в Си++ отличаются только моделью защиты по умолчанию: структуры исходно открыты, тогда

как классы исходно закрыты. В остальном классы и структуры абсолютно равноправны: и те, и другие могут иметь функции-члены (в том числе и виртуальные), а могут их не иметь, и те, и другие могут участвовать в наследовании как в качестве предков, так и в качестве потомков, и их даже можно наследовать друг от друга — класс от структуры и структуру от класса. Тем не менее, программисты обычно следуют определённой традиции: для создания «объектов» в смысле объектно-ориентированного программирования, а также абстрактных типов данных, относящихся к одноимённой парадигме, используются классы, тогда как для создания открытых структур данных, находящихся под управлением чего-то по отношению к ним внешнего, используют структуры. Например, при построении списка или дерева следует использовать структуры в качестве элементов/узлов, а весь список или дерево как единое целое можно «обернуть» в объект, описанный с помощью класса; впрочем, совершенно не факт, что это делать *нужно*, контейнерные классы — явление сомнительное.

Сказанное не означает, что в структурах нельзя описывать методы; напротив, при работе с динамическими данными может оказаться очень удобно снабдить структуру конструктором и деструктором, а в некоторых случаях — и другими методами.

Отметим, что для именования структур чаще всего применяют идентификаторы, традиционные для чистого Си (все слова маленькими буквами, через подчёркивания).

2.11.3. Форматирование заголовков классов

Пока речь идёт о структурах в смысле языка Си, их оформление оказывается довольно просто и ничем не отличается от обсуждавшегося выше для чистого Си (см. § 2.10.2). Но вот когда дело доходит до классов или до структур, имеющих закрытую часть, если у вас такие предусмотрены, мы сталкиваемся с дополнительной сложностью из-за директив `public:`, `private:` и `protected:`. Вариантов с ними ровно два: либо сдвигать их на дополнительный размер отступа, либо не сдвигать. Вообще говоря, оба следующих фрагмента вполне корректны и допустимы:

```
class MyInteger {
        int x;
    public:
        MyInteger(int ax = 0);
        operator int() const;
};
```

```
class MyInteger {
    int x;
public:
    MyInteger(int ax = 0);
    operator int() const;
};
```

Тем не менее, первый вариант обычно встречается только в программах начинающих, а программисты более опытные используют

второй вариант, не предполагающий дополнительного сдвига. В качестве одной из проблем первого варианта можно назвать то, что описания, идущие в начале заголовка класса, т. е. сразу после открывающей фигурной скобки, оказываются сдвинуты на *два* размера отступа. Начинающие иногда пытаются бороться с этим примерно так:

```
class MyInteger {
    int x;
    public:
        MyInteger(int ax = 0);
        operator int() const;
};
```

но вот это уже совсем никуда не годится, потому что описания членов класса, которые имеют, очевидно, *одинаковый* ранг вложенности (ведь они вложены в класс, и более ни во что), оказываются сдвинуты на *разный* размер сдвига.

2.11.4. Форматирование заголовка конструктора

Заголовок конструктора отличается от заголовка обычной функции наличием списка инициализаторов; если списка инициализаторов нет, то никаких особенностей конструктор по сравнению с другими функциями не имеет. Но вот если этот список есть, возникает, во-первых, вопрос, куда его поместить, и, во-вторых, что делать, если он не влез на одну строку.

Прежде всего отметим, что обычно список инициализаторов сносят на отдельную строку (кроме случая, когда конструктор описывается непосредственно в заголовке класса, но об этом позже), причём чаще всего — вместе с двоеточием. Эстетичнее смотрится вариант со сдвигом этой строки на размер отступа, например:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    : MyBaseClass(s), int_field(x)
{
    // here the body comes
}
```

Но можно, в принципе, написать и вот так:

```
MyGoodClass::MyGoodClass(int x, const char *s)
: MyBaseClass(s), int_field(x)
{
    // here the body comes
}
```

Иногда встречается и вот такое форматирование:

```
MyGoodClass::MyGoodClass(int x, const char *s) :
    MyBaseClass(s),
    int_field(x)
{
```

Надо отметить, что с размещением двоеточия возникают наиболее неоднозначные вариации. Можно встретить и вот такое:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    : MyBaseClass(s),
    int_field(x)
{
```

и вот такое:

```
MyGoodClass::MyGoodClass(int x, const char *s)
:
    MyBaseClass(s),
    int_field(x)
{
```

и вот такое:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    :
    MyBaseClass(s),
    int_field(x)
{
```

Сразу хотелось бы предостеречь читателя от одного варианта, который, хотя и допустим (кроме случая использования табуляции для отступов), но всё же не слишком правилен, потому что очередная строка оказывается сдвинута на непонятно какой уровень отступа:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    : MyBaseClass(s),
      int_field(x)
{
```

Довольно удачным оказывается решение, странновато выглядящее на первый взгляд, когда запятую записывают не *после* инициализатора, а *перед следующим*, что позволяет представить двоеточие и запятые как своего рода «символ начала инициализатора» и расставить их в столбик:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    : MyBaseClass(s)
    , int_field(x)
    , another_field(s)
{
```

Отдельного рассмотрения заслуживает случай, когда решено инициализаторы записывать в строчку, но при этом список инициализаторов не умещается на одной строке. Для этого случая можно порекомендовать разорвать строку, по необходимости в нескольких местах; выбрать конкретные места разрыва можно либо из соображений равномерности, либо исходя из некой «группировочной» логики (например, в первой строке записать инициализаторы базовых классов, а инициализаторы полей снести на следующую сроку). Необходимо только помнить, что инициализаторы обязаны перечисляться в том же порядке, в котором в классе объявлены соответствующие поля. Расположить такие строки, а также знак двоеточия можно любым из способов, показанных выше для «столбика» инициализаторов. Например:

```
MyGoodClass::MyGoodClass(int x, const char *s)
    : MyBaseClass(s), int_field(x), first(0), last(0),
    sequence_counter(1), the_collection(0),
    helper(new MyHelperClass(*this))
{
```

2.11.5. Тела функций в заголовке класса

Тело функции-члена класса можно вынести за пределы заголовка класса, но можно написать и в самом заголовке; компилятор попытается (но не обязан) обработать такие функции как инлайновые.

Прежде всего отметим, что пользоваться этой возможностью следует очень осторожно. Важнейшая роль описания класса состоит в том, чтобы при взгляде на него можно было понять, как следует работать с объектами такого класса, то есть каков его набор методов и как ими следует пользоваться. Для этого желательно (если не сказать необходимо), чтобы описание класса оставалось компактным, таким, чтобы охватить его можно было одним взглядом. Ясно, что наличие тел функций-членов отнюдь не способствует лаконичности описания класса. Кроме того, если класс не является внутренним для отдельно взятого модуля, то его заголовок необходимо вынести в заголовочный файл, который с помощью директивы `#include` (то есть текстуально) будет включаться в другие модули; как следствие, объектный код всех функций-членов, тела которых присутствуют в

заголовке класса, будет присутствовать в каждом из таких модулей, что далеко не лучшим образом сказывается как на размере исполняемого файла, так и на скорости компиляции проекта. Всё это позволяет сформулировать довольно простое правило: **в заголовке класса могут присутствовать тела функций-членов, весь код которых состоит из одной строки, реже — из двух, в исключительных случаях — из трёх строк**; описания функций-членов бóльшего объёма следует выносить за пределы описания класса.

Остаётся вопрос, как оформить тело функции-члена, если таковое всё же решено оставить в заголовке класса, и здесь необходимо заметить, что лаконичность описания класса как целого — это самостоятельная цель, ради достижения которой имеет смысл пожертвовать некоторым другими принципами. Так, если тело функции состоит из одной строки, то часто такое тело записывают на одной строке с её заголовком (особенно часто такое встречается для так называемых аксессоров):

```
class Picture {
    // ...
    int color;
public:
    // ...
    int GetColor() const { return color; }
    void SetColor(int col) { color = col; }
    // ...
};
```

Более того, если функция-член состоит из более чем одного оператора, но все эти операторы способны уместиться на одной строке, программисты часто идут и на это (хотя в обычных условиях так писать ни в коем случае не следует):

```
class FileStream {
    // ...
    void SetFd(int f) { if (fd != -1) close(fd); fd = f; }
    // ...
};
```

Если тело никак не желает помещаться в одну строку с заголовком, его можно (целиком) снести на следующую строку, по-прежнему записав одной строчкой. В этом случае открывающую фигурную скобку обычно сдвигают на размер отступа, хотя это, строго говоря, не обязательно:

```
class Table {
    // ...
```

```
    const char *GetNameById(int id) const
        { return ProvideRecordById(id)->GetName(); }
    // ...
};
```

Если всё-таки вы решили, что тело вашей функции будет лучше смотреться, будучи расписанным на несколько строк, то остаётся ещё возможность сэкономить одну строку, разместив открывающую фигурную скобку на одной строке с заголовком функции (отметим, что это тоже специфика ситуации тела функции в заголовке класса; в иных случаях так не делают):

```
class FileStream {
    // ...
    void SetFd(int f) {
        if (fd != -1)
           close(fd);
        fd = f;
    }
    // ...
};
```

Чего точно не следует делать, так это приносить лаконичность заголовка класса в жертву непонятным стилистическим канонам. В частности, если вы не нашли ничего лучшего, нежели написать вот так:

```
class FileStream {
    // ...
    void SetFd(int f)
        {
            if (fd != -1)
               close(fd);
            fd = f;
        }
    // ...
};
```

то будет гораздо правильнее всё-таки убрать тело функции из заголовка класса. При этом вы всё ещё можете объяснить компилятору, что функцию желательно сделать инлайновой:

```
class FileStream {
    // ...
    void SetFd(int f);
    // ...
};
inline void FileStream::SetFd(int f)
```

```
{
    if (fd != -1)
        close(fd);
    fd = f;
}
```

Нужно только помнить, что инлайновая функция — это на самом деле не функция, а подставляемый фрагмент кода, и подстановку, естественно, производит компилятор, а не линкер; как следствие, **тело инлайн-функции, экспортируемой из модуля, должно находиться в заголовочном файле** наравне с макросами, объявлениями глобальных переменных и обычных функций и с описаниями типов (в том числе заголовков классов).

Глава 3

О некоторых языках «с особенностями»

3.1. Язык ассемблера

Языки программирования, в которых строка исходного кода представляет собой основную синтаксическую единицу, в наше время встречаются редко; большинство языков рассматривает конец строки как один из пробельных символов, ничем принципиальным не отличающийся от пробела и табуляции. Обычно имеют построчный синтаксис языки ассемблера; что касается языков высокого уровня, то из используемых ныне можно назвать разве что Фортран. Интересно, что именно Фортран (точнее, его ранние версии) заложил определённую традицию оформления кода, которая сейчас используется для языка ассемблера; что касается самого Фортрана, то для него в последние годы популярен так называемый свободный синтаксис, позволяющий использовать традиционные структурные отступы; встречаются, впрочем, и такие программисты, которые предпочитают писать на Фортране «по-старинке».

Традиция «фиксированного синтаксиса» восходит к тем временам, когда программа на Фортране представляла собой колоду перфокарт. Ранние версии Фортрана требовали, чтобы метка, которая в Фортране представляет собой целое число, записывалась в столбцах с 1-го по 5-й, причём часто метку короче пяти цифр сдвигали вправо, оставляя первые позиции пустыми. В первой позиции можно было также поместить символ `C`, обозначающий комментарий; позже комментарий стало можно обозначать символами `*` или `!`. Текст оператора должен был начинаться строго с 7-й позиции, а перед ним в 6-й позиции необходимо было поместить пробел; непробельный символ

в 6-й позиции означал, что эта строка (точнее, перфокарта) является продолжением предыдущей, при этом пустыми должны были остаться первые пять позиций.

Современный текст на языке ассемблера обычно несколько напоминает программы на ранних версиях Фортрана. Общий принцип оформления кода на языке ассемблера довольно прост. Следует мысленно разделить горизонтальное пространство экрана на две части: область меток и область команд (операторов). Часто выделяют также область комментариев. Код пишется «в столбик» примерно так:

```
        xor ebx, ebx        ; zero ebx
        xor ecx, ecx        ; zero ecx
lp:     mov bl, [esi+ecx]   ; another byte from the string
        cmp bl, 0           ; is the string over?
        je lpquit           ; end the loop if so
        push ebx            ;
        inc ecx             ; next index
        jmp lp              ; repeat the loop
lpquit: jecxz done          ; finish if the string is empty
        mov edi, esi        ; point to the buffer's begin
lp2:    pop ebx             ; get a char
        mov [edi], bl       ; store the char
        inc edi             ; next address
        loop lp2            ; repeat ecx times
done:
```

Если метка не помещается в отведённое для меток пространство, её располагают на отдельной строке:

```
fill_memory:
        jecxz fm_q
fm_lp:  mov [edi], al
        inc edi
        loop fm_lp
fm_q:   ret
```

Обычно при работе на языке ассемблера под метки выделяют столбец шириной в одну табуляцию и, естественно, именно символ табуляции (а не пробелы!) ставят перед каждой командой (в том числе и после меток). Некоторые программисты предпочитают отдать под метки две табуляции; это позволяет использовать более длинные метки без выделения для них отдельных строк:

```
fill_memory:    jecxz fm_q
fm_lp:          mov [edi], al
                inc edi
                loop fm_lp
fm_q:           ret
```

Довольно часто можно встретить стиль оформления, предполагающий отдельную колонку для обозначения команды. Выглядит это примерно так:

```
fill_memory:    jecxz   fm_q
fm_lp:          mov     [edi], al
                inc     edi
                loop    fm_lp
fm_q:           ret
```

К сожалению, этот стиль «ломается» при использовании, например, макросов с именами длиннее семи символов, поскольку имя такого макроса при макровызове следует записать, что вполне естественно, в колонку команды, но пространства в этой колонке не хватает. Впрочем, для макросов можно сделать исключение из правил.

Как можно заметить, никакие структурные отступы здесь не используются, что можно довольно легко объяснить: программа на языке ассемблера представляет собой прообраз содержимого оперативной памяти, которая, согласно принципам фон Неймана, линейна и однородна. Для выделения управляющих конструкций остаются только комментарии, и уж их следует использовать «на всю катушку», если только вы не хотите в конечном результате получить текст, в котором сами никогда не разберётесь.

3.2. Лисп и его диалекты

Язык Лисп знаменит своим синтаксисом, а точнее — полным отсутствием такового. Программа состоит из *списков*, как, собственно, и большая часть возможных данных, ну а список в Лиспе — это открывающая круглая скобка, произвольное количество *элементов* через пробел и закрывающая круглая скобка; элементами могут быть как атомарные выражения любого типа, так и, в свою очередь, списки, и так на произвольную глубину. Такие гетерогенные структуры данных называются *S-выражениями*.

Конечно, при создании программ на Лиспе приходится применять структурные отступы, иначе, пожалуй, в этих скоплениях скобок разобраться было бы совершенно невозможно. При этом вызывает некоторое удивление стиль отступов, «исторически сложившийся» вокруг Лиспа. Рассмотрим для примера типичную функцию на Лиспе:

```
(defun isomorphic (tree1 tree2)
       (cond ((atom tree1) (atom tree2))
             ((atom tree2) NIL)
             (t (and (isomorphic (car tree1) (car tree2))
```

```
                (isomorphic (cdr tree1) (cdr tree2))))))
```

Обращают на себя внимание два важных момента. Во-первых, размер отступа явно «плавает»: форма `cond` сдвинута относительно объемлющей формы на семь пробелов, отдельные предложения этого `cond`'а сдвинуты уже на шесть пробелов, а второй аргумент `and` сдвинут на восемь пробелов правее предыдущего уровня отступа. Обусловлено это главенствующим подходом к форматированию списка: если список умещается в строку, то его записывают на одной строке, если же он не умещается, то на одной строке пишут его начало — первый элемент, который чаще представляет собой имя функции или формы, и второй элемент, то есть первый аргумент функции или формы; остальные аргументы располагают под первым «в столбик»[1]. Если первый аргумент не поместился в строку, его разбивают аналогичным образом, причём его начало на следующую строку не переносят; именно так в приведённом примере получилось с вызовом `and` после символа `t`. В результате размер сдвига определяется длиной имени символа, стоящего в форме первым: слово `defun` вместе со скобкой и пробелом даёт семь символов, отсюда семь пробелов на первом уровне отступа, `cond` вместе со скобкой и пробелом даёт шесть символов — это размер второго уровня отступа, ну а `t`, `and`, две скобки и два пробела — это искомые восемь пробелов на третьем уровне.

Вторая характерная особенность, показанная в примере — это группа из шести закрывающих скобок в конце. Почему поклонники Лиспа предпочитают закрывать вложенные списки именно таким вот образом, не вполне понятно, ведь скобки приходится *считать*, они сливаются друг с другом, очевидно становясь источником труднообнаружимых ошибок. Тем не менее, такой стиль настолько популярен, что существуют даже диалекты Лиспа, в которых есть специальный символ, обозначающий «столько закрывающих скобок, сколько есть незакрытых списков»; так, в диалекте muLISP можно было записать наш пример следующим образом:

```
(defun isomorphic (lambda (tree1 tree2)
       (cond ((atom tree1) (atom tree2))
             ((atom tree2) NIL)
             (t (and (isomorphic (car tree1) (car tree2))
                     (isomorphic (cdr tree1) (cdr tree2]
```

[1] Форма `defun` представляет собой одно из нескольких исключений из общего правила: здесь стараются на одной строке разместить само имя формы, затем имя функции и список параметров, а тело функции обычно сносят на следующую строку, даже если оно уместилось бы в одной строке с «заголовком».

Возможность закрыть сразу все открытые скобки удобна лишь на первый взгляд: используя её, мы сами себя лишаем средства проверки корректности структуры нашего списка.

Между тем никто не мешает писать на Лиспе «по-паскалевски», с постоянным размером отступа и расположением закрывающей скобки точно под началом соответствующей конструкции; нужно только разрывать список, не поместившийся в строку, не после *второго* элемента, а после первого. Выглядеть это будет примерно так (здесь мы применяем отступ на три пробела):

```
(defun isomorphic (tree1 tree2)
   (cond
      ((atom tree1) (atom tree2))
      ((atom tree2) NIL)
      (t
         (and
            (isomorphic (car tree1) (car tree2))
            (isomorphic (cdr tree1) (cdr tree2))
         )
      )
   )
)
```

Конечно, такой код занимает больше строк, но это окупается ясностью его структуры. В частности, расположение нескольких идущих подряд закрывающих круглых скобок вдоль диагонали экрана показывает, что баланс скобок, скорее всего, соблюдён, тогда как нарушение такого расположения однозначно свидетельствует о том, что где-то мы со скобками запутались.

В большинстве случаев синтаксическая структура в Лиспе — это просто список, который начинается и заканчивается скобкой, так что открывающую и закрывающую круглые скобки приходится располагать точно друг над другом (конечно, если они не остались на одной строке). Из этого правила есть несколько исключений, связанных с сокращённой записью квотирования (апостроф), функционального квотирования (комбинация «#'»), с синтаксисом вектора (символ «#»), полуквотированием (semiquotation) и его составляющими, которые используются в макросах, и так далее. Во всех этих случаях перед открывающей скобкой оказывается один или два символа, задающих иной смысл всей конструкции; при этом получается, что открывающая скобка — *не первый* символ конструкции, но закрывающая — по-прежнему *последний* символ. Естественно, если такая конструкция оказалась разнесена на несколько строк, следует располагать закрывающую скобку точно под *началом конструкции*, а не под открывающей скобкой:

```
(mapcar
    #'(lambda (str elem)
        (list ('rec str elem 'endrec))
    )
    list
    cycled-labels
)
```

В этом примере также видно, что lambda-список представляет собой ещё одно исключение из правил — его оформление напоминает оформление формы `defun`; источник такого исключения вполне понятен, ведь `lambda`-список — это безымянная функция, а для функции нам привычнее видеть заголовок, в который включён список параметров.

Ещё один важный особый случай, заслуживающий внимания — это форма `let`, а также всевозможные `let*`, `letrec`, `funlet`, `macrolet` и т. п. Если список локальных переменных, вводимых с помощью `let`, умещается на одной строке с самим словом `let`, следует там его и оставить:

```
(let ((s 0) (p 1))
   (mapcar
      #'(lambda (x) (setq s (+ s x)) (setq p (* p x)))
      lst
   )
)
```

Ситуация резко осложняется, если первый элемент формы `let` приходится расположить на нескольких строках, а такое требуется очень часто — достаточно в одном из инициализаторов появиться сложному выражению. «Правильного» решения для этой ситуации просто нет, все существующие варианты так или иначе оказываются плохи. Программисты в таких случаях изобретают самые разнообразные конструкции, например:

```
(let ((seq (build-sequencer 0))
      (colset '(red orange yellow green blue violet))
      (ls nil))
   (setq ls (mapcar #'list seq colset the-list))
   (store ls)
   (reverse ls))
```

Недостатки такого форматирования (к сожалению, традиционного для Лиспа) мы уже отмечали выше. Чтобы в такой ситуации соблюсти стиль с фиксированным размером отступа, нужно действовать совершенно иначе:

```
(let
   (
      (seq (build-sequencer 0))
      (colset '(red orange yellow green blue violet))
      (ls nil)
   ) ;;
   (setq ls (mapcar #'list seq colset the-list))
   (store ls)
   (reverse ls)
)
```

Обратите внимание на символ комментария в конце списка связываний; применять его, конечно, не обязательно, но он позволяет яснее отделить заголовок формы `let` от её тела. Можно найти и другой допустимый стиль, например:

```
(let (
   (seq (build-sequencer 0))
   (colset '(red orange yellow green blue violet))
   (ls nil)
) ;;
   (setq ls (mapcar #'list seq colset the-list))
   (store ls)
   (reverse ls)
)
```

К такому стилю придётся некоторое время привыкать, очень уж непривычно выглядят две закрывающие скобки на одном уровне в условиях отсутствия открывающей скобки; но, как ни странно, это всё же лучше, чем показанный выше «традиционный» вариант, в котором не соблюдается постоянство размера отступа и закрывающие скобки скапливаются в конце конструкции, образуя нечто неудобоваримое.

3.3. Пролог, Эрланг и другие

Программа на Прологе состоит из *предложений*, каждое из которых имеет заголовок[2] и, возможно, тело, которое в большинстве случаев представляет собой список термов, но может иметь и более сложную структуру благодаря наличию связки «или», обозначаемой точкой с запятой (обычная связка «и» обозначается простой запятой). Предложения с заголовками, имеющими одинаковое имя и «арность», составляют *процедуры*. Аналогичный синтаксис используется в Эрланге и некоторых других языках.

[2] В английском оригинале — просто *head*, то есть «голова».

Выражения Пролога сильно напоминают уже знакомые нам S-выражения Лиспа — собственно говоря, отличие только в появлении функторов и функциональных термов, и плюс к тому некоторые унарные и бинарные термы можно записывать в инфиксной нотации. На первый взгляд может показаться, что синтаксис при этом становится сложнее; однако в контексте рассуждений об оформлении кода ситуация оказывается прямо противоположной: в большинстве случаев оформлять код на Прологе гораздо проще. Дело в том, что программы на Прологе намного реже требуют многоуровневой вложенности конструкций.

Пока предложение умещается в одной строке, его часто так в одну строку и записывают:

```
no_attack(_, []).
no_attack(Q, [X|L]) :- not(attacks(Q, X)), no_attack(Q, L).
```

хотя в некоторых руководствах и примерах применяется другой стиль, предполагающий, что на одной строке с заголовком предложения не пишется больше ничего, даже если правая часть предложения состоит всего из одной «цели»; правая часть всегда записывается, начиная со следующей строки, со сдвигом вправо, причём обычно такой сдвиг выбирают равным табуляции:

```
queens(N, Result) :-
        do_queens(N, 0, [], Result).
```

В этом случае остаётся вопрос, как расположить цели правой части, если их больше одной, но они, в принципе, умещаются в одну строку. Допустима как запись «в столбик»:

```
no_attack(Q, [X|L]) :-
        not(attacks(Q, X)),
        no_attack(Q, L).
```

так и запись «в линейку»:

```
no_attack(Q, [X|L]) :-
        not(attacks(Q, X)), no_attack(Q, L).
```

Какую из них предпочесть, решайте сами; как обычно, единственной рекомендацией здесь будет придерживаться какого-то одного стиля во всей программе.

Если же предложение в одну строку не умещается — то остаётся фактически только один вариант, а именно — записать его «столбиком»:

```
sameset([], []).
sameset([X|L], M) :-
```

```
        member(X, M),
        delete_all(X, M, M1),
        delete_all(X, L, L1),
        sameset(L1, M1).
```

Многие авторы программ на Прологе делают одну и ту же характерную ошибку при оформлении таких многострочных предложений, а именно — записывают первую цель правой части в одной строке с заголовком, а последующие цели размещают точно под первой. Получается примерно так:

```
sameset([], []).
sameset([X|L], M) :- member(X, M),
                     delete_all(X, M, M1),
                     delete_all(X, L, L1),
                     sameset(L1, M1).
```

Недостаток такого решения мы уже неоднократно обсуждали: размер отступа оказывается зависящим от неких обстоятельств, в данном случае — от длины заголовка предложения.

Ситуацию может сильно осложнить появление в предложениях точки с запятой (напомним, она изображает в Прологе дизъюнкцию). В ситуации, когда предложение состоит из нескольких альтернатив, разделённых точкой с запятой, и каждая альтернатива умещается в строку, проще всего так и писать по одной альтернативе в строке:

```
lexic_process_char(Ch, Line, Newstate, Sofar, R) :-
        isspace(Ch), !, R = Sofar, Newstate = home;
        [Ch] = ",",  !, R = [comma|Sofar], Newstate = home;
        [Ch] = ":",  !, R = [colon|Sofar], Newstate = home;
        [Ch] = ";",  !, R = [semicolon|Sofar], Newstate = home;
        [Ch] = ".",  !, R = Sofar, Newstate = stop;
        isalnum(Ch), !, R = [[Ch] | Sofar], Newstate = id;
        !, write('lexical error at line '), write(Line), nl, fail.
```

Если в аналогичной ситуации некоторые альтернативы в строку не помещаются, можно посоветовать что-то вроде следующего:

```
lexic_process_char(Ch, Line, Newstate, Sofar, R) :-
        isspace(Ch), !,
                R = Sofar,
                Newstate = home
        ;
        [Ch] = ",", !,
                R = [comma|Sofar],
                Newstate = home
```

```
;
[Ch] = ":", !,
        R = [colon|Sofar],
        Newstate = home
;
% . . .
```

Здесь мы воспользовались тем, что в примере сопоставление вместе с отсечением представляет собой что-то вроде «заголовка» для альтернативы, поскольку задаёт условие, при котором данная альтернатива будет исполнена, а остальные исполнены не будут.

Ну а для более сложных случаев, в особенности при использовании скобок для группировки целей, можно предложить следовать общим принципам оформления вложенных конструкций: начало и конец конструкции размещать строго друг над другом, а всё вложенное — сдвигать вправо на постоянный размер сдвига.

Вместо заключения

Конечно, в этой сравнительно небольшой книжке невозможно было учесть всего многообразия особенностей различных языков программирования, охватить все ситуации, которые могут возникнуть при написании программ, или перечислить все существующие (и допустимые) подходы к решению проблем, связанных с грамотным оформлением кода.

Как мы видели, во многих (если не во всех) случаях возможны и допустимы различные решения одной и той же проблемы, причём указать среди них один «однозначно лучший» не представляется возможным. В таких условиях процесс оформления программного кода становится делом вполне творческим; впрочем, программирование всегда было скорее искусством, нежели чем-то ещё. Если говорить именно об оформлении, то в особенности это касается декомпозиции программы на подсистемы, модули и отдельные подпрограммы, а в случае ООП — на классы; впрочем, в применении к большим программам речь здесь пойдёт уже не об оформлении, а о *проектировании архитектуры*. Грамотное оформление кода здесь — лишь первый маленький шаг большого пути, но этот шаг оказывается на удивление важен.

Надеюсь, мне удалось показать, что́ представляет собой оформление программного кода как предметная область, и продемонстрировать возможные ответы на некоторые (но не все!) возникающие в этой области вопросы. В любом случае, уважаемые читатели, дальнейшее зависит только от вас. В процессе практического программирования вам могут встретиться ситуации, не охваченные настоящим пособием, вы можете столкнуться с языками программирования, традиции или синтаксические особенности которых вынуждают изобретать что-то новое. Используйте свой опыт, используйте здравый смысл, используйте, наконец, собственные ощущения из области эстетики. Пособие, которое вы прочитали, способно лишь указать более-менее правильное направление, но последовать по этому пути вам придётся самостоятельно. Прощаясь, я хотел бы пожелать вам успехов в том замечательном виде человеческой деятельности, каким, несомненно, является программирование.

Учебное издание

СТОЛЯРОВ Андрей Викторович

ОФОРМЛЕНИЕ ПРОГРАММНОГО КОДА

издание второе, исправленное и дополненное

Рисунок и дизайн обложки Елены Доменновой

Напечатано с готового оригинал-макета

Подписано в печать 30.10.2019 г.
Формат 60x90 1/16. Усл.печ.л. 7,25. Тираж 150 экз. Заказ 253.

Издательство ООО "МАКС Пресс"
Лицензия ИД № 00510 от 01.12.99 г.

11992 ГСП-2, Москва, Ленинские горы,
МГУ им. М.В.Ломоносова, 2-й учебный корпус, 527 к.
Тел. 939-3890, 939-3891. Тел./Факс 939-3891