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.

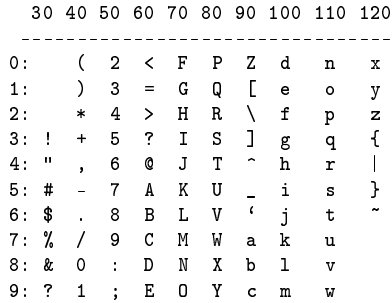

Рис. 1.1: Отображаемые символы ASCII

- 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.**