

Библиотека
Профессионала

Комлев Н. Ю.

Полезное программирование

*Уникальное руководство
к действию*


«СОЛОН»

Комлев Н. Ю.

Полезное Программирование

**Москва
СОЛОН-Пресс
2016**

УДК 681.3
ББК 32.973-18
К 63

Комлев Н. Ю.

Полезное программирование. — М.: СОЛОН-Пресс, 2016. — 256 с.: ил.

ISBN 978-5-91359-171-5

Уникальное руководство к действию, опыт и практика, сконцентрированные на двух сотнях страниц.

Книга для тех, кто умеет программировать и уже готов этим зарабатывать.

Курс молодого, но профессионального программиста. Практические советы.

Что пригодится вам из пройденного и сданного. Что вы будете программировать сначала, а что позже, когда наберётесь опыта. К чему ещё можно применить ваш программистский талант, чтобы руководство одобрило и поощрило. Кратко, доступно, убедительно.

КНИГА - ПОЧТОЙ

Книги издательства «СОЛОН-Пресс» можно заказать наложенным платежом (оплата при получении) по фиксированной цене. Заказ можно оформить одним из трех способов:

1. Послать открытку или письмо по адресу: 123001, Москва, а/я 82.
2. Оформить заказ на сайте www.solon-press.ru в разделе «Книга — почтой».
3. Заказать книгу по тел. (495) 617-39-64, (495) 617-39-65.

Каталог издательства высылается по почте бесплатно.

При оформлении заказа следует правильно и полностью указать адрес, по которому должны быть высланы книги, а также фамилию, имя и отчество получателя. Желательно указать дополнительно свой телефон и адрес электронной почты.

Через Интернет вы можете в любое время получить свежий каталог издательства «СОЛОН-Пресс», считав его с адреса

http://www.solon-press.ru/docs/Katalog_SoIon_Press.xls.

Интернет-магазин размещен на сайте www.solon-press.ru.

По вопросам приобретения обращаться:

ООО «АЛЬЯНС-БУКС»

Тел: (499) 725-54-09, 725-50-27,

www.aliants-kniga.ru

ISBN 978-5-91359-171-5

© Комлев Н. Ю., 2016

© «СОЛОН-Пресс», 2016

*Посвящается
всем людям,
которые хотят получить пользу
от программирования*

СОДЕРЖАНИЕ

Вступление	7
Почему Полезное Программирование? И кому полезное?	7
А на каком языке писать? А какая разница?	9
Что должен знать читатель	9
Уточнения и ограничения	11
Предупреждения	11
Что ещё можно почитать, кроме меня?	12
Самореклама	15
Это сладкое слово — халява	15
Глава 1. Всё-таки учебная. Но в хорошем смысле	17
Вступление в предисловие	17
С высоты птичьего полёта	20
Теперь о главном	24
Продолжение вступления в предисловие	26
Практическая часть. Просто сортировка	27
Зачем нам сортировка Шелла?	30
И факториал!	31
Глава 2. Не совсем учебная. Графики	34
Немного о графиках	34
Немного о графиках. Продолжение	38
Немного о графиках. Финиш	41
Совсем всё. Финиш	48
Есть ли жизнь после финиша?	54
Глава 3. С чего начинается работа	61
О чём речь?	61
Что-то длинное	61
Очень просто — массив. Свёртка	62
Массив. Меняются значения. И меняется длина	63
Похожие задачи, но для файлов	64
Похожие задачи для списков	66
Объекты, вид сбоку	69
Задача напоследок	71
Глава 4. Как продолжается работа. Что-то ветвистое	80
Прембула	80

Что-то посложнее. Короткий раздел	82
А теперь о главном. Важное	83
Что делать?	87
Глава 5. Системы контроля версий	94
О чём эта глава	94
Скучное. Предмет и терминология	95
Лирическое отступление, или а зачем это вообще надо?	97
Какие они вообще бывают?	98
А теперь о главном	99
А теперь всё вместе и разом!	100
Вася один на ландшафте	101
Вспомогательное счастье	103
Подтянулась и компания	108
Как всё сделать хорошо	111
Про Одну Кнопку	115
То, что называется «Бинарные файлы»	124
Как НЕ НАДО работать с VSS	126
Глава 6. О скучном — динамические библиотеки	131
Что это такое, а также историческая справка, которую можно не читать	131
А как с этим теперь?	133
Сначала мы её создадим!	134
Статическая загрузка и как это может пригодиться	137
Динамическая загрузка и расширение возможностей	139
Польза от динамического вызова	141
Библиотеки — технические подробности	146
Реализация проекта	147
Глава 7. Мелкие технические подробности	151
Легко ли быть жадным?	151
А что делать?	152
Теперь конкретно об эффективности	154
Оптимизируем на самом нижнем уровне	156
Улучшаем алгоритм	160
Меняем алгоритм	162
Глава 8. Техника программирования, или Это вы обязательно должны знать.	164
Абстрактные рассуждения или а оно нам надо?	164
Кратко о предмете обсуждения	166
Рядом лежащие вопросы	167
Подумаем конкретно, на примерах. Первый пример	168
Пятый способ, бессмысленный и бесполезный	173

То же самое, вид сбоку. Второй пример	175
Всё не так и всё разное. Третий пример	176
Глава 9. Что должен знать программист? Как он должен себя вести?	178
Нет, не всё	178
О чём я вообще?	179
Математика	180
Информатика (Computer Science)	183
Глава 10. Для дома, для семьи	185
Зачем это надо?	185
Несложный бредогенератор. Постановка задачи	186
Бредогенератор. Сначала немного о ЛИСПе	191
Бредогенератор. Реализация	192
Бредогенератор. Доработка напильником и планы на будущее	199
Искатель тайного смысла. Обсуждение	201
Искатель тайного смысла. Постановка задачи	203
Искатель тайного смысла. Подготовка	203
Искатель тайного смысла. Реализация в первом приближении	210
Искатель тайного смысла. Развитие темы	214
Расстояние между словами. Введение в тему	219
Расстояние между словами. Реализация	221
Приложения. Справочник обо всём	224
Приложение А. Всё о данных	224
Приложение В. Всё об организации программы на нескольких страницах	228
Приложение С. Всё о процедурах и функциях	231
Приложение Д. Всё продвинутое программирование на нескольких страницах	235
Приложение Е. Всё о списках. Или почти всё	240
Приложение Ф. Всё Объектно ориентированное программирование	242
Приложение Г. Простенький модуль для работы со строками. Простой, но очень полезный	247
Задания для самостоятельного развития	253

ВСТУПЛЕНИЕ

Почему Полезное Программирование? И кому полезное?

Чем эта книга отличается от других? Когда я говорю «от других», я, само собой, имею в виду — чем эта книга отличается от других *моих* книг. Отличается она именно тем, что книга эта посвящена именно *полезному* программированию.

Есть книги для начинающих, для тех, кто ничего, или почти ничего, не умеет (читайте мой нетленный «Самоучитель игры на Паскале»), Есть книги для профессионалов (читайте моё искрящееся неподражаемым юмором «Объектно Ориентированное Программирование»), По логике, где-то между ними должна находиться книга для тех, кто что-то кое-где уже порой знает, но ещё далеко не профессионал. И я тоже так сначала думал. И даже почти написал две книги — «Школа игры на (Турбо) Паскале» и «Очень маленькое Delphi». Готовы они были на восемьдесят процентов, и это были неплохие книги, поверьте мне. Потом я решил, что мыслю одномерно и плоско, и пришёл к выводу, что книги эти не нужны. Почему?

С книгой по Турбо Паскалю объяснение простое. Я нисколько не сомневаюсь, что это была бы лучшая книга по программированию на Паскале, не только моя, но и вообще в мире. Это не потому, что я такой скромный и умный, а потому, что это была бы последняя книга по программированию на Турбо Паскале в мире. А тот кто пережил всех конкурентов, дотянул до финиша, и выключает за остальными свет — лучший по определению.

Конечно, Паскаль никуда не делся. Он превратился в язык Object Pascal, который потом незатейливо переименовали в Delphi. Вокруг языка Delphi выросла среда программирования Delphi. Об этом я мог бы написать много полезного, проблема в том, что до меня уже написано много полезного. Число книг по этой теме на русском языке исчисляется десятками. Ну и зачем мне писать сто первую книгу? Да, она будет лучше предыдущих ста, но кто это заметит и оценит?

Поскольку книги я пишу не ради денег — таки разве это деньги? — а из чистого незамутнённого удовольствия, то я выкинул всё мною почти написанное и начал с чистого листа.

- *Способны ль вы кого-нибудь убить?*
- *Вы шутите, такой как я готов убить любого*

- *Вы знаете графиню де Бель Флёр?*
 - *Да, в этом доме кое с кем я дружен.*
 - *Вы можете убить её слугу?*
 - *Готов убить всех слуг, и всех служанок, и даже лошадей её кареты.*
 - *Ну, если так, убейте Теодоро.*
 - *Тут надо взяться чуточку иначе.*
- © Лопе де Вега, Собака на сене

Это всё к тому, что я такой же добрый и трудолюбивый, как и персонаж пьесы. Если что, могу всех, включая лошадей её кареты. Но, в нашем случае, для начала, действительно надо взяться чуточку иначе.

Сейчас модно на обложке программистских книг рисовать горизонтальную диаграмму. Слева абсолютный ноль в программировании, справа барельеф «Билл Гейтс убивает Стива Джобса, не сумевшего запрограммировать Тетрис». На этом пустом горизонтальном прямоугольнике отображается, для кого предназначена книга. Если прямоугольник закрашен слева — для самых начинающих. Если справа — для профессионалов. Посередине — понятно.

А теперь воспарим над плоскостью вверх и возьмёмся чуточку иначе. На левом конце интервала у нас те, кто ничего не знают, на правом конце — те, кто знают почти всё. А теперь поглядим по другому, не с точки зрения объёма знаний, а с точки зрения цели. Слева у нас те, кто хочет научиться программировать, чтобы после этого уметь программировать. Справа те, кто программировать давно умеют и зарабатывают этим на жизнь, и хотят зарабатывать ещё больше. А между ними Сумрак. Серая зона. Нейтральная полоса. И мотивация для программирования и чтения программистских книжек у всех этих категорий совсем разная.

Вот эту Серую Зону я желаю окутить. Объясняю. Между теми, кто только что научился программировать, и между теми, кто программирует за деньги, есть некоторый неохваченный промежуток, и там живые люди. И вот им я хочу помочь, и могу им помочь, потому что я все эти стадии давно прошёл.

Эта книга не учит программированию, я предполагаю, что читателя этому научили. К сожалению, школьникам она не подойдёт. Студентам — да, может быть даже тем, что пишут курсовые и не знают — как? Но и это вряд ли. Студент старших курсов, который уже полдня где-то работает, причём даже за деньги — это да.

Но основной читатель этой книги, которому от неё будет наибольшая польза, всё-таки не студент в процессе обучения, даже при этом работающий, а программист, только что начавший исполнять нелёгкие обязанности программиста. Да, конечно, он в себе абсолютно уверен, он всё знает, но по приходу на **работу** его почему-то приходится переучивать.

Я хочу объяснить, почему одни студенты при приёме на работу **оказываются** лучше других. Чего от студента, извините, уже программиста, на работе **хотят**. Какую работу ему, скорее всего, поручат. В конце концов, я хочу показать **(на**

примерах), как пишется программа. Причём программа не по заданию из учебника — хотя мы с этого непременно начнём. И не программа, которую мы пишем неведомо для кого за бешеные деньги по странному ТЗ (Техническому заданию). Заодно вы узнаете, что такое Техническое Задание, и почему его никогда не бывает.

Ещё раз повторю — я знаю, как надо. Сначала меня учили. Потом я был маленьким программистом под руководством маленьких начальников. Потом я сам учил других и стал маленьким начальником. Поэтому я знаю. Не всё, но почти всё.

А на каком языке писать? А какая разница?

На каком языке программирования буду я объяснять и приводить примеры — в целом это важно, но не очень важно, потому что книга не об этом. Но, поскольку из всех языков я лучше всего знаю Паскаль, переходящий в Delphi, то почти все примеры на нём и будут. Можно рассматривать Паскаль в основном тексте книги в качестве псевдокода, в качестве чего он, Паскаль, массово использовался раньше и часто используется сейчас. Для тех, кому интересно, или полезно, весь Паскаль, ныне Delphi, будет на немногих страницах изложен в приложении, то, что я нём считаю важным, разумеется.

Ещё надо обдумать, не следует ли написать главу о C#, или сразу книгу. Разумеется, все знают, что C# изобрела та же команда, что раньше изобретала Delphi, поэтому сейчас наблюдается массовая миграция программистов с Delphi на C#, точно так же как и миграция с C++ на C#.

Что должен знать читатель

Ещё раз вернёмся к вопросу, кого я предполагаю читателем этой книги. Перед моим внутренним взором встаёт только что закончивший Университет специалист, или магистр, приступивший к своей первой работе. В реальности это скорее будет специалист или магистр, приступивший к своей второй работе, потому что на первой работе он трудился ещё в Университете. Но та работа не вполне считается, она была не вполне настоящая. Не потому что он, магистр, был плохим программистом, а потому, что студентам как-то не до конца доверяют, и причины этому есть.

Помню, однажды пришёл к нам на практику юноша, серьезно пришёл, даже принёс кружку, чай пить. На следующий же день с утра позвонил, и сказал что задержится, потому что ему срочно надо зайти в военкомат... Больше мы его не

видели и не слышали. Искренне верю, что его немедленно призвали и послали. Врать не буду, кружка до сих пор с нами.

Кроме того, студенты всё время бегают, туда-сюда, туда-сюда, работа, университет, военкомат, обследование... Никакому, даже самому маленькому и терпеливому начальнику это не понравится. И ещё студенты всё время хотят поступить в аспирантуру, это тоже не прибавляет к ним симпатий, даже если аспирантура заочная. Так что пусть лучше Нашим Читателем будет человек, разорвавший все связи со своим Высшим Учебным Заведением и пришедший на работу просто работать.

//Уточнение

Всё вышесказанное о недостатках студентов касается в первую очередь их ненадёжности как работников, а совсем не их ума. Бывают и невообразимо тупые, но чаще всё-таки наоборот. Один из бывших у нас студентов позже состоял в команде, выигравшей то ли чемпионат, то ли олимпиаду России по программированию. Занудства ради, на тот момент он был ещё студентом колледжа, бывшего индустриального техникума. Наберите в Google 'колледж имени Коняева хакер'. Нет, это не он.

//конец Уточнения

Читатель должен уметь программировать. От слова вообще. Я не буду объяснять, что такое цикл, массив, процедура, функция. Я даже не буду объяснять, что такое класс и что такое — экземпляр класса. Но, поскольку я очень добрый, я это объясню. Но, поскольку я не только очень добрый, но ещё и очень честный, то, чтобы не увеличивать объём книги, я объясню все понятия программирования в Приложении, очень коротко.

Единственное, что я буду объяснять подробно в основном тексте, это использование указателей, В отличие от переменных, условных операторов, циклов и массивов, указатели никак нельзя назвать интуитивно понятными. Сегодня указатели не все могут понимать. Вернее, понимать могут не только лишь все, мало кто может это делать.

Кстати, раньше многие учёные (Дейкстра) и типа учёные работали в области формального доказательства правильности программ, но указатели при этом стыдливо игнорировались. Ещё, как выяснилось, не все могут сегодня понимать и файлы.

Хотя все примеры в книге на Delphi (в девичестве Pascal), программистам на C++ беспокоится нечего. Как всем известно, Дельфи гораздо проще и примитивнее чем С, так что они (сишники) поймут всё просто с полпинка.

Уточнения и ограничения

Что тут будет и чего не будет. Я — программист. Нет, вы не поняли — я настоящий программист. Нет, вы опять не поняли. Здесь слово «настоящий» не означает «суперпрограммист». Я — просто настоящий программист. От слова «просто». Что это значит? Я не специалист по базам данных — хотя ими занимался. Я не специалист по сетям, хотя ими занимался тоже. Ещё я не Уэб-Дизайнер, потому что этим я не занимался вообще. Я Просто Программист. Чем и горжусь. Последние десять лет я даже не программировал, что называется, «по железу». И ещё я не знаю эту, как её, бухгалтерия какая-то...

К чему это всё? А к тому, что я знаю, как написать правильную, хорошую, годную программу. И чтобы она потом ещё и работала. При условии, что эта программа не относится к базам данных, сетям, Интернету и прочей дребедени. И к бухгалтерии.

А вот цитата, что называется, в мать:

Вы можете получить автомобиль любого цвета, при условии, что этот цвет — чёрный

© Генри Форд.

Применительно к программированию, смысл этой фразы такой. Если вы научитесь программировать вообще, то потом вы легко освоите сети, базы, облачные вычисления и прочую ерунду.

Предупреждения

Сейчас это называется Disclaimer, типа, я вас предупредил, фирма не отвечает, если ваши данные *внезапно* куда-то исчезнут или где-то появятся.

Delphi — это не среда разработки, а уже давно язык программирования. Так решила фирма Borland. Был раньше язык программирования Turbo Pascal, потом появился Object Pascal, а вот теперь оно называется Delphi.

В этой книге будет меньше программных текстов, чем в моих предыдущих творениях и больше просто текста. Это естественно. Чем выше уровень подготовки читателя и сложнее возложенные на него обязанности, тем меньше в книге кода. Почитайте, наконец, я много раз рекомендовал, Брукса младшего, там кода нет вообще, это конечно предельный случай, я до этого ещё не дорос.

Тексты программ здесь не доведены до совершенства и предельного блеска. Не потому, что я не вижу недостатков — я их вижу и, как правило, их даже указываю. Не потому, что я не могу их исправить — честное слово, могу. Не потому, что мне лень — хотя, конечно, лень. А потому, что это книга для хотя и начина-

ющих, но уже профессиональных программистов. Идеального кода в реальной, неидеальной жизни не бывает. Код пишется исходя из того отрезка пути, который программист видит перед собой. А дорога резко поворачивает — и мы видим картину Репина «Приплыли». А по бокам радостно щёлкают клювами Птица Обломинго и литовский политический деятель Обломайтис. И гляжу я на свои программы, написанные совсем недавно и плачу горькими слезами. И, что обидно, с годами ничего не меняется, если опыт в эффективном написании кода накапливается, так же быстро совершенствуется и способность принимать неверные решения.

Это я всё к тому, что если читателю мои программы местами не нравятся, то обычно я это заранее угадываю и заранее рекомендую читателю переписать мутные места прозрачно и ярко. А вообще для программиста, и не только программиста, очень важно, приняв решение, выполнять его. Даже если возникают сомнения.

— Это дорого? — Да!

— Она ведёт к цели? — Да!

— Бежим!

© Фридман, это хороший Фридман, который физик.

И, просто на всякий случай, я вовсе не считаю всех пишущих на C++ идиотами, даже если складывается такое впечатление. Это вполне приличные, милые, забавные люди. Бывают даже и девушки. Просто я глубоко убеждён, что C — язык для очень умных людей, а все остальные должны писать на Delphi.

И ещё, я везде обращаюсь к читателю на «вы» с маленькой буквы, но это не от недостатка уважения. Просто я где-то прочитал, что можно и так.

Что ещё можно почитать, кроме меня?

Разумеется, в полной силе остаётся список литературы из предыдущих книг, но здесь я приведу только те из них, которые касаются только программистов. Те книги, которые относятся к деятельности маленьких и больших начальников, проектировщиков и аналитиков, я опущу. За прошедшее время литература эта ничуть не стала хуже, так что напоминаю, без комментариев.

Из старого, даже очень старого, но того, что полезно будет прочитать:

Б. Керниган, Ф. Пладжер «Элементы стиля программирования», «Радио и связь», 1984 *“The Elements of Programming Style”*

Лу Гринзоу «Философия программирования Windows 95/NT», Санкт-Петербург, «Символ», 1997 *Lou Grinzo Zen of Windows 95 Programming*.

Роджерс «Алгоритмические основы машинной графики» Мир, 1989
 “Procedural Elements for Computer Graphics”

Дж.Фолл, А. Вэн Дэм «Основы интерактивной машинной графики» Мир,
 1985 “Fundamentals of Interactive Computer Graphics” в двух томах

Б.Мейер, К.Бодуэн «Методы программирования» Мир, 1982, для разнообра-
 зия, перевод с французского

Из не очень старого:

Уэллингс «Как не надо программировать на C++» Питер “How Not to
 Program in C++”

Стивенс «Delphi. Готовые алгоритмы» ДМК Пресс “Ready-to-run Delphi
 algorithms”

Керниган, Брайан «Практика программирования» Невский Диалект “The
 Practice of Programming”

Э.ант, Д.Томас «Программист-прагматик» Питер “The
 Pragmatic Programmer”

С.Макконелл «Совершенный код» Питер, 2005 “Code Complete”

Это очень толстая книга, поэтому я прокомментирую. Первые две четверти
 книги унылы, третья четверть веселее и полезнее, дальше ещё лучше.

А ещё, я прочитал книгу, к сожалению, покойного автора. К сожалению для
 меня, не для него — потому что при его жизни я о нём и не слышал, а ему теперь
 вообще всё равно. Это академик Арнольд. Здесь я бы хотел порекомендовать его
 замечательную книгу

В.И. Арнольд «Что такое математика», М.: МЦНМО, 2002

Что такое МЦНМО, я, врать не буду, не знаю, но наверняка что-то страшное
 и ужасное.

Я всегда думал, что я тролль семьдесят девятого уровня. Я ошибался. Я пре-
 бываю в грехе Гордыни, по Библии это Смертный Грех Номер Один, согласно
 номенклатуре смертных грехов. Ещё раз, я ошибался. Я всего-навсего тролль
 семьдесят восьмого уровня. Семьдесят девятого — это Арнольд. А тролль вось-
 мидесятого уровня у нас в стране только один.

Предложу мимоходом две задачки от Арнольда, не из этой книги, просто из
 сборника задач от его имени.

Первая:

*16. Какую долю от объема пятимерного куба составляет объём вписанного
 в него шара? А от десятимерного?*

Арнольд В.И Математический тривиум

Шестнадцать — это номер задачи в списке. Почему я пропустил предыдущие пятнадцать и пропустил все последующие? Потому что предыдущие пятнадцать это задачи математические, а эта чисто программистская задача. Для её решения не надо ничего знать о пятимерных и десятимерных пространствах. Помните верблюда у Пратчетта который жевал жвачку и медитировал о восемнадцатимерных пространствах? Он — верблюд! Но мы же с вами не верблюды! Повторяю — это задача для программистов. Правильный программист обязан решить её, обладая знаниями геометрии в объёме шестого класса (советской) школы.

И, чисто из садизма, ещё одна задачка от Арнольда:

У Маши не хватало для покупки букваря семи копеек, а у Миши одной копейки. Они сложились, чтобы купить один букварь на двоих, но денег все равно не хватило. Сколько стоил букварь?

Из двадцати человек, которым я её предлагал, решил её только один. Не считая меня, само собой (сам не похвалишь — никто не похвалит). Но мы решили в уме. Кроме нас, решила её гениальная девушка с красным дипломом и другими большими достоинствами, она составила систему уравнений и таки решила тоже. И вы попробуйте.

И ещё несколько книг, которых не было в предыдущих списках:

Грэхем Р., Кнут Д., Поташник О. «Конкретная математика. Основание информатики», М. Мир, 1998 “Concrete Mathematics. A Foundation for Computer Science”

Не так, чтобы очень увлекательно, но прочитать стоит.

Поля «Математика и правдоподобные рассуждения»

Да, я знаю, что этой книге скоро будет почти сто лет, но я прочитал её только сейчас. Как математик математику он мне ничего нового не сказал, но меня удивило другое. Автор ставит перед собой задачу, математическую, и решает её, подробно объясняя процесс поиска решения. Меня удивило, насколько близок путь решения математической проблемы к проблеме решения задачи написания программы. Программу писать, само собой, гораздо сложнее.

А ещё несколько щшг я ненавязчиво порекомендую в соответствующих разделах. Например, обязательно надо прочитать хоть что-то по математике. Я взял на себя труд пролистать огромное (ну, очень большое) количество книг математической тематики и выбрал из них несколько, которые *надо* прочитать программисту и несколько книг, которые *можно* прочитать программисту.

Самореклама

Повторюсь — сам не похвалишь, никто не похвалит.

Сначала цитата, из кинофильма:

Coming soon:

History of the World, Part II

Viking's Funeral

Hitler on Ice

Jews in Space

© Мэл Брукс, спасибо дедушке.

Так вот, уже в разработке:

«Абсолютное руководство по написанию игр» в трёх томах.

Тема проектирования, разработки и программирования компьютерных игр будет, наконец, закрыта навсегда.

«Книга для больших программистов и маленьких начальников».

Как всё и сказано в названии — как руководить маленьким коллективом и заставить его написать большую программу, без нанесения серьёзных телесных повреждений.

«Программы-мутанты, или почему они придут за вами!».

Это книга о самомутирующих программах. Врать не буду — на данный момент программы в основном уже есть, а книги — почти ещё нет. Но когда-нибудь они окончательно мутируют и придут за вами. Кстати о мутациях — Word предложил заменить слово «мутируют» или на «матируют» или на «гутируют». Задумался, не включить ли в будущую книгу главу о мутирующих (матирующих) текстах.

Это сладкое слово — халява

«Безвозмездно, то есть даром»

© Сова

В качестве бесплатного приложения на диске (виртуальном) присутствует половина книги «Школа игры на Паскале». Чуть не написал — «на шестиструнном Паскале».

Почему она там лежит? Потому что я пришёл к выводу, что чистый Паскаль, в любом виде, больше не актуален. Не то, чтобы Паскаль как-то сильно изменился за прошедшие годы и даже десятилетия. Он вообще не изменился. Что-то добавилось, и довольно много. И что-то расширилось, — была переменная два байта, стала четыре или даже восемь. Разве это изменения?

Паскаль нисколько не устарел. Delphi — это тот же Паскаль. Ненужной эту книгу я считаю в первую очередь для себя. Мне больше неинтересно учить языку программирования Паскаль. И не потому, что это именно Паскаль. К следующей своей книге в качестве бесплатного приложения я прицеплю половину книги «Очень маленькое Delphi», потому что мне неинтересно учить языку программирования Delphi.

Я не хочу про языки программирования. Я хочу вообще.

А почему я выложу только по половине книги? Не потому, что я жадный, хотя я само собой жадный. А потому, что кроме этих двух половин, я больше ничего и не написал. Но это хорошие половины, самые лучшие из двух.

ГЛАВА 1

Всё-таки учебная. Но в хорошем смысле

Вступление в предисловие

Назначение этой книги в том, чтобы научить писать не высосанные из пальца учебные задания, а программы, хоть сколько-то полезные. Но здесь мы невольно впадаем в бесконечный цикл рассуждений, или, если угодно, в рекурсию. Если мы не пишем учебные программы, а только полезные, то зададим себе вопрос — может ли учебная программа быть полезной?

Может, если эту учебную программу обязательно надо написать, а если не напишешь, то накажут, а может быть, даже и вломят. Поэтому, в этой главе мы рассмотрим несколько типовых учебных программ, относясь к ним как к полезным (я не слишком запутанно выражаюсь?).

Самая первая программа будет и самой полезной. Дело в том, что по своим служебным обязанностям, я, время от времени, провожу собеседования с претендующими стать программистами. К желающим у меня только два незначительных требования. Первое — написать хоть сколько-то адекватное резюме. С этим справляются почти все, хотя один кандидат ухитрился засыпаться и на этом этапе. Это сложно, но он смог.

Я открою страшную тайну. У нас не Америка. Ваши достижения, знания и навыки тщательно в резюме описанные, никого, кроме вас, не волнуют и не интересуют. То есть, само собой, всё это представляет определённый интерес, и, безусловно, будет прочитано. Главное назначение резюме, однако, совсем не в этом. Главное его предназначение — проверка вашей адекватности и вменяемости. Объясняя.

Первый рубеж — если претендент спрашивает «а зачем эта фигня, которая резюме, вообще нужна?» — он подлежит немедленному отстрелу, или, как сейчас модно говорить, демократической люстрации. Претендент должен делать то, что ему скажут. Нам не нужны те, которые умничают. Умничать можно потом, после приёма на работу, да и то очень не сразу. А при поступлении надо вести себя скромно. Только не путайте понятия «быть умным» и «умничать». Это очень разные вещи. «Не умничать» на собеседовании — значит, делать, что говорят и отвечать на заданные вопросы. А если не спрашивают, то не отвечать, и держать своё мнение при себе. Да, мир жесток.

Второй рубеж для каждого свой. И какой именно для кого, зависит не от претендента, а от изучающего резюме. Каждого раздражает своё, но есть и общие принципы. Возможно, я ошибаюсь, но у меня сложилось впечатление, что в Той Стране, откуда к нам и пришло понятие «резюме», чрезмерное внимание уделяют грамотности, то есть орфографии и синтаксису. Видимо, их цивилизация ориентирована на слова, а наша на цифру. У нас всё не так. На ашипки и запятые я давно уже не обращаю внимание. Важно другое.

Если клиент не в состоянии внятно изложить места своей работы и учебы, в хронологическом порядке, неважно как: по-русски — от прошлого к будущему, по-буржуйски — от будущего к прошлому, значит, имеем налицо дезориентацию в пространстве-времени и невозможность отслеживать причинно-следственные связи. Такой клиент нам не нужен. Аналогично, большинству потенциальных руководителей не нравятся претенденты, которые просто не помнят, где и когда они работали. Это настораживает.

Некоторые места учёбы и работы не вполне приветствуются. К примеру, не очень хорошо звучит, в качестве места учёбы, Мухосранский Гуманитарный Университет. И не потому, что Мухосранский. А потому, что Гуманитарный. Да к тому же ещё и Университет.

Не вполне хорошим местом работы следует признать и конторы с названиями типа «ГлавАнализ».

// Воспоминание

Во-первых, для соотечественников это название напоминает пресловутую «Глав-Рыбу», это из книги/кинофильма «Собачье сердце». Для иностранцев — всё гораздо хуже. Однажды, когда я был молод и добр, я переводил для одной милой, однако наивной девушки анкету на иностранный язык. Наивная нечукотская девушка хотела уехать в ЮАР (по буквам — в Южно Африканскую Республику). Наивной полчукотской девушке рассказали, что в их ЮАР, если белый человек идёт по улице с чемоданом, местный чёрный человек должен подбежать, схватить чемодан и нести его вместо белого человека. Чукотская девушка была не в курсе, что в ЮАР уже случилась негритянская революция, к власти пришёл Нельсон Мандела и жена его Винни, и там начали радостно резать белых.

Так вот, вся трудовая биография девушки состояла из названий типа «ГлавТорф», «ТехСнаб» и «СпецПрицеп». Естественно, это вызывало определённые трудности перевода.

Мораль сей басни такова, что девушка осталась здесь, и, если она не дура, то насколько об этом не жалеет.

// конец Воспоминания

Ещё одна история из жизни, на тему резюме. Молодой человек знакомой девушки — ну бывает такое — работал программистом. И он решил поискать работу, тоже программистом, но подороже и в Москве. И он написал резюме. В те времена ещё жила так называемая «Сеть Друзей» ФИДО. Сетью друзей она называлась по той веской причине, что там все друг к другу обращались только на «ты» и высшей

доблестью считалось опустить друга хлебальником в навоз, виртуально, конечно. Ещё там было десять мужиков на одну девушку, но я сейчас не об этом.

Среди прочих, там была эха (форум, по-нынешнему), посвящённая поиску работы. Точнее их было две. Одна московская — там тусовались москвичи, а другая питерская — там собирались более нормальные люди. В промежутках между мерянием достоинствами, любимым их занятием было изучение резюме, которые соискатели выкладывали туда для тестирования. Туда, в питерскую эху, мы и выложили резюме соискателя.

Питерцы поразмышляли, дали пару мелких советов по улучшению и задали один крупный вопрос — между двумя периодами работы претендента в качестве программиста имеется пятилетний пробел. Где он был в это время? Не сидел ли? А если сидел, то не по обидной ли статье? Претендент сообщил, что эти пять лет он работал осветителем в Театре Юного Зрителя. На что ему ответили, что лучше бы он... Ну, в общем, понятно. А мораль?

Не всяко слово в строку пишется. © А.С.Пушкин «Борис Годунов».

Ещё рекомендую чаще вспоминать древнерусскую пословицу «В чужой монастырь со своим уставом не ходят». Пришёл как-то проситься на работу молодой человек, у которого в резюме было всё — тридцать языков программирования, двадцать СУБД и десять операционных систем. Ещё были перечислены всяческие прикладные программы — от Photoshop до Partition Magic. И много-много всякого другого. Его не взяли не глядя. Не взяли, не глядя на него, хватило взгляда на это резюме. Девушка, которая его привела, очень возмущалась, аргументируя тем, что «Это нормальное резюме, в Москве только такие пишут». До неё никак не доходило, что у нас тут совсем не Москва. А какое дали ему формальное объяснение причины неприёма на работу? «Фортран не знает!». Действительно, в списке языков Фортрана не было. Какая отсюда мораль?

На каждую хитрую гайку найдётся свой болт с резьбой © Народ

Второе требование чуть сложнее — прямо здесь, не отходя от кассы, написать несложную программу. Сначала я ограничивал время двадцатью минутами и старался для каждого подобрать индивидуальное задание. По мере того, как моя вера в человечество падала, время я лимитировать перестал. Потом перестал давать разные задачи — давал только одну. Называлась эта задача — пузырьковая сортировка. Вы знаете, что это такое? Как, вы не знаете? Впрочем, неважно. Со временем я стал при выдаче задания для тех, кто не знает, объяснять алгоритм. Половина претендующих на звание программиста всё равно тест не проходила. Вы думаете, это всё, и уже можно смеяться? Нет, как говорит юморист Задорнов, смеяться ещё рано. Смеяться будете здесь.

Однажды ко мне пришёл наниматься на работу юноша, студент факультета информатики местного колледжа. Он не просто так пришёл, он был прислан его

руководителем, как лучший студент курса. Я попросил лучшего студента курса запрограммировать пузырьковую сортировку. Он согласился и сказал, что об этом он что-то слышал. Время я не ограничивал. Я уже был очень добр к кандидатам. Минут через сорок пять он подошёл и сказал, что программу, к сожалению, написать не сможет. Он, к сожалению, не умеет программировать циклы. Но, он считает, что это неважно, поскольку он специалист не по циклам, а по базам данных, по реально крутым базам данных — ну там Oracle, то сё...

Я был очень расстроен, потому что считал, что при должном воспитании на рабочем месте вполне можно было бы его обучить программированию. Ну, примерно как зайца из поговорки играть на барабане или заставить летать, как крокодильчиков из анекдота. Теми же способами. Главное, что бы у него было желание и вера в свои силы. Если бы он всё-таки, несмотря ни на что, попросился к нам на работу, я бы его взял. Но юноша ушёл и мы зарыдали. Ну, или почти зарыдали.

А к чему это всё? А к тому, что в результате у меня сформировалось крайне отрицательное мнение об этом колледже. Но я ошибался! Колледж меня посрамил. И всех посрамил! И страну Америку тоже. Всем радоваться полчаса!

Не прошло и двух лет, как в газетах появилось сообщение об обнаружении в стране Америке банды злобных русских хакеров. Нет, никто из них, к сожалению, не оказался моим несостоявшимся сотрудником. Но один из них оказался выпускником этого самого колледжа. И теперь я сижу и плачу. Сижу и плачу. Выбор невелик. То ли я чего-то не понимаю в этой жизни, то ли мне прислали не самого лучшего студента, то ли это очень просто в Америке быть ужасным русским хакером.

//Update

Как сообщает пресса, оглашение приговора отложено.

//конец Андейта

Кстати, владельцы информационных колледжей! На этом месте могла бы быть ваша реклама.

И ещё раз — к чему это всё? К тому, что я хочу показать, как писать очень незатейливые программы, которые я прошу написать на собеседовании с кандидатами. Кто знает, вдруг и вам пригодится.

С высоты птичьего полёта

Далее будет подробное разъяснение и обоснование тезиса, что я провожу собеседование как Д'Артаньян, а все другие интервьюеры — как известно кто.

Дабы не быть совсем уж голословным, я полез в Интернет, и скачал пару сотен задач, которые задают на собеседовании. Почти половина из них были задачи, которые авторам текстов задали на собеседовании. Ещё почти половина был задачи, которые авторы текстов сами задают на собеседованиях другим. Остаток представляли легендарные вопросы, по преданиям задаваемые на собеседованиях в фирмах, в которые авторы предыдущих сообщений и на порог не ступали.

С последней категорией всё было ясно — «Сколько баскетболистов поместится в этой комнате», «Сколько в нашем городе настройщиков фортепиано» и нестарееющее «Почему крышки канализационных люков делают круглыми». Всё ясно с этим в первую очередь потому, что нас это не касается. Мы в эту фирму не пойдём, да и сомневаюсь я, что там и вправду задают настолько тупые вопросы. Но на первые два вы всё-таки ответьте. На третий не надо, потому что ответ знают все.

А вот по мере чтения основной массы вопросов и задач мне начало казаться, что Д'Артаньяны — это все кроме меня, а я этот самый, наоборот. Вопросы и задачи были умные. Вопросы и задачи были очень умные. Я не говорю про вопросы про SQL и 1С. Эти вопросы я даже с Интернета не скачивал, осознавая свою абсолютную ничтожность в этих вопросах. Я о вопросах и задачах чисто по программированию.

Для начала я решил возгордиться и ответить на вопросы, ответы на которые я должен знать. Например, что-нибудь по Delphi. Поехали:

Что нужно написать в коде программы Object Pascal, чтобы программа после компиляций была консольной?

Что-то смутно помню. Там ещё фигурные скобки и доллар.

Чем отличается MessageBox(), MessageDlg()?

А ещё я ShowMessage знаю.

Опишите действия данного кода:

```
MyThread := TMyThread.Create(True);
```

```
MyThread.FreeOnTerminate:=true;
```

```
MyThread.Resume;
```

Не люблю я эти потоки.

Чем отличается Indy от компонента TNMECHO

А это что за бред?

О чем говорит данная операция?

```
InValidateRect(Canvas.handle, NIL, True);
```

Кого-то перерисовали? А параметры зачем?

Какая функция отвечает за загрузку динамической библиотеки?

LoadLibrary. Кажется. И ещё у неё параметры есть. Кажется.

Что из себя представляет Self?

Чисто случайно, ответ на этот вопрос я знаю хорошо.

Короче, кандидатуру свою я немедленно снимаю, и уйду в угол горько плакать и убиваться об стену. Так бы я там в углу и остался, но вовремя вспомнил старый анекдот:

- *Однако, чукчу в Союз Писателей приняли!*
- *Да ты же читать не умеешь.*
- *Чукча не читатель, чукча писатель!* © Народное

Моя работа не отвечать на вопросы, моя работа вопросы задавать. И я стал искать те вопросы, которые задал бы сам — если бы задавал вопросы, на самом деле я только прошу запрограммировать пузырьковую сортировку. Авторы вопросов я, честное слово, не знаю, всё утащено из Интернета. Я сразу отфильтровал вопросы, которые касались односвязных, двусвязных и, какие они ещё бывают, списков. Туда же отправились и вопросы в которых присутствовало слово «*эффективность*» и все вопросы на сообразительность. Осталось несколько экспонатов, мне понравившихся:

Отсортировать массив, чтобы в начале шли чётные числа, а потом нечётные

Этот вопрос мне больше всех других понравился. Вроде бы пустяк, а многие пугаются.

Почему перебор двумерного массива выполняется быстрее по строкам, чем по столбцам?

Я плакал. Во-первых, автор вопроса не знаком с Фортраном, но это не главное. Главное — кого это вообще интересует? Но если бы этот вопрос на собеседовании задали мне, я бы забормотал что-то идеологически выдержанное и ожидаемое интервьюером.

Перевернуть строку без использования дополнительной памяти

А почему только строку, собственно, а не массив? Если для строки есть какая-то специфика, то я её не знаю. Но вопрос правильный, хороший. Главное — несложный.

Изменить порядок слов в строке на обратный. Запрещено использование дополнительных массивов

Это заметно сложнее, и понятно, при чём здесь строки.

Найти самую длинную последовательность в строке

Этот вопрос понравился мне своей бездонной многозначительностью. О какой последовательности идёт речь? Одинаковых символов — ‘aaa’? Или одинаковых цепочек символов — ‘abcabcabc’? Налагаются ли какие-то дополнительные ограничения? Решить задачу тупо и в лоб несложно. Например, для подсчёта последовательностей одиночных символов можно подсчитать сначала сколько вообще в тексте разных символов, их будет немного, даже в случае, если у нас Unicode. Завести массив записей — символ, первое вхождение, длина. И трудолюбиво бегать по массиву. Не. Всё-таки я подозреваю, что вопрос не так прост, как кажется, и от нас что-то скрывают...

Часто прямо спрашивают, как лучше всего посчитать сумму от 1 до N .

Могут также ограничить/подсказать запретом на использование цикла.

Приведено дословно. Я опять плакал. Куда катится так называемое образование? Нас этому в школе учили. Ещё нас учили, не жалея патронов, стрелять из мелкокалиберной винтовки на пятьдесят метров. Нынешних не учат, ни тому, ни другому. Результаты налицо. Кстати, обратите внимание на слово «часто» и задумайтесь. Но вопрос хороший, годный.

В каждом слове строки символ с номером N заменить на заданный символ за один проход по строке

Задача хорошая. Без подвохов и в меру сложная.

Ещё среди вопросов из Интернета было несколько просто прекрасных, но их я встречал много раньше в нейтральных источниках, до появления привычки задавать задачи принимая на работу.

Вот они, три простенькие задачи, чисто программистские. У первой из них автора точно нет, а вторые попались мне, как ни странно, в бумажном журнале «Знание — Сила», который, обалдеть, выходит до сих пор.

1. Поменять значения двух переменных, не используя вспомогательной переменной. Тип пусть будет целый. С переменной плавающего типа программный код будет тем же, но возможны нюансы (какие, почему?). А как решить задачу для булевской переменной? А для строки? А что будет с указателем?

2. Массив целых чисел, длиной 128 элементов. Берём последовательность чисел от 1 до 129 и случайным образом размещаем их в массиве. Для какого-то одного числа места, ясное дело, не находится. Для альтернативно одарённых — это может быть любое число, не обязательно 129. Задача — узнать, какое число пропущено. Ограничения — только один проход по массиву, никаких вспомогательных массивов, структур, списков, указателей, Можно объявить пару тройку скалярных переменных.

С агрономом не ходи, ноги выдерну.

Можешь пару раз пройти с председателем © Высоцкий

А что делать, если массив большой, не 128 элементов, а, к примеру, $128 \cdot 128$?

3. Есть массив целых чисел. Массив, сразу предупреждаю, длинный, пусть в нём миллион элементов. Массив абсолютно случайным образом заполняется числами 1, 2 и 3. Задача — отсортировать массив. Сортировать массивы умеют всё, ну, или, почти все. Поэтому усложняем задачу — сделайте это как можно проще.

Теперь о главном

Поручик Ржевский заменяет актёра. Роль забыл. Суфлёр подсказывает:

- *Куда мне жениться, стар я стал.*
- *Куда мне жениться, стар я стал.*
- *Руки не те.*
- *Не те, понимаешь, руки.*
- *Ноги не те.*
- *Не те совсем ноги*
- *(медленно встаёт).*
- *А ТЕПЕРЬ О ГЛАВНОМ!* © Народное

К моему великому сожалению, главное успел сказать не я. Главное сказано ещё в 2007 году в статье *Why Can't Programmers... Program?* У статьи есть автор, но он в статье ссылается на других авторов, которые, в свою очередь, ссылаются на третьих авторов. По сути статья — это не статья, а набор цитат из других статей, и в конце концов остается непонятным — *Кто сказал Мя?* Но это неважно, главное, что мысли очень правильные. Несмотря на прошедшие годы, оно по-прежнему лежит вот тут:

<http://blog.codinghorror.com/why-cant-programmers-program/>

Как вы конечно сразу поняли, в заголовке статьи автор печально спрашивает — Почему, ****, эти ***** программисты ни *** разу не умеют программировать? Дабы не было желания обвинить меня в очернении, цитата

199 out of 200 applicants for every programming job can't write code at all. I repeat: they can't write any code whatsoever

Из двухсот претендентов 199 не умеют программировать. Повторяю: не умеют от слова вообще

Как известно, американцам свойственна мания величия. Каждая столовка у них fast food restaurant, каждая общага — family hotel. Даже худший фильм всех времён и народов снял американец (я этот фильм посмотрел — пожалуй, да). Понятно, что не 199, а, допустим, 190. В 190 верю, они ведь гуманитарии. У нас по моему личному опыту, программировать не умеют из двухсот где-то 101.

Когда при мне кто-то начинает кого-то или что-то критиковать, я всегда требую выдвинуть встречные конструктивные предложения, если, конечно, критикует не Начальник. В статье есть и конструктив.

Первое, автор предлагает тест, на котором эти 199 и отсеиваются. На всякий случай, сначала тест во всей своей английской неприкосновенности:

Write a program that prints the numbers from 1 to 100. but for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz"

Вывести числа от 1 до 100. Если число делится на три, вместо него вывести Fizz, если делится на пять, вывести Buzz. Если и то и другое, вывести FizzBuzz

Вот по этому адресу

<http://c2.com/cgi/wiki?FizzBuzzTest>

содержится объяснение того, чем эта задача сложна и нестандартна и приводится множество вариантов решения на разных языках программирования. Поскольку среди этих языков отсутствуют Pascal, PL/I и Fortran, читать это всё не обязательно. Попутно предлагается усложнить задачу, сдвинув диапазон с 0..100 на -50..50. Я тоже много чего придумать умею.

В качестве утешения — автор статьи считает, что если вы можете написать цикл на от 1 до 10 на каждом из тех языков, что вы перечислили в своём резюме, то вы на голову превосходите всех конкурентов. Чем автор статьи меня огорчил, так это навязчивым требованием знать, понимать и уметь применять к реальным задачам рекурсию. На мой взгляд, нам это ни к чему.

Из тех, кто закончил со мной университет по специальности Прикладная Математика, половина программировать не то, что не умела, но и не могла. Не умеют — те, кого не научили или они сами учиться на хотели. А не могут те, кого можно долго и с удовольствием бить и пороть розгами, а программировать они всё рано не смогут.

Из тех, кто программировать как будто умеет, всё равно половина программировать не умеет и не в состоянии написать простую программу на пятнадцать строк по заранее заданному и тщательно объяснённом алгоритму. Справедливости ради, никто из них программировать никогда и не пытался.

Продолжение вступления в предисловие

Напоминаю, я всегда прошу претендентов запрограммировать пузырьковую сортировку и даже, если надо, объясняю алгоритм. Но это — объяснение — исключительно потому, что я очень добрый. Недавно я узнал, что в пресловутой стране Америке в таких случаях вежливо просят запрограммировать последовательность чисел Фибоначчи. Вот тут интересный фрагмент из Интернета на эту тему. Оно на C++, но юмор оценить всё равно можно. Оно длинное, потому при-вожу ссылку:

<http://www.willa.me/2013/11/the-six-most-common-species-of-code.html>

Там шесть вариантов написания программы, но вот этот мне особенно нравится, потому что у нас, в реальной жизни, всё примерно так. Оно на английском, но ведь все программисты знают английский язык, просто по умолчанию, правда?

```
// TODO add Javadoc comments
/**
 *getFibonacciNumer
 */
// TODO Should we move this to a different file?
public int getFibonacciNumer(int n) {
    // TODO Stack may overflow with recursive
    // implementation, switch over to
    //iteration approach at some point?
    if (n < 0) {
        // TODO Thisshould probably throw an exception.
        // Or maybe just print
        // a log message?
        return -1;
    } else if (n = 0) {
        // TODO Generalize the initial conditions?
        return 0;
    } else if (n = 1) {
        return 1;
    } else {
        // TODO Spend some time with my family and kids,
        // I've been at work for over 46 hours straight.
        return getFibonacciNumber (n - 1) + return
        getFibonacciNumber (n - 2)
    }
}
```

Ещё, пожалуй, неплохо бы научиться программировать сортировку Шелла. Вдруг какой-нибудь садист преклонного возраста этим заинтересуется. Когда я был маленьким, сортировка Шелла считалась чем-то мистическим и загадочным — «никто не понимает, как оно работает, а оно работает!».

И, чтобы два раза не вставать, запрограммировать факториал, как это не смешно. А почему смешно? А потому, что, как ни программируй, при хоть сколько-то экономном кодировании вся программа занимает одну строку, не считая заголовков процедуры. Но это, разумеется, не наш путь. Мы постараемся растянуть удовольствие.

Лучше, конечно, чтобы помучаться © Товарищ Сухов

Практическая часть. Просто сортировка

Начнём мы с пузырьковой сортировки. Что такое сортировка вообще? Для простоты и наглядности мы будем говорить и писать только о сортировке чисел, целых (integer), или дробных (single). Есть хаотический набор чисел, для конкретики целых, и в количестве пяти штук — (22, 3, 10, 7, 9). На выходе нашей программы должно получиться — (3, 7, 9, 10, 22). Идея понятна. Тот алгоритм, та процедура или программа, которая переводит первое состояние во второе, называется сортировкой.

Теперь, что такое пузырьковая сортировка, и почему именно она так называется? Называется она так якобы по аналогии с пузырьком воздуха, который неизбежно всплывает наверх в стакане жидкости. Алгоритм очень прост и понятен, поэтому сверхумные программисты пузырьковую сортировку не любят. Сверхумные любят сортировку Шелла — её никто не понимает, а она работает.

Как работает пузырьковая сортировка? Очень просто она работает, за что её и любят люди среднего ума, наподобие меня и вас. Мы проходим по массиву. Сравниваем первый и второй элементы, то есть 22 и 3. Если они идут в возрастающем порядке (чего не наблюдается), то всё хорошо. Если нет, то меняем их местами. То есть, получаем массив (3,22,10,7,9). Теперь сравниваем второй и третий элемент, в уже подправленном массиве — 22 и 10. Опять всё не так, поэтому меняем их местами. После первого прохода по массиву до конца получаем (3,10,7,9,22). Обратите внимание, девятка слегка всплыла со дна, а число 22, наоборот, погрузилось.

На этом мы не останавливаемся. Теперь второй проход по массиву. На этот раз погружаться будет число 10, а девятка приподнимется ещё на один уровень к поверхности. Потом нас ждёт очередной проход по массиву. И до каких пор это будет продолжаться? Или, оставляя тот же смысл, но, формулируя чуть более удобно для нас, когда это прекратится? А прекратится это тогда, когда после очередного прохода по массиву мы не поменяем местами ни одного элемента, точ-

нее, не поменяем местами ни одной пары элементов. С этого момента массив считается отсортированным.

Теперь переведем все эти рассуждения на язык циклов, переменных и условных операторов. Входные данные у нас содержатся в массиве. Выходные данные будут тоже в массиве, причём в том же. Предлагаемый (мною) интерфейс процедуры:

```
procedure BubbleSort( var a : array of integer);
```

Обратите внимание, формальный параметр объявлен как то, что называется *open array parameter*. Это означает, что в качестве фактического параметра можно передать массив любой длины. Совет — прочитайте в справочной системе о динамических массивах (*dynamic arrays*), сравните эти и те массивы. Не одно ли это и то же? А почему?

Но этот, любой, массив сначала надо объявить и проинициализировать, а только потом вызвать нашу процедуру, ещё не написанную, кстати. Объявить можно, например, так:

```
const
    N = 5;
type
    TArray = array[1..N] of integer;
.....
var
    A                : TArray;
begin
    A[1]:=22;
    A[2]:=3;
    A[3]:=10;
    A[4]:=7;
    A[5]:=9;
    BubbleSort(A);
```

Также заметьте, что объявление массива у нас идёт в три этапа. Сначала мы объявляем константу — длину массива. Затем, используя эту константу, мы объявляем тип массива. И только потом мы объявляем сам массив. Причины выбора такого извилистого пути должны быть читателю сей книги понятны — особенности языка Pascal, он же Object Pascal, он же Delphi, мы в этой книге считаем известными, и далее не разъясняем.

В результате, интерфейс процедуры у нас есть. И у нас есть даже и тестовая программа для нашей процедуры. Это очень много, я не шучу. Можно даже сказать, что у нас есть всё самое главное. А написать саму процедуру не так уж и сложно.

Сначала попытаемся догадаться, какие нам нужны переменные. Можно конечно, добавлять их в процессе написания. Я и сам так всегда делаю, но у нас

случай уж очень несложный, так что попытаемся объявить переменные сразу. Явно нужна переменная цикла, ведь придётся неоднократно ходить по массиву. Еще нужна переменная, которая обозначит, что уже *всё*, то есть массив отсортирован. Поскольку массив или отсортирован или нет, то значений у такой переменной ровно два, откуда следует, что переменная имеет булевский тип.

Ещё нам нужна вспомогательная переменная целого типа, чтобы поменять местами два элемента массива. Можно и без неё (подумайте, как), но с ней проще и, главное, думать не надо. Вот они, наши переменные:

```
procedure BubbleSort( var a : array of integer);
var
    gamover           : boolean;
    tmp               : integer;
    i                 : integer;
```

Теперь мы переводим нехитрый алгоритм нашей сортировки на язык Delphi. Процедура будет работать до тех пор, пока весь массив не окажется отсортированным. Потому сам собой напрашивается цикл вида «до тех пор, пока», то есть **repeat-until**:

```
repeat
    gamover:=true;
    // а здесь что-то очень важное и самое главное
until gamover;
```

В таком, первоначально-пустом варианте цикл завершается мгновенно. А что будет внутри цикла? А там будет проход по элементам массива от первого до предпоследнего. А почему только до предпоследнего? А потому, что предпоследний ($a[i-1]$) мы будем сравнивать с последним ($a[i]$). Программа развивается дальше.

```
repeat
    gamover:=true;
    for i:=Low(a) to High(a)-1 do begin
        // что-то внутри цикла
    end;
until gamover;
```

Внутри внутреннего цикла мы должны пробежаться по нашему массиву, сравнить соседние элементы и поменять их местами, если надо, конечно. Проще не бывает. Получаем вот такое:

```
repeat
    gamover:=true;
    for i:=Low(a) to High(a)-1 do begin
```

```
if a[i] > a[i+1] then begin
    tmp:=a[i];
    a[i]:=a[i+1];
    a[i+1]:=tmp;
    gamover:=false;
end;
end;
until gamover;
```

Теперь неплохо бы обсудить только что написанную программу. Обсуждать особенно нечего. Тем не менее, нельзя не сказать о главном — алгоритм полная дрянь. Хуже алгоритма в природе не было, нет, и не будет. Это не я так говорю, это эксперты по этому делу (по сортировке) так говорят. Говоря по простому, проблема с алгоритмом в том, что пожранное им на сортировку время пропорционально квадрату количества сортируемых чисел. А хуже этого уже не бывает. Нет, конечно, если постараться то можно. Но тут у нас будет цитата из фильма «Формула любви» — *за месяц одному не успеть, тут помощник нужен.*

Тем не менее, если надо быстро запрограммировать сортировку, а в реальной жизни программиста такое странное желание временами возникает, я программирую именно её, сортировку пузырьковую. И, при современном уровне быстройдействия процессоров, и при числе элементов массива до тысячи, эффективность алгоритма никакой роли не играет.

И, кроме того, при приеме на работу я прошу запрограммировать именно *это*, то есть пузырьковую сортировку. Мы с вами вряд ли когда-нибудь пересечёмся, но, скорее всего, я не один такой, и предложить запрограммировать *это* может и кто-то другой. Лучше подготовиться заранее.

Зачем нам сортировка Шелла?

Для прохождения собеседования сортировка Шелла не нужна. И для работы она не нужна. Для собеседования она не нужна потому, что если вас попросят запрограммировать сортировку вообще, вы уже способны запрограммировать пузырьковую сортировку, а если закажут сортировку какую-то конкретную, то всё равно угадать невозможно, какую именно попросят.

Тем не менее, прежде чем думать о великой и ужасной сортировке Шелла, стоит, как рекомендуют в популярном фильме, *потренироваться на кошках.* В роли кошки выступает *сортировка расчётской.* Это как пузырьковая сортировка, только лучше. Впрочем, по мнению некоторых экспертов этого дела, это та же пузырьковая сортировка, только хуже. Потому что, если у нас есть простая и тупая вещь, которая работает, улучшать ей не надо. Это вроде как молоток — зачем его улучшать? Я знаю, я этих молотков столько на уроках труда в школе вырубил из цельного куска металла, что и не сосчитаешь.

Но, в качестве упражнения, это неплохо. Почему эта сортировка носит такое нелепое название? А потому, что по-английски она называется *comb sort*. А почему она так называется по-английски? Я прочитал все объяснения, но так и не понял. А если я этого не понял, наверное это плохое, неправильное название.

В чём отличие сортировки расчётной (*comb sort*) от пузырьковой сортировки (*bubble sort*)? В пузырьковой сортировке сравниваются два соседних элемента массива. Основная претензия к этому подходу в том, что если в конце массива содержатся маленькие элементы, которым положено быть в начале, то в начало массива они перемещаются очень медленно, а почему, кстати, подумайте? И это при том, что те крупные элементы, которые вопреки надобности находятся в начале массива, погружаются достаточно быстро, а почему кстати? Формальный ответ — потому, что использованный нами алгоритм несимметричен. А как сделать его симметричным? И стоит ли? К этим вопросам я обещаю вернуться в главе о скорости работы программы и о том, стоит ли вообще обращать на это внимание. Там мы её и обсудим, сортировку расчётной.

В дополнение, просто для расширения эрудиции — раньше, много лет назад, сортировки различались на внутренние и внешние. Такое разделение можно встретить и сейчас, в не очень старых учебниках, написанных очень старыми людьми. Внутренняя сортировка — это когда сортируемые данные помещаются в оперативной памяти. Внешняя, когда не помещаются, не по капризу разработчика, потому что ему так захотелось, а потому что данные огромного размера, и приходится использовать пространство на диске, в широком смысле слова. Сейчас разница между памятью и диском заметно сгладилась, хотя, разумеется, осталась. Но главное — если ваши данные, предназначенные для сортировки, не помещаются в оперативную память — здесь что-то не так! Надо менять алгоритм, но не алгоритм сортировки, а всей программы.

И факториал!

Очень любимая задача, мною и всеми другими авторами книг по программированию. Иногда встречается и в реальной жизни. Врать не буду, я ни разу не программировал факториал по работе и ни разу не использовал встроенную функцию для него — я даже не знаю, существует ли она. Сначала напомним, что такое факториал. Если по умному — факториал — функция, определённая для целого неотрицательного аргумента.

$$n! = 1 * 2 * \dots * (n-1) * n, \text{ где } n \geq 0$$

Если по простому — факториал это произведение чисел от единицы до аргумента функции, причём аргумент положительный или ноль.

$$4! = 1*2*3*4$$

Это должно быть понятно, потому что проще этого ничего не бывает. Соответственно, мало что может быть проще программы расчёта факториала. Тем не менее, эту задачу любят задавать на собеседованиях — во-первых, потому что это быстро, во-вторых, потому что есть простор для фантазии, для фантазии того, кто задаёт задачу.

Сначала напишем простой и очевидный вариант. Я стараюсь показывать, как программа возникает и как она пишется по шагам, но здесь показывать процесс создания бессмысленно, программа возникает из ниоткуда, сразу и готовая.

```
function FactorialTraditional(      N : integer) : integer;
var
    i          : integer;
begin
    result:=1;
    for i:=2 to N do
        result:=result*i;
    end;
```

Вопросы есть? Вопросов нет. Почему цикл начинается с двух, и так понятно. Печаль в том, что ни один сколько-нибудь уважающий себя садист-экзаменатор в такой форме задачу не поставит. Иначе он будет не садист.

В чём смысл рекурсии? Правильно, в том, что функция вызывает сама себя. Печаль нашей ситуации в том, что вообще-то непонятно, зачем нам это делать и чем плохо наше первоначальное решение. Действительно, иногда (редко) бывают ситуации, когда с рекурсией проще, но у нас явно на тот случай. Но, как говорили в Советские Времена, *«Задачи поставлены, цели ясны, за работу, товарищи!»*.

Вот, ещё раз, наш факториал от четырёх:

$$4! = 1*2*3*4$$

Мы должны вызывать из нашей функции для расчёта факториала её саму, то есть ту же самую функцию для расчёта факториала. В переводе на математический язык это означает, что мы должны выразить факториал четырех через какой-то другой факториал. А какой именно? Мысль выразить факториал четырёх через факториал пяти мы очень быстро отбрасываем, потому что это никуда нас не ведёт. Зато легко заметить, что первые три сомножителя представляют собой факториал трёх:

$$4! = 1*2*3*4 = (1*2*3) * 4 = 3!*4$$

Факториал трёх можно представить аналогично через факториал двух.

К счастью, он вовремя вспомнил одно из изречений Червня: «Начав стрелять, не забудь остановиться» © Козачинский, Зелёный фургон

С рекурсией абсолютно то же самое. Самое сложное в рекурсии определить тот момент, когда рекурсии больше не надо, а функция должна дать чёткий и ясный ответ, то есть не вызывать кого-то ещё, а вернуть просто конкретное число. В нашем случае это не очень сложно. С одной стороны понятно, что если аргумент функции равен единице, то для функции вызывать саму себя уже не надо. С другой стороны, факториал единицы равен единице. В результате имеем вот такой текст:

```
function FactorialRecurcive(      N : integer) : integer;  
begin  
    if N = 1  
        then result:=1  
        else result:=N*FactorialRecurcive(N-1);  
end;
```

Если вам рекурсивный вариант нравится больше не рекурсивного, то я бы вас на работу не взял, и вообще, при встрече перешёл бы на другую сторону улицы.

ГЛАВА 2

Не совсем учебная. Графики

Немного о графиках

Сначала я хотел написать страницу, одну, буквально. Потом получился маленький раздел, плавно переросший в три. На этом этапе я осознал необходимость отдельной главы. Причина такого стремительного роста не в сложности самой задачи, задача с первого взгляда не выглядит сложной. Причина скорее в том, с какой стороны на задачу смотреть. Или эта задача учебная — и тогда действительно хватит одной, максимум двух страниц. Или эта задача реальная — тогда этой главы не хватит — уж слишком много я знаю.

Сама по себе задача очень простая и часто встречающаяся в реальной жизни простого программиста. Нарисовать график. Графики, понятное дело, бывают разные. Я не о графиках в виде разноцветных объёмных столбиков, кругов, разделённых на цветные секторы и прочее, прочее. Я о простых, незатейливых графиках, тех, которым меня учили в школе. А вас учили? Ось абсцисс, горизонтальная (ось X), ось ординат, вертикальная (ось Y). Между ними нарисована наша кривая, она же график. Это всё, чем мы будем здесь заниматься. Поверьте, этого более, чем достаточно.

Мы, сначала, нарисуем график в виде отдельных точек. Это просто, но имеет смысл только в случае очень гладких кривых, наподобие синуса в растянутом по горизонтали масштабе. Если это сразу непонятно, то вы быстро поймёте. Потом мы нарисуем график так, как надо, с соединением точек линиями — прямыми, но уже и это совсем неплохо, результат будет гораздо лучшим.

Сначала немного теории. Теория очень простая, а терминология — первая попавшаяся. Координаты бывают физические и математические. Или реальные и геометрические. Или не координаты, а величины. И вообще, эта классификация часто вводит в заблуждение, сейчас поймёте почему.

Мы строим график. По горизонтальной оси у нас дни, по вертикальной градусы Цельсия. Дни и градусы — это физические величины, они же координаты. Обратите внимание, что они разные. Градусы и дни друг в друга в обычной жизни не переводятся. График мы строим на бумаге в клеточку. По горизонтали у нас один день это одна клеточка, а по вертикали один градус тоже одна клеточка. Клеточка — это математическая или геометрическая величина. Все клеточки одинаковы, что по горизонтали, что по вертикали. Соотношение «один день равняется одна клеточка» называется масштаб (по горизонтали). Соотношение

«один градус равняется одна клеточка» тоже масштаб, но по вертикали. Масштабы разные, хотя численно выражаются одинаково. Разные они потому что относятся к физическим величинам, выраженным в разных единицах измерения. Извините, что я так подробно, многие не понимают.

Теперь о путанице. Как только мы переходим от бумаги в клеточку к монитору или принтеру, происходит чудесная метаморфоза. Наши физические величины превращаются в невидимую последовательности байтов и битов. Наши геометрические величины становятся абсолютно реальными пикселями на экране или точками на бумаге. Дополнительную путаницу вносит ситуация, когда по обеим физическим осям у нас не конкретные дни и градусы, а совершенно безразмерная математика. Но хватит о грустном.

Теперь практика. Начнём мы с синуса. Можно и с косинуса, как кому нравиться. Тангенс с котангенсом подходят меньше, ввиду своего не совсем хорошего поведения в определённых местах.

Наш план действий:

- сначала пишем тупую программу для рисования синуса точками
- затем пишем туповатую программу для рисования синуса отрезками
- пишем правильную программу для рисования произвольной функции (математической), заданной в качестве функции (программной). Как частный случай, эта функция будет рисовать и синус
- завершённый вариант программы продаём заинтересованному потребителю за 999 рублей 95 копеек — мы ведь не жадные, зачем нам тысяча.

Как всегда, есть нюансы. Рисовать сразу сам график — это не совсем правильно. Правильно было бы сначала нарисовать оси координат, к которым этот график будет привязан. А ещё лучше будет пририсовать к этим осям разметку, а иначе какой вообще смысл графика с непонятными кривыми, непонятно какие значения имеющими. Но мы пойдём намеченным путём — сначала примитивный график, потом правильный график, а разметка напоследок. Как часто бывает в программировании, основные усилия уходят на предметы совершенно второстепенные.

Не сомневаюсь, что вы это отлично знаете, но всё-таки напомним. Ось абсцисс (ось X) в программировании расположена так же, как и в математике, слева направо. А вот ось ординат (ось Y) расположена строго наоборот. В математике снизу вверх, а в программировании сверху вниз. Понятно, что, строго говоря, программирование здесь не при чём — так заложено в аппаратном обеспечении, иначе говоря, в железе. И только сейчас, именно в этот момент, я задумался — а собственно почему? Почему разработчики разных стран, разных национальностей, вероисповеданий и партийных принадлежностей заложили в свою аппаратуру это странное «сверху вниз»? Не знаю ответа.

Пусть для начала и для определённости, наш экран, точнее наше окно, или наша форма имеет размер 800x600. Рисуем оси. Здесь «сверху вниз» и наоборот никакой роли не играет.

Пишем простую программу. Сразу скажу, что если кто-то на собеседовании такую тупую и примитивную программу не задавая уточняющих вопросов за пятнадцать минут, хотя бы пятнадцать, напишет, то результат будет зачтён как положительный, причём не просто положительный, а даже очень. Но немедленно после одобрения и похвалы я объясню, по какой массе причин такая программа никуда не годится.

// Воспоминание о чудной молодости

Кажется, в этой книге я ещё ни разу не рассказывал случай из моей программистской молодости? Много этих случаев сохранилось в моей цепкой памяти. Тогда я был совершенно невинным старшим инженером, но в продвинутой конторе. Продвинутой не в смысле секретной, предыдущая контора в плане секретности была круче. Продвинутой в смысле приносимой нашей Родине пользы. Она, контора, хотя и немного другая и немного под другим названием, в основном нашу Родину и кормит и поит до сих пор.

Так вот, ко мне подошла очень образованная женщина (два высших образования, кандидат наук), и совсем даже не старая для такого образования, и попросила дать ей секретный алгоритм рисования графика, а лучше всю программу целиком. Я был молодой и глупый и не понимал, чего ей от меня на самом деле надо и стал честно объяснять, что никакого секрета тут нет, всей программы на одну строчку, вот она строчка... Но она на меня обиделась и нажаловалась руководству.

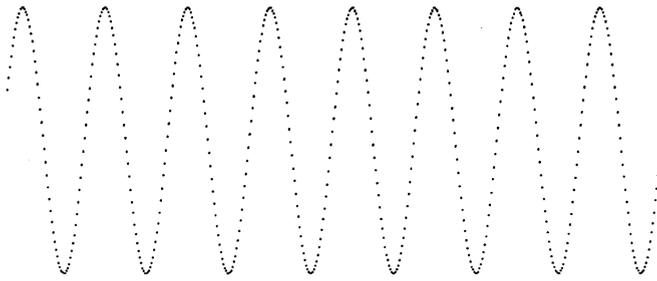
Сейчас она, судя по Одноклассникам, где-то в стране ФРГ. Понятно, почему их Евросоюз стремительно катится куда-то в отхожее место.

// конец Воспоминаний о чудной молодости.

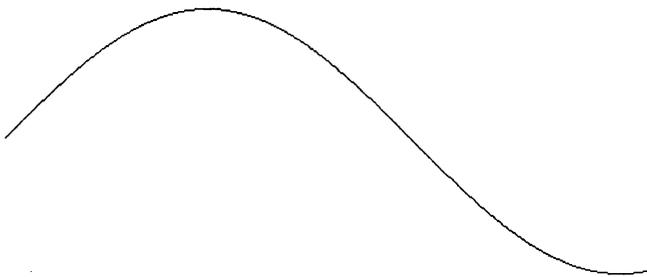
Вот первый, причём *в принципе* работающий, вариант программы.

```
var
  x, y                : single;
  i                   : integer;
begin
  for i:=1 to 500 do begin
    x:=0.1 * i;
    y:=Sin(x);
    Canvas.Pixels[i, -Round(y*100)+100]:=clBlack;
  end;
end;
```

Мне кажется, что это самая простая, из всех возможных, программа рисования синуса, не прибегающая к полухакерским фокусам и вообще к мелкому жульничеству. Тем не менее, даже такую короткую программу надо прокомментировать. Короткая не всегда означает — простая. Кстати, а вот и результат, выглядит неубедительно, честно говоря:



Теперь комментарии. Почему 500? А просто так, я решил посчитать синус в пятистах точках. Обратите внимание, посчитать, а не нарисовать. Потому что в следующей строке нас встречает загадочная константа 0.1. А зачем? Встречный вопрос — а что было бы, если бы её не было? Попробуйте. Наш вполне приличный синус, чистенько, но бедненько, по заклинанию злого волшебника превращается в хаотическое скопление точек. Вполне естественно — если вычислять значения синуса с шагом, равным единице, результат будет очень мало похож на синус (а почему?). Зато если поменять константу на 0.01, то график наш приобретает прямо-таки невероятную красоту. Попробуйте. А почему мы сразу не выставили 0.01?



А потому, что мы не ищем лёгких путей. Мы хотим нарисовать синус с той дискретизацией, с которой мы хотим его нарисовать, а не с той, с которой он лучше выглядит на нашем конкретном устройстве отображения. Синус — хорошая, гладкая функция. Ещё синус то, что называется «аналитическая функция». В реальной жизни, не на собеседовании, аналитические функции встречаются очень редко. Обычно, мы получаем на вход хаотичный поток данных, хорошо, если не в реальном времени, и его надо красиво нарисовать. Поток данных вовсе не заботится от том, чтобы стать гладкой функцией. Об этом после, а пока продолжаем разговор о нашей программе.

А почему минус перед скобками? А я уже говорил. Потому, что в нашей, программистской, не математической, графике, ось X располагается как у нормальных людей, а ось Y — вверх ногами. Это не потому, что программисты идиоты. Точнее, не все программисты идиоты, далеко не все. И не только все.

Зачем мы умножили Y на 100? Потому что естественное значение синуса, не в военное время, редко превышает единицу по абсолютной величине. Если оставить синус не умноженным ни на какой множитель, то такой синус будет, после округления до целого, принимать всего два значения — ноль и единица. Такой синус нам не нужен, умножение на сто растягивает нашу функцию по вертикали, и график становится осмысленным. Напоминаю, что округлять до целого, это не прихоть, а необходимость — координаты пикселей пока принимают только целые значения. (Вы уверены? Проверьте!).

Зачем мы прибавили 100? А потому, что значение синуса бывает и отрицательным, причём ровно в половине случаев. Эта часть графика просто срезается верхней границей формы. А почему именно 100, а не другое число? Потому что максимальное значение синуса по абсолютной величине — единица, а мы умножили эту единицу на сто, вот и приходится сдвигать тоже на сто, извините.

Вопрос для специалистов в Delphi. А кому принадлежит объект Canvas? А какая разница? В нашем конкретном случае он принадлежит форме, на котором лежит кнопка, обработчик которой мы написали.

Немного о графиках. Продолжение

Удивительным образом, очень простая даже не программа, а только фрагмент программного кода — приобретает загадочное и монстрообразное наполнение. Но пока мы говорили только об особенностях и о достоинствах нашего кода, теперь поговорим о недостатках. Недостатков у нас столько, что непонятно, с чего начать, и недостатки эти плавно перетекают один в другой, сворачиваясь в клубок. Поэтому, начнём с того дефекта, который исправлять надо обязательно, и исправлять его просто, потому что с остальными проблемами он мало связан.

Верните константу 0.1 на место, если уже не вернули. Запустите программу, полюбуйте ещё раз на результат. Синус у нас почти как настоящий, это хорошо, но состоит из отдельных точек, между собой несвязанных, что совсем нехорошо. Когда было 0.01 — было гораздо лучше, была красивая гладкая кривая. Почему нельзя навсегда поменять константу на 0.01 и успокоиться? Потому, что кривые бывают разные, и такие гладкие и неторопливо меняющиеся, как синус, попадаются редко. Кроме того, не показалось ли вам, что при замене константы синус не только становится красивым, но и меняет форму, точнее период? Проблему надо решать глобально, для общего случая.

Соединим точки линиями. Первую со второй, далее аналогично, а точку номер $N-1$ с точкой номер N . Правильно ли это? Нет, неправильно. Как можно соединить первую со второй, если вторая точка ещё не нарисована? Меняем концепцию. Вторую точку соединяем с первой, третью со второй, и далее по списку до того момента, как соединим точку N с точкой $N-1$. А первую точку мы ни с чем не соединяем!

В итоге:

```
var
  x, y           : single;
  xP, yP        : integer;
  xPlast, yPlast : integer;
  i             : integer;
begin
  Canvas.Pen.Color:=clBlack;
  Canvas.Pen.Width:=3;

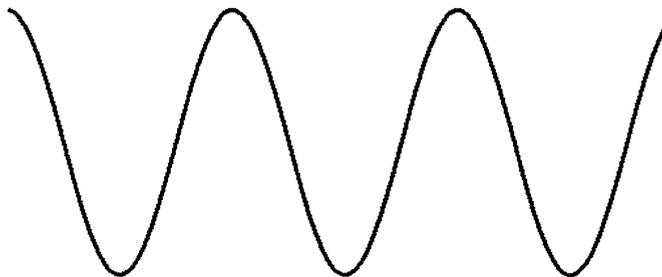
  for i:=1 to 500 do begin
    x:=0.1 * i;
    y:=Sin(x);

    xP:=i;
    yP:=-Round(y*100)+100;

    if i = 1 then begin
      Canvas.Pixels[xP, yP]:=clBlack;
    end else
    if i >=2 then begin
      Canvas.MoveTo(xPlast, yPlast);
      Canvas.LineTo(xP, yP);
    end;

    xPlast:=xP;
    yPlast:=yP;
  end;
```

Это был результат в виде программного кода, а это результат графический:



Два кратких комментария, для тех, кто программировать уже умеет. Первый — для чего объявлены переменные yP и $yPlast$ — понятно. А для чего объявлены xP и $xPlast$, ведь можно использовать переменную цикла i ? Ответ — это зарезервировано на будущее.

Второй — почему методы MoveTo и LineTo используются тупо парой, да в придачу к ним ещё и свойство Pixels? Ведь было бы гораздо проще и, не побоюсь этого слова, изящнее, использовать эти методы, как и положено, по одиночке. Точнее, вместе, но в разных местах. Зачем каждый раз перемещать перо (Pen) туда, где оно и так уже находится? Причин множество, и я их даже полностью перечислять не буду. Назову только первую — ну не нравится мне этот нелепый подход (MoveTo, LineTo), не нравится! Поверьте, что остальные причины настолько же обоснованные и аргументированные.

Кстати, обратите внимание, что кривая наша всё равно, несмотря на все наши усилия, какая-то не очень гладкая, зазубренная. Что делать? В реальной жизни, для реальных данных, это совершенно не важно — реальные данные, сами по себе, такие же зазубренные, да ещё и искажённые объективными обстоятельствами, да ещё и с наложенными поверх них помехами — систематическими и отдельными выбросами, так что вся эта зазубренность будет просто незаметна. Если же вы работаете с приличными аналитическими функциями, то надо, всего-навсего, заменить линейную интерполяцию, которую мы нечаянно сейчас применили, даже не зная этого слова, на что-то более сложное. Например, если вы используете Delphi — сплайны, точнее, кубические кривые Безье. Метод TCanvas.PolyBezier.

Итак, лёгкую изолированную проблему мы решили. Теперь приступаем к разматыванию основного клубка. Первое, что бросается в глаза — константа 500, которую хочется или заменить переменной, или сделать именованной константой, но пока мы об этом думать не будем. Это серьёзно и на самом деле сложно, об этом подумаем потом.

Кровь сосать решили погодить — Вкусное на третье © Высоцкий

А пока сделаем самое необходимое — оформим это всё как процедуру. Это первое. А второе — как параметры передадим место (в пикселах, на форме) где будет отображаться начало отсчёта математических координат [0,0]. Сейчас начало координат математическое [0,0] совпадает с левым верхним углом формы (0,0) и нам приходится искусственно сдвигать график вниз (+100).

Теперь мы при рисовании (MoveTo, LineTo, Pixels) прибавляем к координатам параметры x0 и y0, а это самое (+100) убираем. Чтобы два раза не вставать, передаём в процедуру цвет и толщину линии:

```

procedure Graphic(      x0,y0 : integer;
                        color : TColor;
                        width : integer);
var
    x,y                  : single;
    xP,yP                : integer;
    xPlast, yPlast      : integer;
    i                    : integer;

```

```

begin
  Canvas.Pen.Color:=color;
  Canvas.Pen.Width:=width;

  for i:=1 to 500 do begin
    x:=0.1 * i;
    y:=Sin(x);

    xP:=i;
    yP:=-Round(y*100);

    if i = 1 then begin
      Canvas.Pixels[x0+xP, y0+yP]:=color;
    end else
    if i >=2 then begin
      Canvas.MoveTo(x0+xPlast,y0+yPlast);
      Canvas.LineTo(x0+xP,y0+yP);
    end;

    xPlast:=xP;
    yPlast:=yP;
  end;
end;

```

Вызов будет, к примеру, вот таким:

```
Graphic( 100,150, clGreen, 3);
```

Немного о графиках. Финиш

А теперь о главном. Внешние проявления болезни — у нас до сих пор осталась константа 500 — количество выводимых точек, константа 0.1 — сколько радиан приходится на один пиксел, если вы ещё не догадались. А главное, никому и никогда не нужен *слепой* график — тот, по которому невозможно определить чему равно значение функции и в какой точке оно этому значению равно. А у нас нельзя. Потому что у нас нет осей координат, на которых были бы размечены физические единицы. Физические, в нашем конкретном случае синуса, являются скорее математическими — действительная функция действительного переменного, естественно не имеющая размерности. В жизни это может быть, например, функция температуры от времени. Задаётся она, само собой, не аналитически, а таблично. А ещё они имеют размерность — время измеряется очень часто в секундах, а температура обычно в градусах Цельсия.

Обсуждать размеченные оси координат бессмысленно, не подумав сначала о масштабах. А какой у нас сейчас масштаб? А что такое масштаб? Каждый знает что такое масштаб из уроков географии — сколько километров приходится на один сантиметр карты. И здесь то же самое — сколько секунд или градусов (Цельсия) приходится на один сантиметр. Проблема в том, что у нас сантиметров как таковых нет — у нас есть только экранные пиксели. Мы легко можем задать масштаб в секундах на пиксел для экрана, но пересчитать его для сантиметров бывает затруднительно.

Для принтеров задача решается точно и гарантированно, а для монитора это очень и очень трудно. С другой стороны, для принтера это очень и очень важно и за малейшую погрешность вас будут очень и очень бить. Есть диаграмма, выведенная на бумагу, на ней обязательно написан в заголовке масштаб, сколько физических единиц в одном сантиметре бумаги. Рано или поздно кто-то приложит линейку, произведёт в уме несложную математическую операцию и спросит — «*Вчера ведь было всего минус тридцать, а тут что за ерунда?*». И за вами придут, и будут тыкать мордой в эту бумажку, как Ваньку Жукова в ту селёдку (в вашей школе ещё проходили Ваньку Жукова?). Запомните, бумага это очень и очень серьезно.

Другое дело монитор. На мониторе вы рисуете шкалу с делениями в сантиметр, то есть это вы так говорите, что деления в сантиметрах. Если приложить к ним реальную железную линейку, то видим реальных четырнадцать железных миллиметров, или двенадцать, или семнадцать. Не волнуйтесь, никто и никогда этого не заметит, а если заметит, то это никого не заинтересует. Поэтому, волевым решением для всех практических целей назначаем, что в одном сантиметре экрана у нас 30 пикселей (или пикселов). Причём, независимо от размера монитора и разрешения экрана. Если результат вам покажется неубедительным, замените тридцать на сорок, какая разница.

График не всегда рисуется с нуля, то есть в начале координат не обязательно находится точка, соответствующая нулю секунд или нулю градусов. К примеру, нас интересуют данные, начиная со второго часа (3601-я секунда). К тому же у нас лето. Или мы в тропиках. Ну, нет у нас нуля. Или наша температура вообще не температура воздуха, а температура внутри чего-то, где холоднее плюс двести не бывает.

А сейчас мы напишем волшебную формулу. Сначала пишем для горизонтали. Какие переменные в эту формулу войдут:

Нач — начало отсчёта, в физических единицах, например 60 секунд

Кон — конец отсчёта в физических единицах, например 360 секунд

Msm — масштаб в физических единицах на сантиметр экрана. Пусть будет 60 секунд/сантиметр

Psm — пикселов на сантиметр, всегда (временно всегда) 30

pW — ширина (размер по горизонтали) поля вывода (по оси абсцисс)

А вот и сама формула:

$$((\text{Кон} - \text{Нач}) / \text{Msm}) * \text{Psm} = W$$

Psm у нас константа, про неё можно забыть. Остаются четыре переменных. Понятно, что если три из четырёх заданы, то четвёртая определяется однозначно. Что из этого следует, в переводе на язык программирования? `Const Psm = 30.0;` Это первое. На вход процедуре рисования графика необходимо и достаточно передать три параметра, по нашей воле выбранных из четырёх переменных. А какие именно? Не аргументируя, привожу готовый ответ — передаём начало отсчёта (физическое), масштаб и ширину поля. Заголовок процедуры выглядит так:

```
procedure Graphic(      begX  : single;
                        MsmX  : single;
                        pW    : integer;
                        x0,y0 : integer;
                        color  : TColor;
                        width  : integer);
```

Что это означает в плане программирования? Только одно — мы рассчитываем на основе первых трёх параметров конец отсчёта (конец интервала отображаемых данных) и выводим данные только в этом диапазоне. А как это реализовать практически? Практически это зависит от условий задачи, которые ставим не мы, а природа в лице начальника и заказчика. Данные могут быть заданы в виде математической (или даже аналитической) функции — это в теоретических, учебных, тестовых, испытательных задачах. В практических задачах данные либо задаются таблично, либо вообще заранее неизвестно даже их количество и они поступают на вход программы-обработчика в режиме реального времени.

Пока рассматриваем первый вариант, то есть — рисуем синус. Приятная особенность этого случая в том, что мы имеем возможность выбрать любой шаг изменения физических данных, само собой именно тот, который нам проще программировать. Логично сделать шаг приращения физической переменной равной значению одного пикселя, пересчитав этот шаг в физические единицы с учётом масштаба. Вводим новую сущность — `MXpix`, это будет количество физических единиц (к примеру, радиан) приходящихся на один пиксел по горизонтали. Считается это просто:

$$MXpix = Msm / Psm$$

Теперь я дам совет, полезный, и, главное, универсальный. Прежде чем программировать что-то, далее называемое *модуль*, напишите для проверки этого модуля тестовую программу. То, что писать тестовую программу надо, понятно для большинства. Но вот то, что писать её обязательно *до* программирования тестируемого модуля, а не *после*, доходит не сразу, а чаще — вообще не доходит. А главное, сразу *после* написания тестовой программы, но *до* её первого выполнения надо аккуратно записать или нарисовать на бумажке, что именно мы ожидаем увидеть в результате её выполнения. Да, вычислять ожидаемый результат мы будем сами, руками, мы, а не компьютер.

Почему это надо делать до, а не после? Потому что человек, а особенно его отдельная разновидность — программист, удивительно склонен к самообману. Если создавать тест *после* написания программного кода, то, чудесным образом, тест будет написан именно так, чтобы обойти все возможные подводные камни и противопехотные мины нашей программы. И это не означает гениальность программиста, абсолютно наоборот. Чем программист хуже, тем вероятнее, что он напишет именно тот тест на котором его программа пройдёт проверку стопроцентно.

В нашем случае тестовая программа будет несложная, буквально из одной строчки.

```
Graphic( pi/2, pi*3, 500, 100,150, clGreen, 3);
```

Последние четыре параметра нам уже прекрасно известны, теперь о первых трёх. Почему первый параметр не ноль? А потому, что тест должен хоть что-то тестировать. Ноль — тривиальный вырожденный случай. С нулём наша программа уже прекрасно работает. В случае π , делённого на два, мы должны получить синус не идущий вверх, а идущий вниз. Если мы всё запрограммируем правильно, конечно. Второй параметр означает, что на три сантиметра дисплея должен приходиться один полупериод синуса. А почему? А потому что мне так эстетически приятно. Третий — эти наши вечные (пока) пятьсот точек по горизонтали. Обратите внимание, что всё это относится к рисованию по горизонтали. По вертикали пока как всё было, так и осталось, хотя там есть свои нюансы — шаг по вертикали выбирать не надо.

А теперь постарайтесь сами переписать нашу процедуру в соответствии с выданным мною вам Техническим заданием (ТЗ). Напишите. Запустите. Посмотрите на результат. Подумайте. Видите ли вы то, что ожидали? А почему?

Вы внимательно прочитали всё предыдущее? Вы над прочитанным думали? Там есть ошибка, я не нарочно, честное слово. Я хотел иметь один полупериод синуса на один сантиметр и написал $\pi*3$. В результате я получил три полупериода синуса на один сантиметр, что, честно говоря, смотрится омерзительно. На самом деле, в тестовом вызове процедуры должно быть $\pi/3$.

Самое время привести программный код, но я пока воздержусь. Причина — надо ещё разобраться с осью ординат. Разница в том, что, как я уже заметил, шаг выбирать не надо (почему?), а остальное так же, как и по горизонтали. Надо задать начало отсчёта и масштаб по вертикали. И ещё высоту поля вывода, но здесь перед нами встает маленькая эстетическая проблема. У нашей процедуры уже семь (!) параметров. Если добавить ещё три, то будет десять, что находится за гранью добра и зла. Разумеется, если вы загляните в справочник по MSDN, то найдёте там и не такое, но мы всё-таки должны стремиться не к худшему, а к лучшему и светлому.

Что делать? Параметров слишком много. Параметры передавать надо.

Кот поступил как всякий, кого изгоняют из трамвая, но которому всё-таки ехать-то надо © Булгаков «Мастер и Маргарита»

У нас уже есть ширина поля и его левый верхний угол. Теперь к этому богатству добавляется и высота. Всё вместе полностью и однозначно задаёт прямоугольник. Таким образом, все четыре параметра мы легко объединяем в один параметр типа `TRect`. В Delphi есть какой-то хитрый нюанс — левая верхняя граница в этот прямоугольник входит, а правая и нижняя нет, но мы на это внимания обращать не будем, вообще. Если мы случайно запрограммируем правильно, то хорошо, а если неправильно и ошибёмся на одну точку, то никто не заметит.

Теперь о важном. С горизонталью понятно. Левая граница нашего прямоугольника задаёт начало, в пикселах, откуда рисовать, правая граница — где останавливаться, опять же в пикселах. Шаг отображения по горизонтали у нас фиксирован — ровно один пиксель, так что никаких проблем нет. По вертикали сложнее — мы не можем знать заранее, какое значение примет функция в какой-то конкретной точке. Поэтому, если мы хотим, чтобы наш график уместился в наш прямоугольник, то возможностей у нас три.

Первый вариант — определять масштаб по вертикали исходя из диапазона, то есть — высоты прямоугольника. Но мы уже раньше решили, что это не наш путь, мы будем задавать начало отсчёта и масштаб.

Второй вариант — отсекают данные, выходящие за пределы прямоугольника, каким-либо программно-системным способом. Это неспортивно, я такому учить не буду, книга не о машинной графике.

Третий вариант — отсекают данные самим. Напрашивается простая идея — если точка в пересчёте к масштабу и к количеству пикселей на сантиметр, выходит за пределы, мы её не рисуем. При втором рассмотрении мы, однако, замечаем, что получаем в порядке компенсации другую проблему — точки должны быть соединены между собой, а здесь у нас образуется разрыв. На самом деле, конечно, не такая это серьёзная проблема, но для нашей, пока ещё учебной, программы это сильно увеличит и сделает громоздким и неуклюжим её текст.

Поэтому принимаем за окончательный вариант без номера — прямоугольник вообще ничего не ограничивает. Вначале по вертикали у нас передавалось всего одно значение — начало отсчёта. Теперь значений у нас два — `Top` и `Bottom`. Чтобы добру не пропадать, используем оба — пусть начало отсчёта по вертикали находится ровно посередине между этими двумя точками. А ответственность за то, что график впишется в прямоугольник, справедливо берёт на себя тот, кто задаёт масштаб.

Welcome to the real world © то ли кино какое-то, то ли наоборот

На этом мы не остановимся. Упакуем начало отсчёта по горизонтали и масштаб по ней же в запись, и добавим к ним начало отсчёта и масштаб по вертикали. Разумеется, придётся объявить новый тип и переменную этого типа:

```

type
  TScale = packed record
    begX           : single;
    MsmX           : single;
    begY           : single;
    MsmY           : single;
  end;

var
  sc               : TScale;

```

Заголовок нашей процедуры и, соответственно, её вызов приобретают такой вид:

```

procedure Graphic(      sc      : TScale;
                        rect     : TRect;
                        color    : TColor;
                        width    : integer);

```

```

sc.begX:=pi/2;
  sc.MsmX:=pi/3;
sc.begY:=0;
  sc.MsmY:=1;
Graphic( sc,Rect(100,150,600,250), clGreen, 3);

```

Параметры для Y означают, что синус наш должен рисоваться как есть, от нуля — иначе бы мы его просто не узнали, а масштаб соответствует одной единице математической в одном сантиметре экрана, то есть весь размах синуса (-1,+1) должен уместиться в двух сантиметрах экрана. Попутно посмотрите насколько это правдоподобно выглядит, в смысле наши два сантиметра применительно к любому произвольному монитору и его разрешению (произвольный монитор — это ваш). И стоило ли напрягаться из-за каких-то небольших погрешностей в точности отображения? Хорошо, что мы и не напрягались.

Так вот он, почти окончательный гениальный программный код — не хватает только осей и разметки:

```

procedure Graphic(      sc      : TScale;
                        rect     : TRect;
                        color    : TColor;
                        width    : integer);

```

```

const
  pSm = 30;

```

```

var
  x, y           : single;
  xP, yP        : integer;
  xPlast, yPlast : integer;

```

```

MXpix,MYpix          : single;
x0,y0                : integer;
i                    : integer;
{.....}
procedure Axis;
begin
    // пока пустая, извините
end;
{.....}
begin
    MXpix:=sc.MsmX / Psm;
    MYpix:=sc.MsmY / Psm;

    Canvas.Brush.Color:=clWhite;
    Canvas.FillRect(Canvas.ClipRect);

    // прямоугольник, просто для интереса
    Canvas.Pen.Width:=1;
    Canvas.Pen.Style:=psDot;
    Canvas.Rectangle(rect);

    x0:=rect.Left;
    y0:=(rect.Top+((rect.bottom-rect.Top) div 2)) + Round(sc.
begY/MYpix);

// вызов ненаписанной процедуры, рисующей оси
    Axis;

    // а это начинается сам график
    Canvas.Pen.Width:=width;
    Canvas.Pen.Style:=psSolid;

for i:=1 to (rect.Right - rect.Left) do begin
    x:=sc.begX + (i-1)*MXpix;
    y:=Sin(x) - sc.begY;

    xP:=Round(x/MXpix);
    yP:=-Round(y/MYpix);

    if i = 1 then begin
        Canvas.Pixels[x0+xP, y0+yP]:=color;
    end else
    if i >= 2 then begin
        Canvas.MoveTo(x0+xP, y0+yP);
        Canvas.LineTo(x0+xP, y0+yP);
    end;

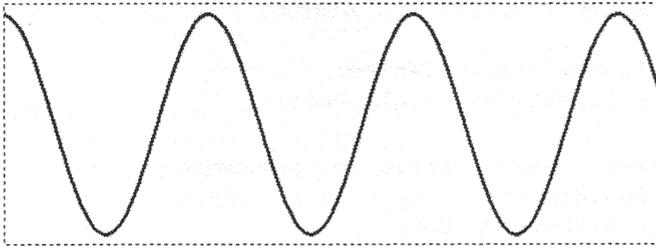
```

```

    xPlast:=xP;
    yPlast:=yP;
  end;
end;

```

Как всегда, гениальности недостаточно, нужны комментарии. Код всё-таки не окончательный. Заодно для контроля нарисован тот самый прямоугольник. Поскольку у вас какое-никакое, но математическое образование есть, вы прекрасно разберётесь, почему в одном месте мы на масштаб умножаем, а в другом месте на масштаб делим.



Попробуйте, протестируйте, попытайтесь сломать. Сломать можно. Сломать можно вообще всё. Подумайте, критично ли это, то что вы сломаете, в смысле насколько это важно для выполнения программой своей задачи. Одно дело, если этой процедурой будете пользоваться вы, ваши друзья и родственники, другое дело, если вы пишете большую, коммерчески ориентированную программу и собираетесь её продавать, и встроить в продаваемый продукт рисование графиков. Графики везде нужны.

Совсем всё. Финиш

Что осталось сделать? Нарисовать оси, сделать разметку и подписать значения. Всё это богатство мы естественно оформим в виде процедуры, причём процедура эта будет вложенной по отношению к процедуре `Graphic`, для того, чтобы получить доступ к переменным и константам в ней (`Graphic`) объявленным. Я не люблю вложенные процедуры, как раз из-за того, из-за чего я её сейчас и применяю — за доступ к переменным, не в ней описанным. Но программа наш отчасти учебная и размещать процедуру `Axis` на том же уровне, что и `Graphic`, поставило бы нас перед нелёгким выбором. Или передавать ей соответствующие значения (например x_0, y_0) как параметры, что легко, но громоздко. Иди сделать их глобальными переменными для обеих процедур, что, по общему мнению, плохо и опасно. Ещё можно оформить всё, что мы уже напрограммировали и ещё напрограммируем в виде класса, это красиво, безопасно и изящно — но сколько возни!

Продолжаем. Первое моё желание — немедленно подписать в начале координат «0» для оси X и «0» для оси Y. Потом я вспоминаю, что начало отсчёта по обеим осям вовсе не обязательно равно нулю, хотя очень часто им бывает. Так что, вместо нулей, пишем в соответствующих местах `sc.begX` и `sc.begY`, точнее, их символическое соответствие.

Теперь надо немного задуматься, а что мы собственно программируем. Если мы пишем эту программу только для сдачи преподавателю и последующему выбросу на помойку, подход один, сами понимаете какой, церемониться нечего. Если мы пишем универсальную процедуру для Excel или Matlab, то требования заметно возрастают. Рассмотрим вариант промежуточный — вы на работе и программируете подпрограмму для конкретного проекта средней важности. Или, как вариант, вы программируете это для себя самого, такая странная фантазия. В этом случае, как минимум, повторяю — *как минимум*, желательнее очень хорошо сделать то, что вам надо или то, что от вас требуют, и ещё немного больше. Больше — потому, что позже неизбежно появятся новые пожелания, от начальника или ваши собственные, и лучше заранее заложиться на них, а то и реализовать их с самого начала. А почему сделать надо только немного — потому, что если вы сразу будете мастерить что-то на порядок больше заданного, то это будет, конечно, хорошо для этого конкретного проекта. Но проектов мало не бывает, вам неизбежно за это время вручат ещё что-то для разработки, и ещё и ещё... А вы всё ещё там...

У нас тут не Мавзолей, не залежишься © Советский анекдот.

Какой из этого вывод? Повторим формулировку задачи и расширим её. Наша задача — отобразить функцию $\text{Sin}(x)$, с заданием начала отсчёта и масштабов. По оси Y физические значения нашей функции не превышают единицы по абсолютной величине. Всё, что нам надо, это подписать значения «1», «-1» и «0», предварительно определив, где они, собственно, находятся на экране. Неплохо было бы и подписать значение в начале координат, тем более, что оно нам уже задано, считать его не надо. Ещё раз — начало координат не обязано совпадать с нулём. Итого имеем три числа через равные интервалы и одно число, что называется, сбоку. Мне кажется, проще и естественнее нарисовать их руками, каждое отдельно, а не в цикле.

Почему я так подробно останавливаюсь на такой с виду ерунде? Потому что это типичный выбор, который приходится делать программисту ежедневно и даже чаще. Сделать быстро и надёжно, а потом расширять функциональность и переделывать. Или сразу предусмотреть все возможные неприятности, но тогда ненадёжно и не успеем. Само собой, никто не отменял вариант «сделать быстро и плохо».

В порядке компенсации, для изящества программного кода, по оси X нарисуем разметку в цикле, до тех пор, пока будет хватать оси абсцисс. И не просто разметим, а с периодом $\pi/2$, и соответственно подпишем — $1/2\pi$, π , $3/2\pi$, 2π . Для

того, чтобы лучше рассмотреть результат, зададим вот такие параметры отображения:

```
sc.begX:=pi/2;
sc.MsmX:=pi/6;
sc.begY:=0;
sc.MsmY:=0.33;
```

Вот она, наша процедура Axis, немного громоздкая, но зато работает. Кажется работает.

```
procedure Axis;
const
  dx = 20;
  dy = 5;
var
  whX, whY           : integer;
  pinadva            : integer;
  stroka              : string;
  i                   : integer;
begin
  // оси
  Canvas.Pen.Style:=psSolid;
  Canvas.Pen.Width:=1;

  x0:=rect.Left;
y0:=(rect.Top+(rect.bottom-rect.Top) div 2)) + Round(sc.begY/
MYpix);

  Canvas.MoveTo(x0, y0);
  Canvas.LineTo(x0+rect.Right, y0);

  Canvas.MoveTo(rect.Left, rect.Top);
  Canvas.LineTo(rect.Left, rect.Bottom);

  // разметка по вертикали
  Canvas.TextOut( x0-dx,y0-dy, FloatToStrF(sc.begY,
ffGeneral, 3,1));

  whY:=y0-Round((0-sc.BegY)/MYpix);
  Canvas.TextOut(x0-dx,whY-dy,FloatToStrF(0,ffGeneral,3,1));
  Canvas.MoveTo(x0-5, whY); Canvas.LineTo(x0+5, whY);

  whY:=y0-Round((1-sc.BegY)/MYpix);
  Canvas.TextOut(x0-dx,whY-dy,FloatToStrF(1,ffGeneral,3,1));
  Canvas.MoveTo(x0-5, whY); Canvas.LineTo(x0+5, whY);
```

```

whY:=y0-Round((-1-sc.BegY)/MYpix);
Canvas.TextOut(x0-dx,whY-dy,FloatToStrF(-1,ffGeneral,3,1));
Canvas.MoveTo(x0-5, whY); Canvas.LineTo(x0+5, whY);

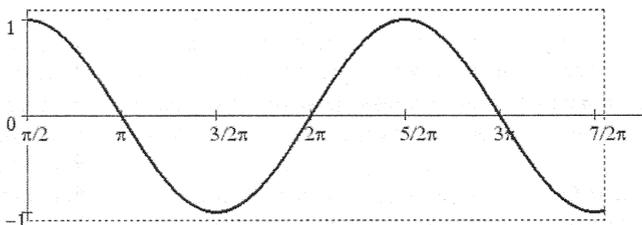
// разметка по горизонтали
pinadva:=Round( sc.begX/(pi/2));

Canvas.Font.Name:='Symbol';
Canvas.Font.Size:=14;

repeat
  whX:=x0 + Round((-sc.begX+pinadva*pi/2)/MXpix);
  Canvas.MoveTo( whX, y0-5);
  Canvas.LineTo( whX, y0+5);

  if (pinadva mod 2) = 0 then begin
    if pinadva = 0 then stroka:='0' else
    if pinadva = 2 then stroka:=#112
    else stroka:=IntToStr(pinadva div 2) + #112;
  end
  else begin
    if pinadva = 1
      then stroka:=#112'/2'
      else stroka:=IntToStr(pinadva) + '/2'#112;
    end;
  Canvas.TextOut( whX-5, y0+5, stroka);
  pinadva:=pinadva + 1;
until (whX >= x0 + rect.Right);
end;
```

А вот так выглядит результат выполнения процедуры, с кривой, осями, разметкой, подписями и прямоугольником.



Можно ли улучшить эту процедуру? Можно, конечно. Самое очевидное — сделать её действительно универсальной. В первую очередь это касается разметки по оси ординат (оси Y). Даже при том, что мы подписали не ней всего три

значения, программный код явно выглядит монотонным, однообразным и сбрасываемыми в глаза повторениями. Их надо организовать в цикл или вынести в отдельную процедуру. Некоторых смущают выражения типа $\cdot/2' \# 112$. Меня самого они до сих пор смущают, как ни странно, но привык. Что важнее, обратите внимание — при рисовании графика мы движемся по геометрическим координатам, с шагом единица, и рассчитываем для них соответствующие физические координаты. При нанесении разметки мы перебираем физические, с шагом $\pi/2$ и рассчитываем для них геометрические. Есть ли в этом какой-то скрытый смысл и является ли это универсальным принципом?

Теперь ещё раз подумаем о том, о чём мы уже много раз думали. Зачем мы это пишем? Для кого мы это пишем? Если только для себя, то, для начала, сойдёт и так. Если для других, то любой пользователь немедленно захочет нарисовать вместо синуса косинус, а то и тангенс — тангенс рисовать труднее, поверьте. Если мы пишем для себя, то можем влезть в исходный текст и руками поменять синус на косинус, хотя нам должно быть очень стыдно — какой из нас после этого программист? Если мы пишем для адекватного пользователя, об этом и речи быть не может. Адекватный пользователь — такой заказчик, который знает, чего он хочет от программы, может это объяснить словами, не прибегая к жестам рук, и понимает, где кончается его компетенция.

// Рассказ об Адекватном пользователе

Главный вывод из моего опыта состоит в том, что адекватный пользователь вовсе не обязан выглядеть с первого взгляда как адекватный пользователь. Обратное, само собой, тоже верно. Однажды мы написали программу. Заказчики были из Плесеца, но это не то, что вы подумали, если, конечно вы вообще что-то подумали. Никакого отношения к одноимённому космодрому они не имели, просто их контора находилась в этом самом городе, а они были люди самого что ни на есть мирного труда. Но в душе романтики, то есть им нравилось программировать. Программировали они на Бейсике, максимальный размер программы — пятнадцать строк, но им это нравилось.

Поэтому, чувствуя себя в глубине души настоящими программистами, и получив от нас заказанную программу, они во чтобы то ни стало захотели получить и её исходный текст. Нам этого категорически не хотелось, потому что случаи разные бывают, и никогда никого добра от передачи исходных кодов ни разу не было. Или украдут, что не страшно, потому что всё равно ворованным исходным текстом никто ещё воспользоваться не сумел. Или, что хуже, внесут изменения, скомпилируют, получат проблемы, обломаются и предъявят претензии за плохую работу. Нам же и предъявят.

Мы не давали. Они требовали. Градус повышался. Начали орать. Я прислушался. Орали всякое, но речь шла и о файлах, которые пишет наша программа. Понятие «файл» заказчикам было знакомо и даже воспринималось совершенно правильно. Файлы их программы, на Бейсике, выглядели как текст. Файлы данных, которые они читали и писали — да, они читали и писали файлы! — при просмотре показывали числа, они писали и читали текстовые файлы. Но когда они смотрели внутрь файлов, записанных нашей программой, то видели там «мордочки». Никакой мистики — посмотрите любой бинарный, не текстовый файл — там обязательно есть мордочки.

С этого момента я понял, что никакой опасности в отдаче исходных текстов не может быть, пошёл и распечатал *всё*. Заказчики, увидев примерно двести страниц текста, программа была не очень большая, вежливо сказали «спасибо, нам уже и не надо», тем самым показав себя Адекватными Пользователями. Взаимовыгодное сотрудничество продолжалось и дальше ко всеобщему взаимодовлетворению.

// конец рассказа об Адекватном Пользователе

Возможно, стоит всё-таки оформить всё это богатство в виде класса? Да, но только если мы собираемся использовать запрограммированное нами долго и всерьёз. Интерфейс класса, в первом приближении, предлагается таким, доработайте напильником, если хотите.

```

unit UsGrGraphic; // 01.12.2014
                  // 03.12.2014
{-----}
interface
  uses
    Types, Graphics;

  type
    TScale = packed record
      begX      : single;
      MsmX      : single;
      begY      : single;
      MsmY      : single;
    end;

  type
    TLegend = packed record
      ///
    end;

  type
    TGoodFunction = function(      x : single) : single;

  type
    TGrGraphic = class
      private
        .....

      public
        property Func : TGoodFunction read fFunc write fFunc;
        property sc   : TScale read fSc write fSc;
        property rect : TRect read fRect write fRect;
        property color : TColor read fColor write fColor;
        property width : integer read fWidth write fWidth;

```

```
property axisOn : boolean read fAxisOn write fAxisOn;
property legend : TLegend read fLegend write fLegend;
```

```
constructor Create( wCanvas : TCanvas);
destructor Destroy; override;
procedure Draw;
```

```
private
  // что-то
```

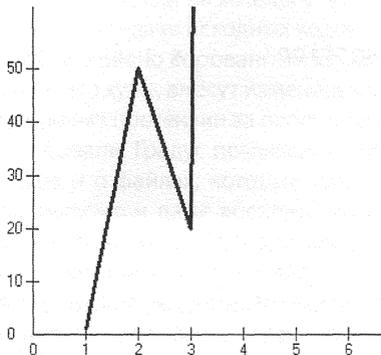
```
end;
```

Есть ли жизнь после финиша?

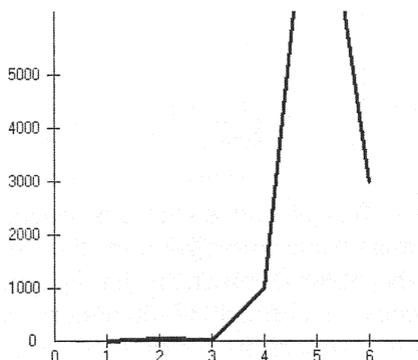
Что дальше? Дальше много чего. Бывает ещё и логарифмический масштаб. Обычно он применяется тогда, когда кривая меняется по вертикали настолько стремительно, что в линейном масштабе уследить за ней становится невозможно. В этом случае по оси Y отображается не сама кривая, а её логарифм. Обратите внимание, логарифмический масштаб бывает только по вертикали, по горизонтали он всегда линейный. Почему? Наверно потому, что две логарифмических оси не смог бы вместить даже самый испорченный разум

Поясняю на примере. У нас есть кривая, принимающая значения в шести точках соответственно 1, 50, 20, 1000, 10000, 3000, неважно чего, пусть для определенности долящих дятлов на квадратный километр леса, сокращенно ддкл. Если мы зададим масштаб 10 ддкл/см, то должны будем нарисовать такие, в сантиметрах, значения — 0.1, 5, 2, 100, 1000, 300 Напоминаю, что один сантиметр мы приравняли к 30 пикселям. Задача явно невыполнима, первое, второе и третье значения мы хоть как-то нарисуем, а следующие уйдут за горизонт.

Вот что у нас получится:



Пойдём другим путём. Можно задать масштаб 1000 ддкл/см. Значения для отрисовки будут такими — 0.001, 0.05, 0.02, 1, 10, 3. Результат всё равно нехорош. Пятая точка не вписалась в диапазон и осталась неизвестной. Первые три точки прекрасно видно, они в диапазон попали. Но вот только у нас во второй точке имеется ярко выраженный локальный максимум, а на графике его можно увидеть только при условии, если заранее знать, где искать. Что делать? Укрупнить масштаб? Тогда не увидим последние точки. Сделать масштаб помельче, чтоб материализовалась пятая точка? Тогда первые три сольются в прямую.



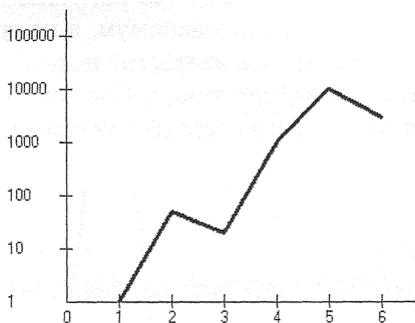
Для таких случаев умные люди придумали логарифмический масштаб. Смысл его в том, что по оси X откладываются не значения кривых в определенном масштабе, а значения их логарифмов, тоже в масштабе, само собой. Хотя в математике по умолчанию употребляются логарифмы натуральные, в реальной жизни пользуются исключительно десятичными логарифмами. Таким образом, логарифмы наших значений будут равны соответственно 0, 1.7, 1.3, 3, 4, 3.47. Отобразить это не представляет никаких затруднений.

Усложняет реализацию то, что по оси ординат при этом принято откладывать не значения логарифмов, а исходные значения данных, то есть — их значения до логарифмирования. Интервал от логарифма нуля до логарифма десяти называется первой декадой, от логарифма десяти до логарифма ста — второй декадой и так далее. Декады отделяются друг от друга горизонтальными линиями — это просто, расстояния по вертикали между ними одинаковые. Сложнее то, что крайне желательно отображать и значения всех промежуточных значений между линиями декад.

Первая декада начинается с единицы, кончается в десяти (или там начинается вторая декада, неважно, у нас тут не математический анализ, у нас реальная жизнь. То есть, надо нарисовать линии для значений 2,3,4 и та далее. Во второй декаде такими значениями будут 20,30,40... Печаль в том, что расстояния между этими линиями совсем не будут одинаковыми. Подумайте почему.

А теперь о главном! О главном применительно к логарифмическому масштабу, само собой. В логарифмическом масштабе нельзя отображать отрицательные

значения, ну нельзя и всё! Что ещё хуже, в логарифмическом масштабе нельзя отобразить даже такое безобидное число как ноль. Логарифмическую разметку может начинать, к примеру, со значения, 0.000001, это пожалуйста. А с нуля — никак.



Теперь опять о главном. В реальной жизни, на существование которой я вам неоднократно намекал, аналитические функции, или, проще говоря, функции заданные в виде хоть каких угодно формул, рисуют крайне редко, если вы не разработчик MathLab. И рисовать их относительно просто, даже если они не аналитические и даже если они не могут быть заданы в виде формул. Ну что самое страшное есть в математике? Функция Дирихле?

Справка:

Функция Дирихле — функция, принимающая значение 1, если аргумент есть рациональное число, и значение 0, если аргумент есть иррациональное число.

Можно ли нарисовать её график? — спросил я у математика. И математик заковырял в носу, а потом начал тихо качать в воздухе ботинками. А программисту — раз плюнуть. Или, выражаясь вульгарно, «как два байта переслать». Нарисуйте.

Однако, хватит о маленьких радостях жизни, пора вспомнить о сложностях. Почему относительно легко и просто рисовать математические функции? Потому что рисовать их легко по сравнению с функциями, заданными таблично, то есть, к примеру, в виде двух массивов плавающих чисел одинаковой длины. В одном массиве у нас значения аргументов, в другом массиве значения функции от них. Говорить о функции в этом случае можно только в самом общем математическом смысле, потому что никакого аналитического, в виде формулы, выражения для неё нет. Более того, эта функция не определена для всех действительных чисел (они же плавающие). Надо чётко понимать, что в промежутках между заданными значениями первого массива этой функции просто нет, мы можем

предполагать там любые величины. А откуда это всё берётся? Я спрашиваю о наших двух входных массивах.

Обычно рисуются на графике данные, поступившие из жизни, говоря иначе с како-то устройства снятия отсчётов, например уличного термометра за окном. Каждый день мы на него смотрим и снимаем данные, то есть записываем их в тетрадку в два столбика, в один дату, во второй градусы. Как известно, в молодости Иосиф Джугашвили, более известный как Сталин, работал наблюдателем в Тифлисской обсерватории. В его обязанности как раз и входило снятие отсчётов с метеорологических приборов. Несмотря на флегматический характер будущего Сталина, это унылое занятие ему быстро надоело и он решил заняться чем-нибудь другим.. Чтобы подобные случаи не повторялись, процедуру эту давно автоматизировали.

В лучшем случае, датчик выдает данные с фиксированным шагом по оси абсцисс, то есть по X. Чаще он выдаёт данные с произвольным, блуждающим шагом. Или не выдаёт вообще. Но о грустном позже.

Что касается разметки по оси абсцисс (оси X), то она всегда должна быть фиксированной, то есть с постоянным, заданным снаружи шагом, а не с тем плавающим, переменным, который имеют данные. Разумеется, по оси Y шаг тоже фиксированный, но это должно быть и так понятно.

А теперь об отображении на графике данных, заранее не известных. То есть не известных перед отображением, а поступающих постепенно. Там всё то же самое, только хуже.

— *Как часто вы чистили у него ботинки?*

— *Иногда часто, иногда редко, иногда вообще не чистил!*

© Кино «Новые приключения неуловимых»

Этот диалог в точности отражает процесс поступления данных на отображение в реальном времени — поступают данные, иногда часто, иногда редко, иногда вообще не поступают.

Ещё о реальной жизни. Как я сказал, в лучшем случае данные поступают с равномерным шагом. Но, даже в самом лучшем случае, почти наверняка от вас потребуется отображать данные с, разумеется, тоже равномерным шагом, но другим, не совпадающим с исходным. Подходить в таком случае к задаче формально не надо. Формально — имеется в виду подход «я это сделал, и какие ещё могут быть вопросы?». Вопросы будут. То, что вы сделали, это хорошо. Но начальник-то хотел другого! Он хотел не формального исполнения приказа. Он хотел, чтобы вы сделали то, что он при этом подразумевал, имел в виду, о чём он мечтал, а совсем не то, что он произнёс вслух. Когда вам дают задание, об этом явно не говорят, потому что начальник уверен, что вы адекватный человек и при этом даже программист. А когда вы приносите начальнику результат и там именно то, слово в слово, что он вам и сказал, то начальник огорчается.

Вернёмся к нашему заданию, пусть у нас будет температура за окном. Снимается она с шагом, к примеру в один час. Если данные снимаются с интервалом в один час, то, возможно, они снимаются вручную. В этом случае интервалы между снятиями будут неизбежно плавать — 55 минут, 1 час 3 минуты. Мы исходим из предположения, что у нас идеально честный сниматель отсчётов, потому что у реального в тетрадке всё будет идеально, все отсчёты ровно через 60 минут. К электронному снимателю отсчётов это замечание тоже относится. Кто его знает, когда он замерил температуру, но в файле всё будет хорошо. Но мы сейчас не о морали, мы об отображении на графике.

Окончательно сформулируем задачу, на смеси программистского языка с математическим. Есть массив X и массив Y плавающих чисел. В массиве Y у нас может быть всё что угодно, это понятно и неинтересно. В массиве X содержатся возрастающие значения, они могут быть с фиксированным приращением (шагом) или с переменным. Случай, когда данные по горизонтальной оси повторяются или даже убывают, тоже вполне возможен, но отвлекаться не будем. Наша задача — отобразить данные из массивов на графике с фиксированным шагом, который (шаг) не обязан совпадать с существующим шагом по X , даже если этот шаг фиксирован.

То, чем мы сейчас будем заниматься, по научному должно называться интерполяция, экстраполяция и аппроксимация и, возможно, ещё как-то. «Должно» — потому что все эти прекрасные методы не вполне удобны и не вполне применимы в реальных компьютерных задачах. Если вас интересует теория, то рекомендую вот эту книгу, точнее, первые её главы:

А.Половко, П.Бутусов «Интерполяция. Методы и компьютерные технологии их реализации» БХВ-Петербург, 2004

Только первые главы потому, что в них излагается чистая математика, а дальше идет применение этой теории к программам типа Matlab. Разумеется, с помощью этой теории можно *теоретически* решить все наши проблемы, но лучше там, где можно, всё сделать по-простому, без высшей математики, чем мы сейчас и займёмся. Главное, что мы узнаем из этой книги — то, что бывает интерполяция и аппроксимация. В любом случае, у нас есть, к примеру, три точки. Или, говоря иначе, два массива из трёх элементов, в одном массиве значения по оси X , в другом массиве значения по оси Y . Это общее. Дальше нам дают значение по оси X и просят сказать, сколько это будет по оси Y . А вот после этого происходит большая разница. Если точка не совпадает по оси X ни с одной из трёх заранее известных точек, то принципиальной разницы нет. А если совпадает, то принципиальная разница есть. В случае интерполяции гарантируется, что рассчитанное значение в любой из этих точек совпадёт с заранее известным. Поясняю. Это наши данные на входе, те самые три точки:

X	1.0	2.0	3.0
Y	10.0	200.0	350.0

Данные условные, потому что очень простые. Если у нас работают интерполяционные формулы, то задав новый $X=1.5$, мы получим что-то. Что именно, будет зависеть от выбранного метода интерполяции. В случае аппроксимации будет то же самое. Разница проявится при выборе $X=2.0$, например. При интерполяции мы гарантированно получим 200.0, которые были и на входе. При аппроксимации нечто в диапазоне 199.9 — 200.1 (значения условные, само собой). Если пользователь не видит исходных данных, то и так неплохо. Если пользователь точно знает, что в этой точке должно быть 200, то он огорчится. С теорией всё, теперь поглядим как теория воплощается в жизнь.

Сначала рассмотрим идеальный случай — данные поступают равномерно с шагом один час. Если на графике требуется отобразить их с шагом тоже один час, то эта задача нами уже решена. Требование сложнее — на выходе требуется отображать данные с другим шагом, но кратным исходному.

Первый случай, это, когда исходный шаг меньше заказанного. Например, исходный шаг один час, а отображать на графике надо с шагом четыре часа. Значит, надо четыре входных значения ужать в одно выходное. Как? А без затей. Разумеется, и в этом случае было бы разумным и необходимым, с математической точки зрения, применить методы интерполяции. С программистской точки зрения, это всё лишнее. Берём четыре входные значения, складываем их и делим на четыре. Это и есть искомое нами значение. Программный код приводить не буду, мы же программисты, всё-таки. В случае, когда требуемый шаг меньше исходного, тоже умничать не надо. Пусть входной шаг тот же один час, а выходной заказан пятнадцать минут. Зачем, ну скажите мне зачем? Ведь информации не прибавится! Но пользователи — они такие затейники, надо, значит надо. Стало быть, между каждыми двумя соседними точками вставляем ещё три, полученных методом линейной интерполяции. Далее повторяя сам себя в циклическом режиме — правильнее было бы интерполировать, но никто разницы не заметит. И программу писать не буду, для вас это должно быть просто.

В более сложном случае, когда исходный и заказанный шаги не кратны, приходится, наконец, прибегнуть к интерполяции. В случае, если значения некратные, но имеют то, что называется, Наибольшее Общее Кратное, то проблема может быть решена средствами, сводимыми к ранее изложенным. Но мы ищем простое, универсальное, хотя и неточное решение. Пусть наши входные данные по оси X совершенно хаотичны (но строго возрастают). В этом случае рекомендуется интерполяция по четырём точкам. Две точки берём до нашей, несуществующей, две после. Далее интерполируем по Ньютону. Что делать, если наша точка по счёту первая или вторая, она же последняя или предпоследняя? А я вас всегда предупреждал, что крайние случаи самые сложные. Хотя, конечно, это было известно задолго до того, как вы спросили моё мнение. Вылавливайте этот случай и применяйте интерполяцию по меньшему числу точек, или выбирайте другой набор точек для интерполяции. Если в этом тексте не всё понятно, то я разъясню это всё в одной из моих будущих книг «Математика для идиотов программистов».

Теперь мимоходом упомянем случай, когда данные неизвестны нам заранее вообще, то есть, нет у нас двух массивов, в которых содержатся значения по X и по Y . К примеру у нас есть класс, мы его опрашиваем и время от времени получаем от него информацию, в непредсказуемые моменты и с непредсказуемым шагом. Только пожалуйста, не надо уточняющих вопросов, всё это очень отдельная тема.

Итак — шаг по X непредсказуем. Диапазон изменения по Y тоже непредсказуем. Разумеется, вначале он задан, но что делать, если реальные данные из него выходят, не отдельными выбросами, а уверенно и надолго? Само собой, количество данных не то, чтобы неизвестно, оно неизвестно в принципе. И что делать, когда мы доползём до правой границы поля вывода? Вопросы, вопросы, вопросы...

А вообще, об этом я напишу в одной из своих следующих книг — «Системы реального времени для программистов идиотов».

Что мы видим в результате? Задача, которая казалась смешной и незатейливой до неприличия, разрослась до невероятных космических масштабов.

ГЛАВА 3

С чего начинается работа

О чём речь?

Я объясню, чего обычно хотят от свежего, молодого, начинающего программиста. То есть — чего от него обычно хочу я. Что я ему обычно поручаю, какие задачи перед ним ставлю. Как если бы я хотел выбрать правильного программиста для себя.

Я пишу «от него», но не подумайте, что это дискриминация по половому признаку. Вообще мне больше нравятся программистки, а не программисты, и не только потому, что они эстетически привлекательные, а ориентация у меня правильная. Они дисциплинированные, охотно исполняют указания и их легче заставить ходить строем, а мне это нравится.

Есть один недостаток. К сожалению, очень пугаются, когда на них орут.

Что-то длинное

Что поручают начинающему программисту? Что-то несложное, это само собой. Ещё желательно, чтобы от него, программиста, и от неё, его работы, не было очень уж большого вреда. Если юный программист накосячит, желательно чтобы его безобразия были более-менее локализованы, а не расползались по всему проекту, подобно нефтяному пятну в Мексиканском Заливе.

Случаи бывают разные. Здесь мы рассмотрим первый случай. Задача для недоразвитого программиста — обработка последовательности данных. На входе последовательность — на выходе последовательность. А что такое последовательность? Это может быть массив. Может — файл. Может — жадный объект, отдающий данные по одному экземпляру. Редко — список, хотя их очень любят авторы-теоретики.

То, что поступает на вход, обычно вылезает и на выходе. Закон природы. Я имею в виду типы данных. Если на входе массивы, то и на выходе чаще всего массив. Например, тот же массив, но отсортированный. Возможен вариант когда входной массив сворачивается в отдельные величины, например мы считаем минимум, максимум, среднее и дисперсию. Таким образом, со входом и выходом

мы, в первом приближении, разобрались. Позже уточним детали. А пока о том, что находится между входом и выходом.

Между входом и выходом находится обработка наших данных, как ни банально это звучит. Если у нас на входе массив, то и на выходе будет точно такой же массив, только другой. Вы ведь меня поняли, правда? В более сложном варианте на выходе образуется массив с другим количеством элементов, но об этом пока не будем.

Теперь передо мной стоит выбор. Можно рассмотреть разные источники и приёмники данных — как-то файл, массив, список. Это вдоль. А можно поперёк — что наша программа будет делать с данными — сортировать, фильтровать, удалять подозрительные элементы?

Нелегкий вопрос. и я принял нелёгкое решение — мы будем вдоль. То есть, рассмотрим варианты задачи в зависимости от источника и приемника данных.

В качестве возможных алгоритмов рассмотрим три их разновидности. Первое — последовательность значений, далее именуемая массив, сворачивается в скалярное значение. Под массивом здесь понимается как массив в узком смысле (**array of**), так и файл и список. Второе — массив меняет значение, сохраняя длину. Третье — массив меняет ещё и количество элементов.

Очень просто – массив. Свёртка

Сначала объявляем вот такое:

```
const
  N = 5;
type
  TNArray = array[1..N] of single;
var
  A          : TNArray;
```

Здесь всё должно быть понятно. Что мы с этим массивом сделаем? Первым делом — найдем минимум, максимум и среднее. Дисперсию считать не будем, потому что нужен второй проход по массиву (или не нужен? а как? а оно вам нужно?), а пример нам хотелось бы максимально краткий, простой и примитивный. То есть у нас на входе массив, на выходе скалярные переменные.

Заполним массив случайными значениями.

```
Randomize;

// заполним чем-то массив
for i:=1 to N do
  A[i]:=Random(100) - 50;
```

Теперь элементарное, за один проход, определение необходимых значений:

```
// свернуть массив в скалярные величины
// минимум, максимум, среднее
```

```
min:=A[1];
max:=A[1];
sum:=0;

for i:=1 to N do begin
  if A[i] < min then min:=A[i];
  if A[i] > max then max:=A[i];
  sum:=sum + A[i];
end;

av:=sum/N;
```

Вас не травмирует, что первый элемент массива мы сравниваем с ним же? Меня — нет,нисколько. И ещё, я не проверяю константу N на неравенство нулю. А кстати, может ли она быть отрицательной?

Массив. Меняются значения. И меняется длина

Задача посложнее, но всё равно очень простая — сглаживание скользящим средним. На входе массив и на выходе массив, причём тот же, что, конечно, не обязательно.

```
for i:=2 to N-1 do begin
  A[i]:= (A[i-1] + A[i] + A[i+1])/3;
end;
```

Очень просто. Да, я в курсе, что скользящее среднее в теории рассчитывается немного иначе. Но если нас интересует только результат, а не теория — то какая разница?

А теперь задача номер три, в которой число элементов массива должно измениться. Динамические массивы мы здесь принципиально не используем, поэтому для записи результатов объявим специальный массив специального типа — длиной в два раза больше. А задача наша будет такая — растянуть массив в два раза. Зачем это надо? Задача совсем не высосанная из пальца. Задача реальная. Есть кривая. Что такое кривая? Это функция, ось X и ось Y. По оси X (ординат)

идут отсчёты с фиксированным шагом. Нам нужен другой шаг, тоже фиксированный. Значения по оси X мы здесь пересчитывать не будем, потому что это тривиально. А вот пересчёт значений по другой оси (абсцисс) — это чуть менее тривиально, этим мы и займёмся.

Немного расширим объявления типов и данных:

```

type
  TNArray = array[1..N]      of single;
  T2NArray = array[1..2*N]  of single;
var
  A          : TNArray;
  B          : T2NArray;

```

Элементов массива было N . Стало $2*N$. Откуда взялись промежуточные? Промежуточные представляет собой линейную интерполяцию между соседними. В реальных условиях используется интерполяция по Ньютону, а затем всплывает волшебное слово «сплайн». Для наших учебных целей хватит линейной интерполяции, да и в практических целях её, линейной интерполяции, обычно хватает.

```

for i:=1 to N do begin
  B[i*2-1]:=A[i];
end;

for i:=1 to N-1 do begin
  B[i*2]:=(A[i] + A[i+1]) / 2;
end;

```

Можно ли было обойтись одним массивом? Да, конечно, но это всё равно потребовало бы двух проходов по массиву, с заметно утраченной наглядностью.

Похожие задачи, но для файлов.

Технология работы с файлами коротко и ясно изложена в приложениях. Напоминаю, с файлами можно работать средствами языка, средствами Windows API и через потоки — классификация моя, неправильная, но мне нравится.

животные делятся на:

- а) принадлежащих Императору,*
- б) набальзамированных,*
- в) прирученных,*
- г) молочных поросят,*
- д) сирен,*

- е) сказочных,
- ж) бродячих собак,
- з) включённых в эту классификацию,
- и) бегающих как сумасшедшие,
- к) бесчисленных,
- л) нарисованных тончайшей кистью из верблюжьей шерсти,
- м) прочих,
- н) разбивших цветочную вазу,
- о) похожих издали на мух.

© китайская энциклопедия «Небесная империя благодетельных знаний»,

Х.Л.Борхес

Здесь продемонстрируем самый простой — для меня — вариант. Работа средствами Delphi с нетипированными файлами. Задача — разделить один файл на два файла по какому-то признаку. Пусть в файле содержатся четырёхбайтовые целые числа. Напоминаю, при работе с файлом, любым, типированным или нетипированным, мы должны точно знать, данные какого типа и какого формата в нём находятся. Поделим данные по очень простому критерию — чётные числа направим в первый выходной файл, нечётные числа — во второй. У меня получилось так:

```

const
  N = 20;
var
  Fin           : File;
  Fout_1, Fout_2 : File;
  int           : integer;
  i             : integer;
begin
  Randomize;
  ChDir(ExtractFilePath(ParamStr(0)));

  AssignFile( Fin, 'file.in');

  ReWrite( Fin, 1);

  for i:=1 to N do begin
    int:=Random(10) + 1;
    BlockWrite( Fin, int, 4);
  end;

  CloseFile(Fin);

  AssignFile( Fin, 'file.in');
  ReSet( Fin, 1);

```

```

AssignFile( Fout_1, 'file.out1');
ReWrite( Fout_1, 1);

AssignFile( Fout_2, 'file.out2');
ReWrite( Fout_2, 1);

for i:=1 to N do begin
  BlockRead( Fin, int, 4);
  if (int mod 2) = 0 then begin
    BlockWrite( Fout_1, int, 4);
  end
  else begin
    BlockWrite( Fout_2, int, 4);
  end;
end;

CloseFile(Fout_1);
Closefile(Fout_2);
CloseFile(Fin);

ShowMessage(,Ok`);
end;

```

Мелкие, но желательные комментарии. Разумеется, там где указано конкретно 4 байта, вместо этого следует обращаться к соответствующей функции, вы ведь помните к какой? Интереснее другой вопрос — а сели на надо поделить числа не на две категории, а на десять? Пусть, к примеру, на простые, делящиеся на два, делящиеся на три, на пять, на семь? Подумайте. Предложите свой вариант.

Похожие задачи для списков.

Про списки смотрите в приложении. Написал всё, что знал. Здесь мы это применим. Задача такова — есть список, в котором, искусственным способом, хранятся целые (integer) значения. Посчитаем дисперсию. Потом сделаем из дисперсии среднеквадратичное отклонение. Потом пройдемся по списку и все значения, которые выходят за пределы трёх ранее посчитанных среднеквадратичных отклонений, аккуратно обстрижём. Математические термины я подробно растолкую в своей следующей книге, следите за рекламой.

Чтобы задача не выглядела, да и не была, настолько банальной, внесём элемент новизны. Пусть у нас не просто список, а список строк. В списке *строк* хранятся *слова* — разница понятна, надеюсь? Считаем среднюю длину слова,

считаем дисперсию, отклонение и слишком длинные слова обрезаем. А потому что нечего умничать.

Начнём с начала, а сначала нам надо получить список слов. Будем писать программу как реальную — то есть, предполагая, что исходный текст находится во внешнем файле, текстовом, естественно. Чтобы лишний раз не думать, выходной список слов тоже отправим в текстовый файл. А чтобы вообще и совсем не думать, для разбора на слова используем полезные функции из модуля UsStrUtils, они где-то создали, в приложениях.

Для разнообразия, начнём не с самой программы, а с объявления переменных, которые она будет использовать. Угадать правильный набор переменных заранее никогда нельзя, если здесь вообще уместно слово *угадать*. Само собой разумеется, переменные, которые я далее привожу, совсем не те, которые я написал в первом варианте программы:

```
var
  bigSL      : TStringList;
  smSL      : TStringList;
  stroka, word : string;
  num       : integer;
  sum, av, skv : single;
  maxLen    : integer;
  i, k      : integer;
```

Обратите внимание на первые две строки и, соответственно, первые две переменные. Первый список строк содержит в себе исходный текст, предположительно, разбитый на строки. Второй список в качестве одного элемента имеет одно слово, выделенное из первого списка. Слово — то, что отделено от другого слова пробелами. Знаки препинания мы презираем. Sum, av, skv — сумма, среднее и среднеквадратичное отклонение соответственно.

Далее чтение из тестового текстового файла и разбор строк на слова:

```
bigSL:=TStringList.Create;
bigSL.LoadFromFile('aaa.bbb');
smSL:=TStringList.Create;

for i:=0 to bigSL.Count-1 do begin
  stroka:=bigSL[i];
  num:=WordCount(stroka);
  for k:=1 to num do begin
    word:=GetWord(stroka,k);
    smSL.Add(word);
  end;
end;
```

Напоминаю, непонятные функции содержатся в приложении.

Класс TStringList содержит индексированное свойство Objects, предназначенное, как нетрудно догадаться, для хранения ссылок на объекты. Но, если там можно хранить объекты, то можно хранить и целые числа, в нашем случае это будет длина слова. Это самое сложное место во всей программе, остальное — ерунда.

```
for k:=0 to smSL.Count-1 do
    smSL.Objects[k]:=Pointer(Length(smSL[k]));
```

Далее вся высшая математика:

```
sum:=0;
for k:=0 to smSL.Count-1 do
    sum:=sum + Integer(smSL.Objects[k]);
av:=sum/smSL.Count;
skv:=0;
for k:=0 to smSL.Count-1 do
    skv:=skv + Sqr(av - Integer(smSL.Objects[k]));
skv:=Sqrt(skv)/smSL.Count;
```

Теперь укорачиваем слишком умные слова и сохраняем результат в файл.

```
maxLen:=Round(av + 3*skv);
for k:=0 to smSL.Count-1 do
    smSL[k]:=Copy(smSL[k], 1, maxLen);
```

```
bigSL.Clear;
stroka:='';
for k:=0 to smSL.Count-1 do
    stroka:=stroka + smSL[k] + ' ';
```

```
bigSL.Add(stroka);
bigSL.SaveToFile('aaa+.bbb');
```

```
smSL.Free;
bigSL.Free;
```

Я проверил, всё работает.

Объекты, вид сбоку

О том, что такое классы, они же объекты, я уже написал в своей книге, не самой лучшей на свете, допускаю, но неплохой. Я стараюсь ничего не повторять в новой книге из того, что уже было написано в старых. Тем более что *эта* книга ориентирована на тех, кто программировать, хотя бы формально, уже умеет. Поэтому — здесь и сейчас — речь пойдёт исключительно о применении объектов для конкретных, несложных, но жизненных ситуаций. Примеры будут упрощённые, приблизительно как пластмассовая модель танка Т-34 в масштабе 1:35. В процессе изготовления детали дорабатываются кусачками, скальпелем и наждачной бумагой, а затем модель раскрашивается в приятные глазу цвета. С нашими примерами всё точно так же.

Пишем класс — в него загружается массив. Для этого массива наш класс возвращает несколько скалярных величин — сумма, среднее, минимум, максимум, всё, что вы в состоянии придумать. И ещё что-то в виде массива той же размерности — пусть это будет массив с теми же элементами, но возведёнными в степень.

Детали реализации и путь мысли, приведший к ней, я разъяснять не буду, набросаю только общую схему. У нашего класса в секции **public** будут только конструктор и деструктор, никаких методов не будет. Вся остальная функциональность будет оформлена в виде свойств. Самое главное свойство — массив — является *динамическим* массивом. Вы ведь знаете, что такое динамические массивы, как им пользоваться, и, самое главное, как ими *не* пользоваться? Вы ведь знаете, что случится, если присвоить один обычный массив другому? Правильно, все элементы одного станут равными всем элементам другого. А что будет, если присвоить один динамический массив другому? Правильно, обе переменные будут указывать на один и тот же массив. Так почему же я применил эту технологию? Потому что это проще, проще для разработчика класса, не для того, кто будет его использовать. Если в свойствах класса присутствует массив, правильным будет это свойство дублировать — первый раз массив целиком, второй раз — индексированное свойство для каждого элемента массива. Но у нас пока что не рабочая программа, а её демо-версия.

Сначала интерфейс класса:

type

```
TArrayProcessing = class
public
    property arr : TArr read GetArr write SetArr;
    property Ave : single read GetAve;
    property Min : single read GetMin;
    property Max : single read GetMax;
    property Sum : single read GetSum;
```

```

property pow : single read fPow write fPow;
property arrPow : TArr read GetArrPow;

constructor Create;
destructor Destroy; override;
end;

```

Обратите внимание, секцию **private** я выкусил, для экономии бумаги и развития вашей сообразительности. Pow — степень, в которую мы хотим возвести каждый элемент массива, может быть дробной — какая разница, считать-то не нам.

Реализация конструктора и деструктора:

```

constructor TArrayProcessing.Create;
begin
    fArr:=nil;
    fArrPow:=nil;
    fPow:=1;
end;
{-----}
destructor TArrayProcessing.Destroy;
begin
    fArr:=nil;
    fArrPow:=nil;
end;

```

И тот и другой удивительно незатейливы. Напоминаю, что присвоение переменной динамического массива константы **nil** уничтожает все его, массива, данные и освобождает память.

Из функций для возврата значений свойств приведу возвращающую среднее. Минимум, максимум и сумма совершенно аналогичны.

```

function TArrayProcessing.GetAve : single;
var
    i          : integer;
begin
    result:=0;
    for i:=Low(fArr) to High(fArr) do begin
        result:=result + fArr[i];
    end;
    result:=result / ((High(fArr) - Low(fArr)) + 1);
end;

```

Следующая функция чуть сложнее — та, что возводит массив в степень.

```
function TArrayProcessing.GetArrPow : TArray;  
  var  
    i : integer;  
begin  
  fArrPow:=Copy(fArr);  
  for i:=Low(fArrPow) to High(fArrPow) do begin  
    fArrPow[i]:=Power(fArrPow[i], fPow);  
  end;  
  
  result:=fArrPow;  
end;
```

Сложнее она только своим первым оператором. Как я только что написал, если присвоить один динамический массив другому, результат будет отличаться от того, который мы, скорее всего, изначально ожидали.

Задача напоследок

На входе объект — источник данных, на выходе объект приёмник данных. Для разнообразия, в отличие от предыдущих примеров, которые имеют абсолютно практический характер, даже если вы в этом до сих пор сомневаетесь, этот пример будет красивый и бесполезный.

Мы напишем программу для проверки Гипотезы Гольдбаха. О чём идёт речь и кто все эти люди? В 1742-м году мелкий немецкий математик Гольдбах отправил письмо великому русскому математику Эйлеру в исконно русский город Калининград. В письме Гольдбах объявил, что каждое чётное число является суммой двух простых чисел. Изящно и красиво. А главное, всем понятно. Доказать Гольдбах это не смог. И никто не смог. Гипотеза Гольдбаха до сих пор не доказана. Скорее всего потому, что за её доказательство не предложено ни копейки американских денег.

//Уточнение

Проблема Гольдбаха является так называемой Шестой Проблемой Гильберта. Как говорится, это многое объясняет. То есть, если вы её докажете, вы станете знаменитым в математических кругах. Но денег всё равно нидадут.

//конец Уточнения

За теорией загляните в уже рекомендованную книгу — Д.Пойа «Математика и правдоподобные рассуждения».

Если вы хотя бы на чуть-чуть математик, то прекрасно понимаете, что приведение сколько угодно большого (но конечного) количества подтверждающих

примеров любую гипотезу не доказывает, но и не опровергает. Обратное неверно — для убиения гипотезы достаточно одного отрицательного примера.

//Совет

Посмотрите в интернетах, что такое теорема Гёделя о неполноте. Поразмышляйте, не подпадает ли под её действие Гипотеза Гольдбаха. Потом поразмышляйте, какое отношение вся эта фигня имеет к реальному миру, в котором мы живём,

//конец Совета

Так что мы, конечно, ничего не докажем и не опровергнем, но хотя бы развлечёмся. Например, суммой каких двух простых чисел является 1000. Я, врать не буду, могу сосчитать в уме, и быстро, но из принципа не буду. Потому что мы сейчас напишем программу, немного искусственную. В программе будут два объекта, в смысле два экземпляра двух разных классов. Один класс будет порождать простые числа, другой будет записывать куда-то результаты их разложения в сумму. Посредине будет наша программа.

Поскольку мы всё равно не докажем и не опровергнем гипотезу Гольдбаха, то давайте хотя бы проверим её до определённого предела, например для чисел до тысячи. Можно и больше, быстроедействие современных компьютеров позволяет проверить её для чисел хотя бы и до миллиона за считанные секунды. Но не будем увлекаться. Хватит и тысячи. Для целей самоограничения немедленно определяем константу

```
const
    maxNumOfPrime = 1000;
```

А теперь первый класс, тот класс, который выдаёт нам простые числа, примерно вот такой, в отношении интерфейса, включая секцию **private**, для удобства дальнейшего изложения:

```
type
    TPrimeIn = class
    public
        constructor Create;
        destructor Destroy; override;

        function GetNext : integer;
        procedure Reset;

    private
        Lin           : TList;
        next          : integer;
```

```

    procedure CalculatePrimes;
end;
```

Комментарий. Что делают конструктор и деструктор, я пока не знаю. Но пусть будут. Точнее, на счёт конструктора после размышлений я пришёл к выводу, что он создаст запас простых чисел, столько, сколько нам понадобится для проверки. Сколь бы презрительно я не относился к проблемам оптимизации, рассчитывать все простые числа до тысячи каждый раз, когда нам нужно всего одно простое число — это слишком.

Ещё один комментарий. Наш класс имеет имя TPrimeIn, второй класс будет иметь имя TSumOut. С точки зрения самого класса, если, конечно, она, точка зрения, у него имеется, это нечестно и несправедливо. Потому что первый класс выдаёт простые числа наружу, во внешний мир (о котором он ничего не знает, это принципиально, класс не может знать, как будут использованы эти числа) а второй класс, напротив, принимает результаты на вход из внешнего мира. Но, с эгоистической точки зрения нашей программы (которой ещё нет), всё совсем наоборот. Для неё простые числа поступают на вход, а разбиение на суммы уходит на выход. А поскольку все мы по природе своей немного эгоисты, именно в такой терминологии мы и будем рассуждать.

Как будет работать наша основная программа, я ещё не решил, но мне уже кажется, для нашего первого класса, который генерирует простые числа, желательна процедура, которая заставит его выдавать всю последовательность простых чисел с самого начала.

Покажем на примере. Мы запрашиваем пять простых чисел. Получаем 2,3,5,7,11. Мы запрашиваем ещё пять простых чисел. Получаем, естественно, 13,17,19,23,29. А если мы вызовем метод Reset, то снова получим 2,3,5,7,11.

Выходной класс, тот который TSumOut, будет по своим задачам прост и незатейлив, что не означает, что он будет очень уж прост по своей реализации. Вся его задача в том, чтобы принять на вход разложение чётного числа на два простых, запомнить эту комбинацию и выдать при запросе. Если нам захочется сохранить информацию для вечности в файл, то расширить функциональность класса будет несложно. А в начале он будет хранить запомненную информацию только для текущей сессии.

На вход этому классу будет подаваться исходное чётное число и два числа, представляющее его разложение на сумму простых чисел. Хотелось бы этим ограничиться и сделать наш класс самодостаточным. Увы, для этого придётся выполнять грязную работу с указателями. Чтобы сократить возню с указателями до минимума, объявим вспомогательный, на один раз, тип.

```

type
```

```

    TGoldbachRec = packed record
        number           : integer;
        compl            : integer;
```

```

    comp2                : integer;
end;
```

Как легко догадаться, первое поле записи — это исходное чётное число, а два следующих поля — сумма двух простых, на которые оно разлагается.

Класс для записи результатов очень прост:

```

type
  TSumOut = class
    public
      constructor Create;
      destructor Destroy; override;

      procedure Put(      number      : integer;
                       comp1,comp2 : integer);
      function ToStr(    ind : integer) : string;
      function ToStrAll : string;
      procedure Clear;
      procedure SaveToTextFile(      fname : string);

    private
      Lout                : TList;
end;
```

Всё ясно и понятно и очень похоже на симметричный ему класс TPrimeIn. Разница в том, что у того класса был метод Reset, а у этого есть метод Clear, назначение которых очень похоже.

Если в бассейн что-то втекает, то неизбежно должно и что-то вытекать, иначе, зачем нам такой бассейн. Если в класс что-то входит, должен быть предусмотрен и обратный процесс. Иначе это называется чёрная дыра, а зачем нам чёрная дыра? Наш класс будет выдавать наружу данные в самом примитивном, текстовом виде. Примитивном не потому, что это самое простое для реализации, а потому, что с этими данными дальше нельзя сделать ничего, кроме как на них посмотреть. Метод ToStr возвращает одну запись в текстовом формате, а метод ToStrAll выдаёт, как несложно догадаться, все записи.

Где нами будут храниться хранимые нами данные, предназначенные для хранения? В списке, само собой. Список самый обычный, TList. Напоминаю, мы программируем на Delphi и используем стандартные классы. И, напоминаю, этот список хранит просто указатели. Ещё он легко и непринужденно может хранить любые совместимые с указателями по размеру (четырёхбайтовые) данные, но в данном конкретном случае нам это ни к чему. Список мы назовем Lout, и не забудем сослаться в секции **uses** на модуль Classes.

Остаётся ответить на простой и незатейливый вопрос — и в каком же месте мы вкрутим собственно проверку гипотезы Гольдбаха? Сначала я думал, что это бу-

дет какая-то промежуточная, самая главная, процедура. Потом я махнул рукой и вместо процедуры написал метод класса TPrimeIn, с вот таким интерфейсом:

```
function EvenToPrime(      even          : integer;
                        var prime1,prime2 : integer)   : boolean;
```

Технику программирования это заметно упрощает, все методы и переменные класса здесь, под рукой. Объявим и вот такую константу, это максимальное количество простых чисел, которые помнит наш класс, программа у нас всё-таки учебная, увлекаться не будем.

```
const
    maxNumOfPrime = 1024;
```

Реализация обоих классов проста и незатейлива, поэтому никаких предварительных обсуждений и рассказов, как мы дошли до жизни такой, не будет, только готовый текст. Вот это первый:

```
constructor TPrimeIn.Create;
begin
    Lin:=TList.Create;
    CalculatePrimes;
    next:=0;
end;
{-----}
destructor TPrimeIn.Destroy;
begin
    Lin.Free;
end;
{-----}
procedure TPrimeIn.CalculatePrimes;
var
    number          : integer;
    isPrime         : boolean;
    i               : integer;
begin
    Lin.Clear;

    number:=2;
    Lin.Add(Pointer(number));

while (number < maxNumOfPrime) do begin
    number:=number + 1;
    isPrime:=true;
```

```

for i:=2 to Ceil(Sqrt(number)) do begin
    if (number mod i) = 0 then begin
        isPrime:=false;
        Break;
    end;
end;

```

```

    if isPrime
        then Lin.Add((Pointer(number)));

```

```

end;

```

```

end;

```

```

{-----}

```

```

function TPrimeIn.GetPrime(      ind : integer) : integer;

```

```

begin

```

```

    if (ind >= 1) and (ind < Lin.Count) then begin

```

```

        result:=Integer( Lin[ind-1]);

```

```

    end

```

```

    else result:=0;

```

```

end;

```

```

{-----}

```

```

function TPrimeIn.IsPrime(      number : integer) : boolean;

```

```

    var

```

```

        i                : integer;

```

```

begin

```

```

    result:=false;

```

```

    if ((number >= Integer(Lin[0])) and

```

```

        (number <= Integer(Lin[Lin.Count-1])))

```

```

    then begin

```

```

        for i:=1 to Lin.Count do begin

```

```

            if number = Integer(Lin[i-1]) then begin

```

```

                result:=true;

```

```

                Break;

```

```

            end;

```

```

        end;

```

```

    end;

```

```

end;

```

```

{-----}

```

```

procedure TPrimeIn.Reset;

```

```

begin

```

```

    next:=0;

```

```

end;

```

```

{-----}

```

```

function TPrimeIn.EvenToPrime(      even          : integer;

```

```

                                var prime1,prime2 : integer) :

```

```

boolean;

```

```

    var

```

```

        i                : integer;

```

```

begin
    result:=false;

    for i:=1 to (even div 2) do begin
        prime1:=GetPrime(i);
        prime2:=even - prime1;
        if isPrime(prime2) then begin
            result:=true;
            Break;
        end;
    end;
end;

```

А вот второй:

```

constructor TSumOut.Create;
begin
    Lout:=TList.Create;
end;
{-----}
destructor TSumOut.Destroy;
begin
    Clear;
    Lout.Free;
end;
{-----}
procedure TSumOut.Put (    number      : integer;
                          comp1,comp2 : integer);

    var
        p          : pointer;
        rec         : TGoldbachRec;
begin
    rec.number:=number;
    rec.comp1:=comp1;
    rec.comp2:=comp2;
    GetMem( p, SizeOf(TGoldbachRec));
    Move( rec, p^, SizeOf(rec));
    Lout.Add(p);
end;
{-----}
function TSumOut.ToStr(    ind : integer) : string;
    var
        rec         : TGoldbachRec;
begin
    rec:=TGoldbachRec(Lout[ind-1]^);
    result:=IntToStr( rec.number) + ' = ' +
            IntToStr( rec.comp1) + ' + ' +

```

```

        IntToStr( rec.comp2);
end;
{-----}
function TSumOut.ToStrAll : string;
var
    i          : integer;
begin
    result:='';

    for i:=1 to Lout.Count do begin
        result:=result + ToString(i) + #13#10;
    end;
end;
{-----}
procedure TSumOut.Clear;
var
    i          : integer;
begin
    for i:=Lout.Count downto 1 do begin
        FreeMem( Lout[i-1], SizeOf(TGoldBachRec));
        Lout.Delete(i-1);
    end;
end;
{-----}
procedure TSumOut.SaveToFile(    fname : string);
var
    SL          : TStringList;
begin
    SL:=TStringList.Create;
    SL.Add(toStrAll);
    SL.SaveToFile(fname);
    SL.Free;
end;

```

А теперь комментарии, как же без комментариев.

Вечная тема — обратите внимание на метод Put. Мы там выделяем память (GetMem присутствует), но не освобождаем (FreeMem отсутствует). Вам это должно быть понятно. Если непонятно, не паникуйте. Того, кому это понятно, я бы взял на работу без малейших размышлений. Того, кому это непонятно, я бы взял на работу под честное слово, что он это выучит. При этом половина сотрудников до сих пор в этом путается, так что, не переживайте.

#13#10 — Означает, что следующая порция вывода начнётся со следующей строки. Ничего утончённо хакерского тут нет.

Метод ToStringAll сам, по сути, ничего не выводит, а обращается в цикле к методу ToString. Так удобнее. На всякий случай.

А это основная программа:

```
PI:=TPrimeIn.Create;  
SO:=TSumOut.Create;  
  
for i:=2 to 800 do begin  
  even:=i*2;  
  if PI.EventToPrime( even, comp1,comp2) then begin  
    SO.Put( even, comp1, comp2);  
  end  
  else ShowMessage('no hai');  
end;  
  
SO.SaveToTextFile('gold.txt');  
  
SO.Free;  
PI.Free;
```

Если формат вывода вас не удовлетворяет эстетически — ну, сделайте что-нибудь, в конце концов!

ГЛАВА 4

Как продолжается работа. Что-то ветвистое

Преамбула

В предыдущих разделах у нас получилось что-то длинное и прямое, наподобие гадюки или ужа. А теперь займёмся чем-то круглым и ветвистым, наподобие паучка или жабы.

Данные могут быть в большом количестве, но одного типа, даже если этот тип сложный (в умеренных границах). В этом случае данные организуются в массив, одноуровневый список, или у нас есть какой-то виртуальный источник данных на входе, выдающий их по одному, и виртуальный приёмник данных на выходе, опять же принимающий их на вход тонкой стружкой. Именно это мы и наблюдали в нашем предыдущем примере, о проверке Гипотезы Гольдбаха.

Такие типы данных очень часты, именно в таком виде хранятся основные данные, которые записывают, обрабатывают, выводят на экран, печатают, и что угодно ещё. Можно сказать, что это *данные*. Но, кроме *данных*, есть нечто, что можно назвать словом *метаданные*. Это может быть чем угодно, но метаданными я (по своему личному произволу) назвал их потому, что очень часто их назначением является описание того, как понимать, как интерпретировать, как обрабатывать и куда послать дальше наши основные данные.

Непонятно? Простой пример,.

Мы записали данные. Пусть они хранятся в одномерном массиве, или в файле, что почти одно и то же. Это какие-то измерения — температура за окном в зависимости от дня недели, ваш вес в зависимости от номера недели в году, или альbedo планеты Юпитер в зависимости от Знака Зодиака. Данные мы знаем, но, кроме данных, мы знаем, что данные наши не совсем честные и точные. В случае градусника за окном и напольных весов на полу никто беспокоиться не будет. Но бывают и другие случаи.

Во всех случаях измеряющий прибор врёт. Он просто не может не врать. Отсюда возникает вопрос — как из искажённых, испорченных — то есть измеренных данных получить не то, чтобы истинные, но хотя бы сколько-то к истинным приближенные? Возвращаемся к напольным весам.

Самый простой метод — ставим на весы пудовую гирию и смотрим, что они покажут. Весы показали восемнадцать килограмм. После чего, при каждом взвешивании себя, любимого, мы, со счастливым осознанием нашего похудения, вычитаем из замеренного веса два килограмма.

Но мы-то математики. Нет, лучше физики. Мы ставим на весы две пудовые гири. Нет, лучше три. Переводя с устаревших мер, мы установили на весы общий вес в 48 кг. Теперь мы смотрим на стрелку и видим отнюдь не ожидаемые 50 кг, а, наоборот, всего 44. Это хорошо или плохо? Так что же нам показывают загадочные весы, а главное — сколько там килограммов на самом деле?

Как ни странно, те, кому приходится регулярно что-то замерять, ответ на этот вопрос знают прекрасно. Весы надо откалибровать, Нет, с самими весами делать ничего не надо. Когда мы в первый раз поставили на весы гирию в 16кг, стрелка показала на 18 кг, и мы решили всегда вычитать из показаний весов два килограмма, мы поступили абсолютно правильно с математической точки зрения. Мы выполнили калибровку по одной точке. То есть, мы сделали один замер, абсолютно будучи уверенными, чему должен быть равен ответ — ведь на заводе чугунных гирь никогда не ошибаются — и получили результат на наших стрелочных весах. И сделали вывод. Это, повторюсь, абсолютно верно, и совершенно по математически. Проблема в том, что в жизни это не работает.

Мы сделали второй замер и получили странный результат. Что это значит? Это значит, что, как минимум, нам нужна калибровка по двум точкам. Мы поставили на весы два разных груза и получили для них два результата (неожиданных). Это и есть наши две точки.

Сейчас немного математики, конкретно алгебры, если не ошибаюсь, за восьмой класс. Я проверю.

Истинный вес у нас в первом замере 16, во втором замере 48. Слово «килограммы» отбрасываем, ведь мы же математики. Наши весы врут как многочлен первой степени. Почему первой, хотя у нас две точки? А потому, что у многочлена первой степени два коэффициента. Да, это путает, ничего не поделаешь, такая условность. Если наш истинный вес мы обозначим как X , то вес измеренный (или кажущийся) мы получим по формуле

$$X_{\text{каж}} = AX + B$$

Поскольку точек у нас две, имеем систему уравнений:

$$18 = 16A + B$$

$$44 = 48A + B$$

Решаем систему уравнений. Как это делается, рассказывать не будем. Получаем ответ:

$$A = 13/16$$

$$B = 5$$

Отсюда следует, что

$$X = (X_{\text{каж}} - B) / A = (X_{\text{каж}} - 5) / (13/16)$$

Для примера, если эти весы показали ваш вес 80, то на самом деле он равен

$$\text{Честный вес} = (80 - 5) / (13/16) = 92.3$$

После чего, как говорили в ФИДО, надо пойти в угол, выпить яду и убиться об стену. А кому легко?

Что-то посложнее. Короткий раздел

Начнём с простого. Точнее, даже с простейшего. Нам надо сохранить два коэффициента (А,В) для калибровки наших весов. Сохранить надолго — следовательно, записать в файл. Как записать два числа в один файл, я объяснять не буду.

Решив задачу, мы должны её немедленно обобщить и решить уже обобщённую. Как можно обобщить нашу задачу? У нас тысяча напольных весов и для каждого два коэффициента (А,В) и всю эту тысячу надо сохранить. Банально, а главное, эту задачу мы уже решили. Пойдём расширять в другую сторону.

У нас в файле, классе, структуре (в широком смысле) хранится информация по одним весам, которые в спальне. У нас есть ещё другие весы, в ванной. И ещё на работе. И все мы их откалибровали. Как узнать, к каким весам какие коэффициенты А и В относятся? По имени файла? Можно, конечно, но это вариант скорее вспомогательный. Для напольных весов он подойдёт. А для чего-то более серьёзного — уже никак. Файл могут в любой момент переименовать. Нет, вы-то конечно не переименуете. И жена, и кошка и аквариумные лягушки тоже не переименуют, если, конечно, речь идёт именно о напольных весах. А вот если наша калибровка относится к более серьёзному и важному прибору, от которого зависит что-то существенно, то, как пишут в русских народных сказках — «тут ему и конец пришёл». Всё испортят и все сломают. Я знаю, я проверял.

Значит, имя объекта калибровки надо записать непосредственно в файл. Немного подумав, понимаешь, что туда же стоит записать и дату проведения калибровки — весы, что естественно, со временем разбалтываются и расползаются. Время от времени их надо калибровать заново, а для этих целей дата калибровки просто необходима.

Таким образом, структура информации о наших весах, и, одновременно, структура нашего файла, приобретает такой вид:

```

TBathroomScalesRec = packed record
  Name           : string[63];
  Date           : TDateTime;
  A              : single;
  B              : single;
end;

```

Немного прокомментирую. Почему не просто строка (**string**), а короткая строка (**string**[63]), наследство Паскаля. Потому, что просто строка (**string**) представляет из себя в первом приближении указатель, а во втором приближении нечто гораздо худшее, и сохранить её в файл без некоторых, не очень сложных, впрочем, манипуляций нельзя. Короткая строка, напротив, сохраняется в файл безо всяких вспомогательных танцев с бубном. А почему именно 63? А потому что один дополнительный байт в начале короткой строки выделяется для хранения её длины, так что в сумме имеем $63+1=64$. Преимущество в том, что это число образует кратное количество строк в любом просмотрщике или редакторе шестнадцатеричных значений. Это очень облегчает дальнейшие попытки разобраться, что же там такое («*мать-мать мать — привычно отозвалось эхо*» © Поручик Ржевский.)

А теперь о главном. Важное

Мы создали информационный файл с очень простым содержанием. Или, если так звучит лучше, мы создали файл с метainформацией несложной структуры. Что такое сложная структура, мы рассмотрим в следующем разделе. А сейчас поговорим о том, какого вида информация должна быть в файле безотносительно его содержания. Или, какого вида информация должна быть в файле даже в том случае, если в нём ничего полезного нет. Или вообще ничего нет. Как советует нам великий Д.Кнут — если не знаете с чего начать, рассмотрите предельный случай.

Наш предельный случай такой — что должно быть в файле с данными, если данных в нём нет вообще?

Первое — мы должны как-то узнать, что это именно тот самый файл, именно такого типа. Нет, расширение файла не подходит для этой цели, я уже говорил. В начале файла должна быть сигнатура. Что такое сигнатура? — это не имеющее никакой иной смысловой нагрузки сочетание символов, определяющее тип файла. Посмотрите любой исполняемый файл (с расширением EXE). В начале файла вы увидите волшебные буквы MZ (сокращение от имени автора формата в английском написании — Мишка Цукерман, не шутка). Вот это и есть сигнатура. Наша сигнатура будет, само собой «Bathroom Scales». Зачем пробел в конце? Чтобы было ровно шестнадцать символов. Так удобнее. Тридцать два символа тоже неплохо.

И что с этой сигнатурой делать? При открытии файла надо *попытаться* прочитать её. Обратите внимание, я не говорю *прочитать*, я говорю *попытаться прочитать*. Почему? А потому что, как говорят на родине Мишки Цукермана, *это две большие разницы*.

Рассмотрим ультра предельный случай. В файле не то что нет данных, в файле нет вообще ничего — у файла нулевая длина. Значит, перед тем как прочитать наши шестнадцать байт сигнатуры, мы должны убедиться что они, шестнадцать байт там вообще есть. А что это значит?

В чём вообще смысл чтения сигнатуры, каков результат этого чтения? Результат очень простой — или да, или нет. Или это наш файл, или это не наш файл. Если шестнадцати байт в начале файла нет вообще, неважно каких — это не наш файл. Если в файле есть шестнадцать байт или больше, то есть надежда, что файл наш. Мы читаем эти шестнадцать байт и сравниваем с нашим «Bathroom Scales». Если совпадает — наше, не совпадает — увы, или ура, дальше можно не читать. Файл не читать, я имею в виду. Извините, что я так подробно объясняю. Некоторые с первого раза не понимают.

Это очень просто, потому что ответов предусмотрено только два — да или нет, и реакция на них очень простая — читать дальше или уже всё. Теперь о сложном.

Проблема со всеми файлами, которые содержат данные и, причём, самая главная проблема — то, что их структура со временем меняется. Это относится к данным всех типов. Или это линейные данные простой структуры. Или это что-то ветвистое и кустистое, но в одном экземпляре.

Если это файл линейный и содержит, к примеру, данные по температуре за каждый день, то рано или поздно придёт клиент и скажет, что ему очень нужны данные по атмосферному давлению, он без них жить не может, и их обязательно надо добавить в файл.

Обсуждение того, что делать в этом случае, мы оставим на потом, а пока что вернёмся к нашим напольным весам.

Однажды нам захочется добавить к записи ещё одно поле. Что в нем будет содержаться? Предположим что я (вы) очень умный. И не просто умный, а ещё и математически образованный, в пределах полного курса средней школы. Нас (меня и вас) внезапно посетила мысль о том, что зависимость истинного веса от кажущегося не обязана быть линейной. И мы решили добавить третий коэффициент. У нас были А и В, мы рассчитывали истинный вес как линейный в зависимости от кажущегося. А теперь мы считаем его как квадратичную зависимость. Вы не видите в этом большого смысла? Согласен, не такая уж это серьёзная проблема, и не такой интересный вопрос, сколько я вешу на самом деле. Если вам такое расширение не нравится, то давайте запишем в файл цену весов, в рублях. Когда у нас наберётся достаточная статистика, мы сможем проанализировать, стоят ли дорогие весы своих денег, или — какая разница?

Так, или иначе, мы хотим добавить в наше описание весов два поля — коэффициент С и цену весов. Оба плавающего типа (single). Какие неожиданности

встретят нас на этом пути? Или, формулируя вопрос по другому, как мы можем это сделать?

Вариант первый, нехороший. Мы просто добавляем эти два поля в описание записи. Вот так:

```
TBathroomScalesRec = packed record
  Name           : string[63];
  Date           : TDateTime;
  A              : single;
  B              : single;
  {N}C           : single;
  {N}Price       : single;
end;
```

Букровка N в фигурных скобках — комментарий, намекающий, что это поле записи — новое, только что добавленное. Индикация эта хотя и корявая, но очень полезная, поскольку позволяет ясно видеть, где в первую очередь искать ошибки, если что-то не заработает.

А что в этом нехорошего? Если мы всё ещё делаем вид, что мы говорим о программе для домашних весов в домашней ванне для домашнего употребления, то всё хорошо. Если мы подразумеваем программу калибровки диаметра некоторого устройства, которое полетит позднее в сторону города Парижа, то вопросы возникают. Возникают, если не у нас, то у заказчиков. Плавали, знаем. Но мы пока продолжаем рассуждать в категориях пружинных весов.

Мы внесли изменения в формат наших данных. Они стали лучше, с нашей точки зрения. Но старые файлы старого формата никуда не делись. То есть, у нас есть данные — файлы старого формата, и есть данные — файлы нового формата. Два варианта, пока.

При этом никуда не делись и старые программы. Если эта программа существует и живёт только у нас, то нет проблемы. Но если ей пользуются сосед и Зинка из сорок третьей квартиры, то возникают некоторые непонятки.

Сразу скажу о печальном — вы никогда не добьётесь абсолютно правильной работы вашей программы. Ошибки неустранимы. Вы читали книги о доказательной теории правильности программ? Это ведь просто шутка, вы ведь сразу поняли? Только эту шутку некоторые приняли всерьёз. Невозможно доказать правильность реальной программы. Никогда нельзя сказать — следствие закончено, забудьте. Программа никогда не будет работать сто сорок шесть процентов правильно, не мечтайте. Надо чтобы она работала правдоподобно правильно, хотя бы процентов на девяносто девять.

Подойдём к проблеме как математики, хотя и очень незатейливые. Что касается данных, у нас есть два варианта — данные старого формата и данные нового формата. Что касается программы, то у нас (нас персонально, Зинки из сорок третьей, и прочих) вариантов столько же. Или у них программа новая, то есть,

заточенная под работу с данными нового формата, или у них программа старая, для старого формата.

Два на два — имеем четыре. Всего четыре варианта. Медленно и осмотрительно изучим их.

Новая программа и новые данные. Всё хорошо, сомнений нет.

Старая программа и старые данные. Аналогично.

В детстве я собирался жить вечно. И пока — получается! © Интеллектуальный анекдот.

Иными словами, пока у нас, программистов, всё идёт хорошо.

Теперь — новая программа и старые данные. Жить вечно уже никак не получается. Прочитав первые четыре поля из файла, которые там, безусловно, есть, наша программа попытается прочитать и ещё два поля, которых в файле, безусловно, нет. Результат предсказуем. Причём, результат этот печален независимо от конкретной структуры нашего файла.

Старая программа и новые данные. Не всё так страшно. Старая программа прочитает только свою часть, а то, что это даже не конец файла просто не догадается. Казалось бы, счастье, но только «казалось бы». Потому что мы сейчас рассмотрели идеальный, счастливый случай. Реальность неидеальна. В реальности возможны варианты. Новые данные можно ведь воткнуть и в середину записи, не обязательно в конец. Зачем? Случаи, они, как известно разные бывают. Например, если бы наши коэффициенты А и В находились в начале записи, то было бы логичным добавить туда же и коэффициент С. Результат выполнения программы предсказуем и печален. Разумеется, здесь речь идёт скорее об эстетических пристрастиях, и можно на них наплевать, тупо дописав новое поле в конец записи. Но, как известно, некрасивые самолёты не летают. Некрасивые программы не самолёты, они обычно работают, но вносить в них изменения очень неприятно.

Ещё ситуация. Ситуация чуть сложнее, тоже преодолимая. Нас не удовлетворяют эти жалкие шестьдесят три символа для записи текстового описания весов. Мы не хотим «Танька из сорок третьей квартиры», мы хотим «Танька из сорок третьей квартиры с большими выпуклыми глазами». Нам хочется хотя бы 127 символов. Мы меняем 63 на 127 — результат печально предсказуем. Разумеется, и эту неприятность можно преодолеть. Очень несложно оставить старое поле просто неиспользуемым, и добавить в конец файла новое, хорошее.

Примерно так:

```
TBathroomScalesRec = packed record
    musorName           : string[63];
    Date                : TDateTime;
    A                   : single;
    B                   : single;
    C                   : single;
```

```
Price           : single;
Name            : string[127];
end;
```

Какие-то жалкие потерянные 64 байта нас, надеюсь, совершенно не волнуют? То есть, пока всё идёт хорошо, но как-то хрупко и неустойчиво. Усугубляем ситуацию.

В один прекрасный момент просветления нас огорчит тот факт, что информация по каждому отдельным весам хранится в своём отдельном файле. Что в этом плохого? К примеру, нам захотелось перенести нашу программу на другой компьютер. Вместе с программой непременно надо перенести и хранимые ею данные. Собрать и тащить кучу файлов как-то не очень хорошо — как ни старайся, всё равно, хоть один — но потеряем. Лучше было бы собрать все их в одну кучу, то есть в один общий файл. Это общий принцип, это относится ко всем случаям, а не только к взвешиванию огурцов в банке, или я уже забыл, что мы там собственно программируем?

Теперь серьёзнее. Когда мы соберём все маленькие плохие файлы в один большой хороший файл, мы получим полную несовместимость во все стороны — старых версий программы с новыми форматами, и новых версий программы со старыми форматами. Что ещё печальнее, предположим, что мы проявили невиданную (для программиста) предусмотрительность, и с самого начала разработки ввели новый формат данных. То есть, с самого начала все данные хранятся в одном файле. Спасёт ли это нас? Не спасёт. Как только мы захотим добавить хотя бы одно поле в запись, нашу программу ждёт быстрый и бесславный конец. В плохом смысле этого слова. Почему это так, разберитесь сами.

Что делать?

Что делать? Для такого длинного вступления ответ будет неожиданно коротким. Хранить в файле номер версии файла. Хранить в программе номер версии файла, с которой она может работать. Если всё правильно организовать, то всё будет работать почти само собой, безо всяких дополнительных усилий.

Как с этим работать, рассмотрим на элементарном уровне, как простые незатейливые программисты. Я имею в виду безо всяких там классов и объектов. Так, что ни говори, понятнее..

И ещё, если не лень загляните в книгу Лу Гринзоу “Философия программирования Windows 95/NT”. Я её уже рекомендовал. Там эта проблема и методы её решения рассматриваются детально, и, возможно, даже впервые на бумаге. Написано очень давно, а ничего не изменилось. Даже я ничего лучше придумать не могу.

Сначала рассмотрим предельно простой случай, чтобы всё было как можно нагляднее. В этом случае можно обойтись и без всяких ухищрений. Но лучше сначала понять идею в целом, и лучше её понять на простом примере. У нас есть такая запись:

```
type
  TTabRecord = packed record
    A           : integer;
    B           : integer;
end;
```

И эту запись мы, извините за тавтологию, записываем в файл. А теперь формат этой записи немного меняется:

```
type
  TTabRecordPlus = packed record
    A           : integer;
    B           : integer;
    Bonus      : double;
end;
```

Что делать? Повторюсь, в начале нашего файла (любого файла!) должен храниться номер версии. Почти вначале, сразу после сигнатуры. В каком формате и размере? Я рекомендую целое (integer), четыре байта. Но одного этого мало. Проблему надо решать комплексно. Весь комплекс мероприятий выглядеть должен примерно так.

Процедуру рассмотрим медленно и с самого начала. Что мы делаем в начале? В начале — это когда у нас самый первый формат файла. Обратите внимание, в этот момент мы понятия не имеем, каким образом поменяется в будущем формат нашего файла или записи, и поменяются ли они вообще. Мы только *подозреваем*, что они могут поменяться.

Мы объявляем переменную, в которой хранится номер текущей версии формата. Именно переменную, а не константу потому, что с переменной обращаться проще. Разумеется, значение этой переменной меняться не будет. Я в курсе, что в Дельфи это делается, или должно делаться по-другому, но мы рассматриваем самый простой, примитивный и универсальный случай

```
var
  thisVersion      : integer;
```

В самом начале нашей программы, например в секции инициализации главного модуля, или модуля, в котором эта переменная объявлена, мы инициализируем её, переменную. Номер версии у нас, само собой, пока что равен единице. Сигнатуру мы уже инициализировали, но повторим на бис, для наглядности:

```
thisVersion:=1;
signature:= 'bathroom scales ';
```

Разумеется, в случае, если бы мы сразу мыслили в категориях классов и объектов, всё было бы гораздо проще. Но мы не ищем лёгких путей.

Далее. У нас есть две операции — запись в файл и чтение из файла. Исходя из моих личных предпочтений, у нас будет запись в поток и чтение из потока. На логическом уровне, это почти то же самое, что и работа с файлом.

Запись в файл происходит безо всяких ухищрений, только с комментариями на память для себя самого. Сначала мы записываем сигнатуру и номер версии, объявления переменных очевидны:

```
const
  maxSignLen = 128;
  verLen      = 4;

var
  signLen           : integer;
  signStroka       : array[1..maxSignLen] of char;
  signString       : string;
  verS              : array[1..verLen] of char;

signLen:=Min( Length(Signature), maxSignLen);
for i:=1 to signLen do
  signStroka[i]:=Signature[i];
S.Write(signStroka,signLen);

verStr:=Format( '%.4d', [thisVersion]);
for i:=1 to verLen do
  verS[i]:=verStr[i];
S.Write(verS,verLen);
```

Почему мы пишем номер версии не как целое число, а преобразованным в строку длиной четыре? Да потому, что так её, версию, проще разглядеть и понять невооружённым взглядом снаружи.

Чтение из файла заметно сложнее. Почему? Первое — мы должны проверить сигнатуру. Это, напомним, очень просто, потому что ответов всего два — или это наш файл, или не наш. Второе — мы читаем номер версии. А вот тут вариантов много.

Сначала объясню на словах, а потом в виде программного кода. Если версия нашей программы *совпадает* с версией записанной в файле, то всё хорошо, или, по крайней мере, неплохо. Если версия файла *ниже* версии программы — об этом мы поговорим. А если версия читаемого файла *выше* версии нашей программы? Почему такое может случиться, понять нетрудно — нашей старой программе подсунули новый файл.

Старую собаку новым трюкам не научишь © Народная мудрость

А вот нашу старую программу заранее надо обучить обращению с новыми файлами. И лучше в два приёма. Процедура чтения устанавливает признак того, что на входе новая версия файла, а основная программа, получив от процедуры чтения вместо данных этот печальный результат, выдаёт пользователю соответствующее печальное сообщение. Главное, что логика эта должна быть заложена в программу с самого начала. Добавлять её потом будет поздно.

Логика этой работа несложна, но всё же требует некоторого умственного напряжения. У нас есть программа. В программе записан номер версии, с которой она работает — максимальный номер версии из всех предыдущих. В нашем случае, поскольку мы только начали разработку, этот номер равен единице. Разумеется и безусловно, эта константа (переменная) должна быть объявлена не в самой программе, а в модуле, реализующем запись и чтение в файл. В идеале — в классе. У нас эта переменная называется `thisVersion`.

Дальше — очень просто. Наша программа записывает свои данные в файл, тот, который на диске, и существует реально и вечно. В этом файле, сразу после сигнатуры, будет записан номер версии — первый (символьная единица в четырёх байтах).

// Вечность

У меня есть трехдюймовые дискеты. Этим никого не удивишь. У меня есть пятидюймовые дискеты. Этим мало кого удивишь. А ещё у меня есть восьмидюймовые дискеты. На них что-то записано, и я бы очень хотел узнать, что именно, ведь это я сам что-то на них записывал, и это было что-то очень для меня важное.

А ещё, говорят, у американцев есть магнитные ленты с телеметрией данных от полетов на Луну, но прочитать их не на чем, ни одного магнитофона не осталось. Придётся верить на слово, что они туда летали.

Я вообще люблю теории заговора. Титаник подменили. И Ельцина тоже подменили. А ещё девушка в платье в горошек, это не из песни, это у американцев такая вязчивая идея, и я об этом тоже книгу написал, но её я вам не покажу.

// Конец Вечности

Напомним нашу задачу. Мы хотим, чтобы наша новая программа читала наши старые данные. Но нам этого мало. Мы хотим, чтобы наша старая программа читала из новых данных то, что она знает, и не умирала, встретив непонятные данные. Это наша задача. А наша сверхзадача в том, чтобы старая программа, встретив файл данных нового формата, вежливо предупреждала пользователя — что-то пошло не так, и что, конкретно, данные эти принадлежат новой версии, с которой она, программа, справиться не может.

То, что относится к сверхзадаче в рамках самого модуля, реализующего формат наших данных, осуществлено быть не может, это само собой разумеется. Для этого требуется привлечение той, основной программы, которая этот модуль использует. Так что, как говорится, слона будем есть по частям.

Наша основная, главная программа, читает наш файл. Мы всё ещё говорим про наш самый первый вариант нашего модуля для работы с файлом. Обратите, пожалуйста, самое пристально внимание на то, что наша основная программа не меняется вообще и никак. То есть, не меняется относительно наших целей — контроля сигнатур и версий. Почему я говорю во множественном числе? А потому, что таких внешних файлов у достаточно сложной и серьёзной программы будет не один.

Наша программа читает наш файл, проверяет сигнатуру — сигнатура совпадает, об этом больше не вспоминаем. Наша программа читает версию данных из файла и сравнивает её с версией данных из модуля, который к ней прикомпонован.

На этом месте у некоторых наступает то, что ныне модно называть *когнитивный диссонанс*. Да, это один и то же модуль. Проблема в том, что модуль, прицепленный к нашей программе, может быть прошлогодним по отношению к модулю, прицепленному к той нашей программе, которая записала наш файл, при участии той же программы. Или наоборот. Я всё понятно объяснил? Это неважно, далее мы всё рассмотрим ещё раз, подробно и по шагам.

Но пока что всё просто. Мы читаем наш файл. В файле номер версии равен единице, потому что в коде, его записывающем, он, само собой тоже равен единице. И в коде, этот файл читающем, он тоже равен единице. Всё совпадает, все счастливы. Ничего интересного. Да, разумеется, чтение сигнатуры и номера версии абсолютно симметрично их же записи.

Но прошёл месяц, наступила осень. Мы поменяли формат нашего файла. В нём появился тот самый бонус, добавленный ранее в объявление записи. Мы при этом очень аккуратно меняем номер версии (`thisVersion`) на двойку (разумеется, всё в том четырехбайтовом символьном формате). И вот тут наступает разнообразие. На нас наступает. Случаев при чтении нашего файла нашей же программой уже четыре.

Случай первый. Наша программа новая, поддерживающая версию номер два. К ней прикомпонован модуль со второй (новой) версией файла. Наша новая программа по счастливой случайности читает наш новый файл второй версии. В этом случае всё хорошо, но не надо радоваться заранее. А как наша программа проверит, что это именно новый файл? Вопрос не настолько глупый, каким кажется. Но оставим это позже, до получения окончательного программного кода.

Случай второй. Наша старая программа читает файл новой версии. Разумеется, мы всё заранее предусмотрели, и в нашей самой старой программе, с самого начала её разработки есть примерно такие строки:

```
if thisVersion < verS then begin
  ShowMessage('Усё пропало, шэф!');
end;
```

`thisVersion` — актуальная версия нашей программы, `verS` — версия, записанная в заголовок файла. Проще говоря, если версия, прочитанная из файла больше

версии, которую поддерживает наша программа, то есть та, которая объявлена в модуле, который раньше записывал старые файлы, версия которых меньше версии новых файлов, то мы выдаём сообщение. Пользователи от таких сообщений обычно пугаются и немедленно обращаются к разработчикам за разъяснениями. В сущности, это даже хорошо, потому что их становится гораздо легче уговорить перейти всем пользовательским коллективом на новую версию программы.

Задача нашей программы в этом случае прочитать из новой версии всё, что она может прочитать, то есть то, что уже существовало в её время, и ей уже понятно. Если вы это сможете гарантировать — вы очень хороший программист. В реальности никто от вас такого подвига не ждёт, достаточно, чтобы старая программа не падала на новых файлах.

Случай третий. Наша старая программа читает наш же старый файл. Опять всё хорошо, но есть особенности. Точнее, на данном этапе, когда у нас всего две версии структуры файла — старая и новая — всё хорошо. А вот позже, когда будет версии не только старая и новая и но и много промежуточных, как всегда возникнут нюансы.

Кстати, вы заметили, как расплывчато и неясно я выражаюсь — старая версия файла, структура файла, структура записи, организация данных? Это не потому, что я не слежу за своими мыслями. Это потому, что всё так расплывчато и есть на самом деле, несмотря на то, что информационные технологии — наука не то, что точная, но даже и дискретная.

Итак, наша старая программа версии один читает файл версии один. Вопросов нет. Но у нас сейчас, к примеру, актуальной является версия номер четыре. Таким образом, возможна ситуация когда программа версии четыре читает файл версии два, или когда программа версии два читает файл версии три. Эти варианты, как легко догадаться, сводятся к предыдущему — старая программа и новый файл — или к следующему — новая программа и старый файл.

Четвёртый случай — программа новая, файл старый. Основная задача — не перепутать, какую именно из старых версий файла мы читаем. Те данные, которые в той, старой версии уже были, мы читаем. Те данные, которые придумали потом, заполняем значениями по умолчанию. Напоминаю уже полузабытый пример:

```

type
  TTabRecord = packed record // первая версия
    A          : integer;
    B          : integer;
  end;

type
  TTabRecordPlus = packed record // вторая версия
    A          : integer;
    B          : integer;
    Bonus      : double;

```

```
end;
```

```
var  
TR : TtabRecordPlus;
```

Как должна вести себя наша программа? Программа, ещё раз напоминаю, у нас одна на все случаи жизни, то есть, одна, для всех возможных сочетаний поддерживаемых и читаемых версий. Только она, программа, всё время меняется.

```
// чтение и проверка сигнатуры  
// чтение версии. verS - та версия, которая записана в файле  
if verS >= 1 then begin  
    S.Read( TR.A, 4);  
    S.Read( TR.B, 4);  
end else  
if verS >= 2 then begin  
    S.Read( TR.Bonus, 8);  
end else  
if verS >=3 then begin  
    ShowMessage('Ну не шмогла я, не шмогла...');  
end;
```

Разумеется, лично я использовал бы функция `SizeOf` вместо тупой цифры 4. Ещё я задумался бы над целесообразностью применения записей в такой ситуации, но это всё мелочи. Важнее упомянутый ранее случай, когда новые поля появляются, по нашей воле, в середине записи. Ещё интереснее вариант, когда мы заранее резервируем запасные байты в нашей записи. С одной стороны, это хорошо, потому что размер файла не меняется. С другой стороны — как узнать использовано резервное поле или ещё нет? Подумайте об этом.

ГЛАВА 5

Системы контроля версий

О чём эта глава

О чём вообще эта глава, видно из её названия — о системах контроля версий, но из названия пока не видно, что это такое и зачем это нужно. На вопрос «зачем?» отвечает, по сути, вся глава, а пока ответим на вопрос «что?».

Сначала вспомним старый исторический анекдот:

Математик Чебышёв занимался, кроме прочего, развёртками фигур на плоскости. Попросили его как-то прочитать лекцию портным о методах оптимального раскроя. После первой фразы «Предположим для простоты, что человек имеет форму шара», аудитория опустела.

Отсюда пошло математическое выражение «Рассмотрим случай сферического коня в вакууме. Глупо? Нет, не глупо. Любую задачу полезно рассмотреть сначала в её вырожденном, предельно упрощённом случае, загляните ещё раз в уже рекомендованную книгу Пойа.

Для нашей задачи рассмотрим сначала случай одинокого (сферического) программиста. Одинокий программист пишет ровно одну программу, для полной сферичности эта его программа не связана никакими интерфейсами ни с какими другими программами. Программа эта состоит из одного программного модуля, физически представляющего собой один файл с исходным текстом. Сейчас так делать не принято (в Delphi), но раньше считалось даже и хорошим тоном разбивать один модуль и даже одну процедуру на несколько отдельных физических (текстовых) файлов. Наберите в Гугле «что такое include». Набрали? Я набрал и был удивлён, оказывается, эта технология вполне ещё востребована.

Шаг в сторону. Почему я иногда отвлекаюсь на предметы, которых, вроде бы, уже нет, они не применяются и почти забыты? А потому, что как нас учили на курсе диалектического материализма «Развитие идёт по спирали». Вчера было, сегодня посмеялись, завтра забыли, а послезавтра — «Здравствуйте, я ваша тётя!». Видал я этот спиральный процесс, и не раз.

Тот модуль, что пишет наш абстрактный программист, является головной программой. В переводе на дельфийский — это или консольное приложение, или вся функциональность сосредоточена в файле проекта, что нетипично. Это чудо лежит в одном единственном каталоге, что очевидно. За дельфийским про-

ектом, как хвост за кометой, тянутся понятные и непонятные файлы с расширениями CFG, DOF, DSK, DSM, RES а также ещё кое-что с тильдой спереди. Всё это мы игнорируем.

Ещё мы сделаем совершенно невероятное предположение — что наш сферический программист очень аккуратный и ответственный. Он каждый день копирует свой каталог на внешний носитель. А лучше на два. Или на три, по схеме «дед-отец-сын». А в воскресенье приходит тайком на работу и делает интегральную резервную копию за всю неделю сразу. Как говорится — *«Лучше перебдеть, чем недобдеть»*.

// Печальное и завистливое размышление

А можно я сейчас о зломном и завистливом? О тех детях, которые с детства владеют компьютером, как своими пальцами и туалетной бумагой? А как на них умиляются их родственники! И почему я на них злобный — на их детей и на их родственников?

Я очень злобный, завистливый и любознательный. Всё время, свободное от поиска углеводов и написания книг по программированию и детективов, я отдаю ползанию по Интернету. Естественно, меня больше привлекает структурированная информация — живой журнал, дневники, то-сё. В голове у меня есть фильтр — он позволяет смотреть на всю эту писанину через очки разного цвета. Можно видеть политику, можно жизненные обстоятельства, а можно видеть Компьютерный Мир. И в этом Мире бесконечно повторяется одно и то же. Девочки хранят фоточки. Потом что-то накрывается и фоточки пропадают. Обидно — для девочек. Уважаемый человек и местами даже аналитик, член правительства в прошлом, хранит на диске свою докторскую диссертацию. Диск накрывается, диссертация идёт лесом. Это, конечно, горе, но, как загадочно утверждает народная мудрость — *«горе не беда»*. Беда — когда пропадут не фоточки и не диссертация, а что-то действительно ценное и неповторимое. Мои неопубликованные книги, к примеру. Шучу.

Компьютер напоминает собаку — дети говорят *«Дай, дай!»*. А когда получают зверушку, очень удивляются, что скотину надо регулярно выгуливать, иначе она нагадит в квартире.

// конец Печального и завистливого размышления

А это я вообще к чему? Это я об одном преимуществе систем контроля версий — копировать становится проще. Если в проекте десять разработчиков, то заставить их производить резервное копирование несколько сложнее, чем заставить делать это одного администратора системы контроля версий. И сложнее не в десять раз, а скорее в десять, возведённое в квадрат, практически — невозможно.

Скучное. Предмет и терминология

Терминология. Далее будет немного занудно.

Официальная цитата из английского языка.

Microsoft Visual SourceSafe is a file-level version control system

Для целей этого раздела и этой главы:

Программный Продукт — то, что, в конце концов, получается в результате программирования, то, что уже можно продать или подарить. Или украсть. Стоит или из одного исполняемого модуля (с расширением EXE) или из нескольких. Реже состоит из одной или из нескольких динамических библиотек. Может состоять из произвольной смеси как исполняемых модулей, так и библиотек. Как модули, так и библиотеки могут быть и чужими, то есть взятыми в уже готовом виде. Далее говоря об исполняемых модулях, будем иметь в виду и библиотеки.

Сейчас для нас важно то, что, с точки зрения того, о чём мы говорим в этой главе, модули делятся на те, которые мы изготавливаем сам, и те, которые мы получаем готовыми — или из внешнего мира (например библиотеки BPL в Delphi), или от соседей — совместных разработчиков Продукта, которые почему-то не дают нам свои исходные тексты (а почему?) и передают готовые исполняемые модули и библиотеки.

Кроме этого, в Продукте неизбежно присутствует куча неисполняемых, но используемых файлов — справочные файлы, изображения, заставки и даже, страшно сказать, всякие дисклеймеры. Всё далее сказанное относится к ним в той же степени, что и к полученным нами в готовом виде исполняемым модулям и библиотекам.

Вместо «Система контроля исходных версий» я буду говорить «База исходных текстов» или просто «База», для краткости. Другими словами, база — это совокупность исходных текстов и бинарных файлов, помещённых в систему контроля исходных текстов.

Очень глубоко в технические детали я вдаваться не буду, в первую очередь потому, что системы контроля версий бывают разные, хотя выбор очень невелик. Я буду описывать конкретно систему контроля версий Microsoft VSS.

Проект — что это такое? К сожалению, этот термин захватили грязными руками и сделали его смысл в высшей степени расплывчатым и мутным. И ничего уже поделать с этим нельзя, приходится распознавать смысл на лету. Первое — есть большой Проект, с большой буквы. Это тот Программный Продукт, который мы изготавливаем. Второе — Большой Проект состоит из отдельных блоков — исполняемых модулей (библиотек). Каждый из них тоже принято называть проектом, и, извините, за интимные подробности, помещать в отдельный каталог. Это упрощает дальнейшую работу с системой контроля версий. Третье — в VSS тоже есть понятие проекта. Причём, там это скорее напоминает понятие каталога — есть Большой проект, соответствующий Главному Проекту, то есть всему нашему программному продукту, он содержит проекты (подпроекты), соответствующие отдельным модулям, а вот эти маленькие проекты содержат уже исходные тексты. Обычно к проекту базы привязан каталог на диске. Но взаимные отношения могут быть и более сложными.

Извините, что непонятно и запутанно, далее я объясню подробно и понятно. И ещё, термин «модуль» тоже вводит в заблуждение — это может быть как отдельный «физический» блок программы — исполняемый модуль (*.EXE, *.DLL), так и программный модуль (unit). Придётся разбираться по контексту.

Лирическое отступление, или а зачем это вообще надо?

Очень может быть, что вы о системах контроля версий уже всё знаете. Тогда прошу дальше не читать.

Очень может быть, что вы об этом ничего не знаете, тогда читать надо. Вроде бы это банально и очевидно, но есть повод задуматься, а главное посмотреть с другой точки зрения на то, что вам кажется естественным и очевидным. Например, возникала ли у вас необходимость в применении в ваших разработках вейвлет-анализа? Нет? Вот и у меня тоже никогда не возникала, просто потому что я не знал, что это такое. А когда узнал, от одного знакомого аспиранта, то очень задумался.

Можно усилить — можете ли вы взять интеграл, посчитать производную, или хотя бы запрограммировать пузырьковую сортировку? Предположим, что нет, и вы ничего от этого не теряете, или, по крайней мере, вы в этом уверены. У вас никогда не возникало потребности в этих знаниях, просто потому, что вы и не подозревали о существовании таких понятий.

А знаете ли вы теорему Пифагора? Очень многие теперь не знают. А уж простейшее следствие из неё — определить расстояние между двумя точками на плоскости — это что-то запредельное. Почти никто не знает. Теперь это называется высшая математика, раздел аналитическая геометрия. И если вы не знаете, то никогда об этом просто не задумаетесь. А в программировании любого графического приложения очень часто надо определять расстояние между двумя точками.

Это я всё к чему — если вы чего-то не знаете, то оно вам никогда и не понадобится. Просто потому, что вы не подозреваете о существовании этого чего-то. Оставайтесь при своём и прозябайте.

И снова о вейвлет-анализе. Лично я когда-то слышал только название и описание в двух абзацах у како-то популяризатора математики. Десять лет спустя я понял, что это очень нужная и полезная вещь, особенно в той области, в которой я работаю, только я о вейвлет-анализе ничего не знаю. Постараюсь изучить, если не поздно.

И не забивайте себе голову глупейшем тезисом «если надо, посмотрю в Интернете». Эта равнозначно надежде на умного начальника, который врежет вам сапогом куда надо и даст указания что делать, и какими методами. А вы потом посмотрите эти методы в Интернете, ага. Печаль том, что с какого-то момента ре-

шения принимаете вы. Пусть даже вы не стали начальником, с определённого момента вашей компетентности и вашей зарплаты никто не будет объяснять вам, как делать. Вам просто скажут, что надо сделать, и всё. И если вы никогда не слышали про дисперсию, теорему Пифагора или вейвлеты, это становится вашей личной проблемой. А проблемы индейцев, как хорошо известно, шерифа не волнуют.

Какие они вообще бывают?

Системы контроля версий относятся к тем предметам, которые совершенно не нужны тем, кто о них ничего не знает. О системах во множественном числе я говорю потому, что их много. Не так, чтобы очень много, предмет всё-таки штучный, но несколько. Делятся они, с одной стороны, на платные и бесплатные. Насчёт бесплатности радоваться не надо, потому что далее системы делятся на абсолютно самодостаточные, поставил и работает, и те, которым для работы требуется уже установленная СУБД (система управления базами данных). СУБД, в свою очередь, делятся на платные и бесплатные, но, в любом случае, для установки, настройки и эксплуатации требуют определенной квалификации. За всё приходится платить, так или иначе.

Выясните, что такое Borland's Together technology. Об этом даже книжки написаны. Прочитайте. Громко и вслух докажите преимущества этой штуковины. Объясните, почему лично мне это не понравилось.

Несмотря на то, что я патриот Borland, Inprise, или каким ещё смешным словом они нынче называются, я рекомендую самое простое решение — использовать систему контроля версий Source Safe. Полное название — Microsoft Visual SourceSafe, Лично у меня далее идут циферки 2005, что как бы намекает на год выпуска. Система закуплена два года назад, что как бы намекает на неспешность её обновления, что, в сущности, оправдано — с такими вещами торопиться не надо. Точнее, это вообще последняя версия продукта, Тем не менее, продукт, изготовленный девять лет назад, вполне продавался всего два года назад, что подчеркивает его непреходящую ценность.

После изучения вопроса, обнаруживаем, что последняя версия — Visual SourceSafe 2005, как и ожидалось. Теперь Микрософт предлагает взамен Team Foundation Server. Об этом я напишу в своей следующей книге.

Система от фирмы Microsoft, что у некоторых вызывает сомнения. Система платная. Система ориентирована в первую очередь на C++, с которым имеется определённая интеграция.

Само собой, для работы системы нужна сеть (если вы не используете системы в режиме только для себя, что возможно). Впрочем, в предыдущей версии (ранее MVSS 2005) была возможность использования вместо сети чего-то физически

носимого — на тот момент это были дискеты. Возможно, такой режим есть и в текущей версии, просто это уже никому не интересно.

Система состоит из россыпи исполняемых модулей, главные из которых два — Microsoft Visual SourceSafe Explorer и Microsoft Visual SourceSafe Administrator. Второй, как легко догадаться, является средством администрирования системы. Вам, скорее всего, оно не понадобится — вас много, администратор, как всегда, один. Хотя бывает интересная традиция — поручить это вредное занятие молодому программисту, его не жалко.

А теперь о главном

Теперь о главном — что это такое и зачем вообще это надо. Тут мы имеем обычную ситуацию — как объяснить пещерному человеку, зачем надо мыться. А и правда, зачем?

// Демонстрация Эрудиции

А вы знаете, каким был третий указ Петра Алексеевича, тогда ещё не Петра Великого, после возвращения из Европы? Ну, первый указ о бритье бород. Второй о ношении венгерской и немецкой одежды. А третий о запрете бань. Потому что в Европе тогда не мылись, а Пётр мечтал приобщиться к цивилизованным странам. Ну ладно, вру. Указ был не третьим, а попозже.

// конец Демонстрации Эрудиции

Более того, не то что надо объяснять пользу от систем контроля версий, ещё надо объяснять и местами даже оправдываться, почему от них один вред. Они мешают жить и работать как хочется. Примерно как Правила Дорожного Движения мешают жить и ехать куда заблагорассудится.

Поясню на примере. Когда вообще нужны системы контроля версий? Ответ *«всегда»* мне нравится, но всё же пойдём на уступку ленивым. Если вы единственный разработчик, то разрешаю вам её (систему) не использовать. Хотя, если это не совсем незатейливая программа, лучше пользоваться даже в этом случае.

Но пусть разработчиков будет трое. Каждый пишет что-то своё и то, что он пишет, делится на две категории — то, что он сам пишет и сам же использует, он и только он. И то, что он пишет сам, а используют ещё и другие, а может быть, только другие, это нормальная практика. Пусть вся система состоит из одного единственного исполняемого модуля. И, само собой, из кучи какие-то неисполняемых файлов — словари, картинки, справочные файлы. Это тоже входит в зону ответственности VSS, но об этом позже и мимоходом.

Как происходит процесс разработки в отсутствие VSS? Когда один из разработчиков (условно Вася) меняет что-то в своём модуле, который использует кто-то из двух остальных программистов (условно Петя и Анка), он выкладывает его

на общедоступный ресурс в сеть и предупреждает об этом Петьку или Анку. Если модуль используется только им и больше никем, он никого об этом не предупреждает — зачем засорять людям мозги — «во многия знания многия печали». Петька или Анка берут то, что им причитается и копируют к себе. В идеале.

Само собой, Петя иногда забывает взять то, что ему нужно взять — ведь он же очень занят — работой или наоборот, мы же все люди. Само собой, Анка иногда копирует это не в тот каталог, ей о другом пора думать. А ещё Вася иногда путается, надо ли давать модуль кому-то. Вася думает, что это его личный персональный модуль, и забывает, что он поделился им позапрошлым летом с Анкой на сеновале. Отдавать всем — лень. Или им брать лень.

Потом всё это транслируют и получают Программный Продут. Потом ищут крайнего. Не в смысле синонима «последнего», как сейчас принято говорить, а в смысле крайнего, который виноват.

И это у нас легкий случай — это у нас Программный Продукт состоит из одного исполняемого модуля. А если из двадцати? Как их кстати изготовить? Двадцать раз загрузить в Дельфи, ЦэПлюсПлюс или в какой-нибудь Haskell? Омерзительное зрелище этот ваш Haskell, между нами говоря, я сейчас этот ужас изучаю. Язык-то вроде и ничего, но разработчики реализаций и сочинители руководств сделали всё для того, чтобы загубить идею на корню. Если будете плохо себя вести, я напишу об этом книгу.

Но проблема изготовления Продукта нажатием одной кнопки — всё-таки проблема номер два. Проблема номер один — обеспечить синхронность (одинаковость, если угодно) совместно используемых программных модулей всеми, кто их совместно использует.

А теперь всё вместе и разом!

Итак, у нас программисты Вася и Петя и программистка Анка. Они разрабатывают втроём один проект. Вася самый умный, потому что его назвали в честь Василия Ивановича Чапаева, поэтому Вася пишет головной модуль, а остальные что-то вспомогательное. Пусть Петя пишет формулы, Аня рисует графику. Само собой очевидно, что Васин головной модуль обращается ко всем их (Петьки и Ани) модулям. При рисовании графики нужны формулы, а при расчете формул нужно что-то графическое — хотя бы для введения настроек, коэффициентов и чего-то ещё.

Расклад получается такой:

Васе нужны Петька и Анка

Петьке нужна Анка

Анке нужен Петька

Если у вас сложилось впечатление, что Вася плачет в углу, потому что он никому не нужен, и его никто не любит, то это неверно. Вася тут самый главный, потому что это он транслирует весь Программный Продукт целиком.

Теперь конкретно, кто что имеет? В смысле, у кого какие модули?

Вася: `vasiaMain.pas`, `vasiaSupport.pas`. Это всё от Васи, и это самое главное в проекте.

Петя: `petiaFormula1.pas`, `petiaFormula2.pas` — Петины формулы

Анка: `ankaGraf1.pas`, `ankaGraf2.pas` — что-то розовое и с бантиками.

Кроме этого, неизбежно, у каждого из них есть и какие-то свои личные модули. Доступ к этим модулям другим разработчиков совершенно необязателен. Что это за модули? В случае, если речь идёт о простом проекте, это какие-то тестовые программы, или сервисные — сами подумайте, чем они друг от друга отличаются.

А ещё бывает, что Проект включает в себя несколько исполняемых модулей (сейчас это не модно) или несколько динамических библиотек (это часто). В этом случае у Петьки, который изготавливает эту библиотеку, повторяется ситуация Васи — эти программные модули кодирует Петя, транслирует Петя и никому не даёт — а зачем? Ещё испортят что-нибудь. Разумеется, при этом он использует что-то полезное от Анки. Имеем ситуацию, когда часть программных модулей Пети должна быть выложена на всеобщее обозрение и всеобщий доступ, а часть остаться только для него, хотя они отнюдь не носят лично-служебного характера.

Теперь пойдём по шагам.

Вася один на ландшафте

Поскольку жадина Вася, самый главный в проекте, ничего никому не даёт, а только всё время хочет что-то от других, с него и начнём. Даже упростим задачу — пусть у них там самое начало проекта, Петьку и Анку ещё даже не приняли на работу, Вася справляется один и весь проект состоит из двух Васиных модулей. Так что нам предоставляется прекрасный случай изучить сферического Васю в вакууме на фоне VSS (напоминаю, сокращение для Microsoft Visual SourceSafe).

Уже здесь возникает Зловещий Человек под названием Администратор, хотя, возможно, на этом этапе Вася сам себе Администратор. Первое, что делает Вася-Сам-Себе-Администратор, это добавляет *пользователя* «Вася» (можно по-русски). Добавляет не совсем точно, в том смысле, что пользователь «Вася» там пока первый и единственный. Далее создается *проект* *Vasia*. Все права на проект *Vasia* выдаются пользователю «Вася».

У Васи, напоминая, есть два программных модуля — `vasiaMain.pas` и `vasiaSupport.pas`. То, что расширение именно `PAS`, непринципиально, напоминая, изначально система была и есть часть `Microsoft Visual Studio`, так что ориентирована система совсем не на Паскаль.

Сначала Вася внутри своего проекта `Vasia` создает ещё один проект. Зачем? Затем, что у Васи не одна программа, а много. Ну или будет их много. И сваливать все программы в один каталог как-то некрасиво. А пока назовем новый проект `ProgOne`. А теперь торжественный момент — Вася помещает (командой `Add`) в проект `ProgOne` два своих модуля — `vasiaMain.pas` и `vasiaSupport.pas`. Это очень просто. Всё!

Но, как всегда, есть нюансы. Сначала первый. При попытке отредактировать файл, к примеру, `vasiaMain.pas`, Вася, к некоторому своему удивлению, обнаружит, что это невозможно. Причина — файл теперь `read-only`. Это поведение Системы по умолчанию, и это хорошо, это правильно. Потому что это дисциплинирует.

И, чтобы два раза не вставать, у Системы есть ещё одна опция, по умолчанию, наоборот, отключенная. Если же её включить, то, при положении файла в Систему, он, файл, автоматически удаляется с Васиного жёсткого диска. То есть, с этого момента Васин файл существует в одном единственном экземпляре в Системе, в её базе. Таким образом автоматически и навсегда решается проблема уникальности файла — файл существует ровно в одном экземпляре. Это дисциплинирует, причём очень сильно. Но, врать не буду, за все годы использования `VSS` я не видел ни одного клиента, работающего таким образом. Слабоват народец, боится чего-то. Так что Васе лучше оставить эту опцию выключенной, как она и есть по умолчанию.

Теперь надо произвести очень несложную операцию — задать для созданного проекта (`ProgOne`) рабочий каталог. Рабочий каталог в терминологии `VSS` Это `Working Folder`. Или считайте наоборот, их `WF` в моей терминологии `ПК`. Для чего нужен рабочий каталог, вы узнаете чуть позже.

Добавление исходного текста в базу происходит один раз в жизни — в жизни текста, разумеется. А вот редактирование текста происходит неоднократно. Алгоритм редактирования соответствует алгоритму укладывания слона в холодильник — открыть холодильник, положить слона, закрыть холодильник. Этот алгоритм ясен и прозрачен, а наш я опишу чуть подробнее.

Взять исходный текст из базы. Это делается командой `Check Out`. Файл из базы помещается в рабочий каталог, если рабочий каталог не был задан, об этом нам невежливо напоминают. При этом с файла автоматически снимается атрибут `read-only`.

Дальше мы файл редактируем. Это самое простое и понятное. После редактирования мы возвращаем его в базу командой `Check In`. Когда мы что-то в базу кладём, перед нами всплывает окно для ввода комментария. Настоятельно рекомендую этим окном не пренебрегать и комментарий из последних сил ввести. По традиционной схеме, с префиксами:

A: что-то добавили

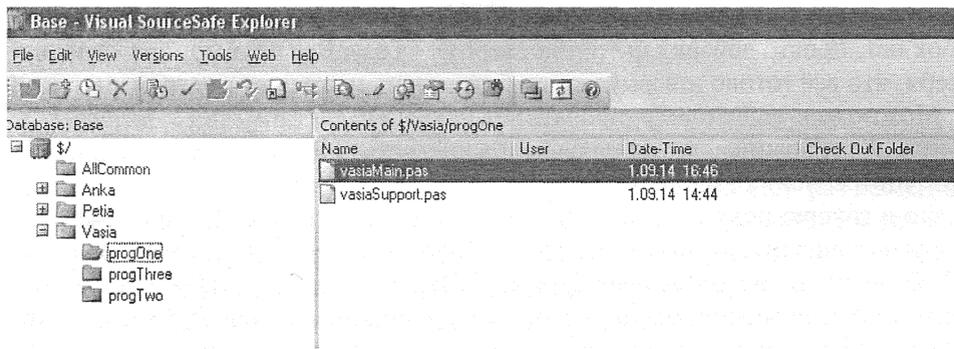
Ch: что-то изменили

D: что-то убрали

F: поправили ошибку.

После возвращения файла в базу, он немедленно приобретёт атрибут «только для чтения» (у нас на жёстком диске).

А теперь картинка — просто показать, как это выглядит. Чтобы два раза не вставать, в базу здесь уже пристроены Петька и Анка и кое-что ещё.



Слева у нас полный список проектов базы верхнего уровня. Васин проект развёрнут, и в нём виден полный список подпроектов первого уровня. Выбран progOne, и справа показаны содержащиеся в нём два файла исходных текстов. Файлы находятся, если можно так выразиться, в пассивном состоянии — тихо лежат в базе, никто из них не находится в состоянии Check Out.

Вспомогательное счастье

Здесь я ещё раз попытаюсь ответить на незаданный вопрос — а зачем всё это надо? В чём счастье? Сначала от трудном и непонятном счастье. Потом о счастье маленьком и очевидном.

Внедряя систему контроля версий приходится сверху, то есть принудительно. Трудящиеся визжат и сопротивляются. Основная психологическая проблема в том, что приходится выложить всё своё творчество на всеобщее обозрение. Некоторые, особенно очень продвинутые и креативные, опасаются, что украдут их гениальные разработки. Но таких мало, и они всегда в соседней комнате. Потому что, если такие завелись в нашей комнате, их немедленно надо травить, как тараканов, средством «Коба». Вы не застали это средство? Ах да, тараканы уже вымерли — из-за электромагнитного излучения компьютеров.

На самом деле, вопросы авторских прав мало кого волнуют. Большинству просто неприятно, что их тексты в любой момент может увидеть кто угодно. Даже если установлен самый тактичный и деликатный режим разделения доступа, начальник всегда видит всё, потому что ему предоставлены полные права доступа ко всем проектам. Мучает не только то, что можно посмотреть текст, огорчает тот факт, что можно посмотреть и историю изменений — не получится втихую исправить никем ранее не замеченную ошибку и подбросить код обратно, как ни в чём не бывало.

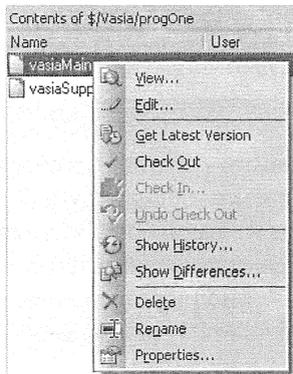
Страшная проблема не только в начальнике — страшная проблема в себе самом. Надо четко управлять собой — обнаружил проблему, взял код, исправил проблему, вернул код. Или, пусть даже в более мирном случае, когда никаких ошибок нет, просто добавил функцию, вернул в базу. Вот этот момент — решил для себя, что всё готово и вернул код в базу, очень напрягает программистский мозг. Можно ведь ещё помедитировать над исходным текстом, ещё немного, ещё чуть-чуть. А тут возникает печальная необходимость принять решение — или код готов к употреблению, или как.

И это я говорю пока о простой человеческой психологии. Дальше хуже, потому что на горизонте витают монстры Технологических Необходимостей, которым VSS очень, ну очень мешает. Пример? Первое, с чего мы начали переход к системе контроля версий, было устранение дубликатов файлов. Дубликат — это когда один файл, с одним и тем же именем, но, возможно, и, скорее всего, с разным содержанием, живёт на разных компьютерах и в разных каталогах и возможно принадлежит разным программистам. И эти три возможности сочетаются во всех комбинациях.

И мы взяли бесплатную для частных лиц программу Duplicate File Finder (DFF) (считайте это бесплатной рекламой бесплатной для частных лиц программы DFF) и её применили. И стали мы все версии файлов, более одного экземпляра, уничтожать. И тут случилась, как говорят эти ваши девочки в этих ваших интернетах — «Грустька, Пичалька и Депреська».

Начался вселенский крик и вопль о невозможности обойтись одной версией файла с одним именем, что я отстал от жизни, и что я всего боюсь — это да, это безусловно, боюсь. Вопрос урегулировали, но осадок остался.

А теперь по мелочи, или, что полезного мы получаем ещё? Если мы нажмем правой кнопкой мыши на имени файла в базе, то появится вот такое локальное меню:



Сейчас нас интересуют пункты «Show History» и «Show Differences».

Напоминаю, файл `vasiaMain.pas` был сначала помещён в базу (Add). Затем его отредактировали (Check In/Check Out). И пусть потом его ещё раз изменили (опять Check In/Check Out). Это для наглядности. Повторяя другими словами для очень умных, у нас в базе теперь лежат три версии исходного текста — исходная, после первого редактирования и после второго редактирования.

Для полной наглядности далее привожу все три версии.

Первая:

```
program Kuku;
begin
end.
```

Вторая:

```
program Kuku;
var
  R           : single;
  S           : single;
  stroka     : string;
begin
  R:=10;
  S:=3.1415 * R*R;
end.
```

Третья:

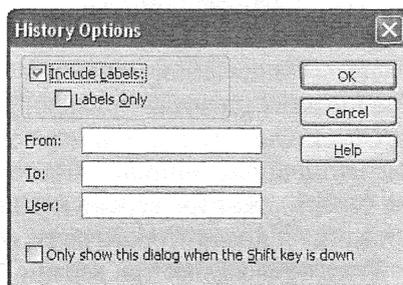
```
program Kuku;
var
  R           : single;
  V           : single;
begin
```

```
R:=10;
V:=(4/3) * 3.1415 * R*R*R;
Writeln( 'V = ', V:7:2);
```

end.

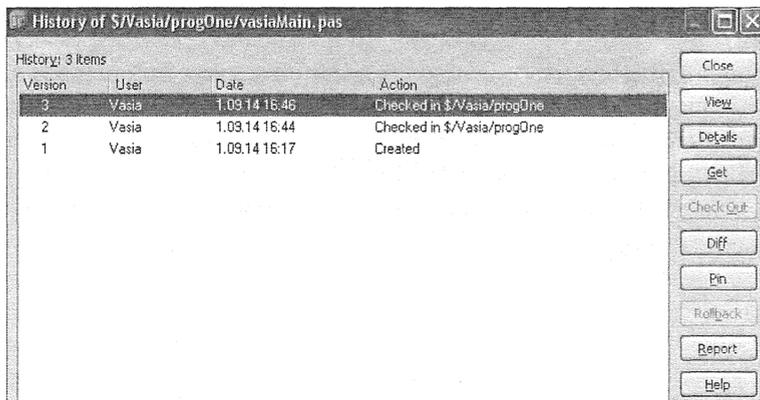
Идея понятна. Первый вариант — пустая заготовка, болванка программы. Второй вариант работает — считаем площадь круга. Третий — концепция изменилась, считаем объём шара с тем же радиусом. Во втором варианте мы объявили строковую переменную, наверное, для целей вывода результата, а в третьем отказались от неё, и вывели результат непосредственно.

Теперь выбираем пункт меню «Show History». Видим следующее:



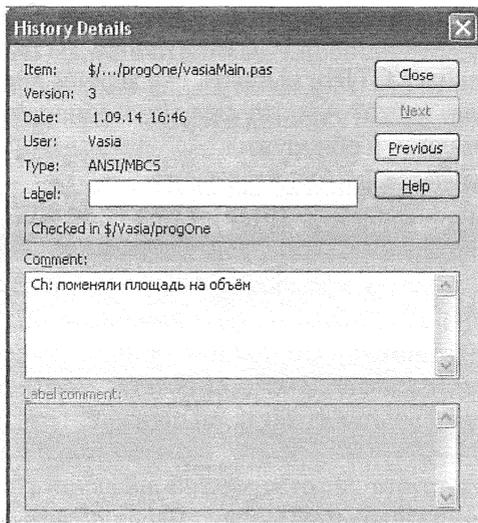
Это нам не интересно, хотя в этом окне можно указать, от какой версии до какой версии файла нас интересует эта самая история. Мне это кажется совершенно избыточным, хотя, поставив птичку в левом нижнем углу, можно от этого окна навсегда избавиться. А пока нажимаем «OK».

Теперь имеем вот это:



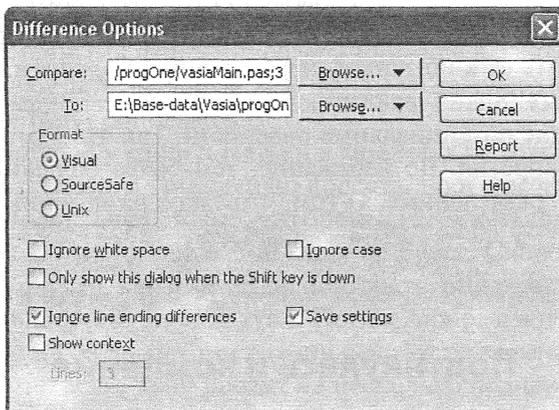
Это полезнее. Мы можем видеть, сколько раз и когда именно менялся этот файл, это здесь самое важное. Вопрос у начальника всегда один — когда последний раз прикасались к файлу? Обратите внимание, что читать надо снизу вверх.

Обратите внимание на кнопки справа. Теперь выбираем интересующую нас версию и нажимаем кнопку «Details». Получаем следующее:



В поле «Comment» мы видим комментарий, который мы написали при возвращении файла в базу после внесения в него изменений. Вы ведь написали комментарий? Я же вас просил написать?

Ещё интереснее возможность просмотреть различия между версиями файла. Делается это или выбором пункта «Show Differences» из локального меню, или нажатием кнопки «Diff» из предыдущего окна «History». Сначала плохой способ.



К сожалению, или тут не всё так просто, или Microsoft опять накосячил. Точнее, всё относительно просто, но очень неудобно. Или, как обычно, то ли лыжи не едут, то ли я непонятливый.

Из верхнего поля редактирования «Compare» копируем его содержимое и заменяем этим содержимым то, что находится в нижнем поле «To:». После этого своими собственными руками заменяем номер версии 3 на тот номер, с которым мы хотим сравнить, обычно на единицу меньше, то есть 2. Нажимаем Ok.

Того же результата можно добиться, вернувшись на окно раньше — History. Далее, удерживая клавишу CTRL, выбрать две версии для сравнения и нажать кнопку <Diff>. Правда, легко? А я это случайно обнаружил десять лет спустя. В смысле, мне рассказали, и я обнаружил.

И вот, наконец, результат:

```

Differences between vasiaMain.pas version 3 and vasiaMain.pas version 2

$Vasia\progOne\vasiaMain.pas; 3
1 program Kuku;
2   var
3     R           : single;
4     V           : single;
5 begin
6   R:=10;
7   V:=(4/3) * 3.1415 * R*R*R;
8   Writeln('V = ', V:7:2);
9 end.
10

$Vasia\progOne\vasiaMain.pas; 2
1 program Kuku;
2   var
3     R           : single;
4     S           : single;
5     stroka      : string;
6 begin
7   R:=10;
8   S:=3.1415 * R*R;
9 end.
10
  
```

А что мы здесь видим? Мы видим различия программных текстов разных версий.

К величайшему сожалению, книга наша чёрно-белая, цветная книга денег стоит. Монитор цветной, это бесплатно. Говорят раньше, очень давно, были люди, которые могли, глядя в чёрно-белый телевизор увидеть исходную цветную картинку. И вы попробуйте, наша картинка была исходно цветной.

В VSS предусмотрена возможность возвращения к предыдущей версии файла, называется Rollback, закопана она довольно глубоко, видимо, во избежание со-блазна. Пользоваться ей не рекомендую. Сама по себе она совершенно не страшная, если применять её к обычным, неразделённым, файлам. При использовании на разделённых (Share) файлах возникают нюансы, перерастающие в проблемы, вызывающие потенциальные опасности. Так что я это безобразие сразу строго-настрога запретил. Во избежание, тем более большинство сколько-то важных файлов — разделённые. О том, что такое разделённые файлы, мы ещё поговорим.

Подтянулась и компания

А теперь о главном. На работу приняли Петьку и Анку и они изготовили свои программные модули. Для них, модулей Петьки и Анки, создали соответствующие проекты в базе — Petia и Anka. Они, Петька и Анка, добавили туда (Add)

свои модули. Кроме того, они добавили туда кучу каких-то личных исходных текстов, которые никого, кроме них не интересуют.

Теперь о грустном. Есть очень нехорошие люди, которые кладут в базу всё. Это ведь нетрудно и есть оправдание — а вдруг пригодится? В случае Delphi, они кладут туда файлы CFG, DSK, что хотя бы имеет некоторый смысл. Но им этого мало, они отправляют туда же и файлы типа DSM, что по меньшей мере странно. На этом процесс не останавливается и в копилку складываются DCU и EXE (или DLL). Поубивал бы.

Запомните простое правило. Пока вы начинающий программист, кладите в базу только файлы типов PAS, DFM и DPR. Когда вы поумнеете, можете класть туда всё, что вам угодно. Так что наши умненькие Петя и Аня положили туда только по два файла исходных текстов с расширением *.pas.

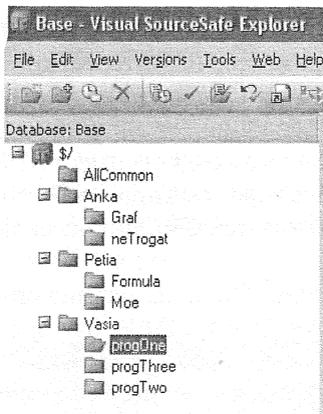
Теперь наш Василий Иванович должен изготовить Программный Продукт. Об изготовлении Продукта одной кнопкой мы пока ещё и не заикаемся. Изготовить хоть как-то. Объяснять подробно, как он его изготавливает на этом этапе, не будем — всё равно это неправильно, и так делать не надо. Как это надо делать правильно я объясню позже.

Но, к примеру, пусть пока наш Вася просто скопирует из базы нужные ему для Проекта файлы к себе, в рабочий каталог. От Петьки он копирует файл petiaForm1.pas из проекта VSS (каталога, по сути) Petia. От Анки копирует файл ankaGraf1.pas из каталога базы Anka. Я перечисляю это так долго, уныло и занудно для того, чтобы показать нелёгкую задачу стоящую перед Васей. Нужно помнить два каталога в базе. Нужно помнить, какие файлы из них взять. Нужно помнить *когда* их взять — иными словами, изменилась ли версия файла, и не пора ли его обновить.

Последнюю проблему — проблему последней версии — упрощает пункт локального меню с говорящим именем «Get Latest Version». Только вызывать локальное меню с этим пунктом лучше не на имени файла, а на всём каталоге сразу. По этой команде будут осмотрены все файлы, находящиеся в каталоге базы и сравнены с одноимёнными файлами, содержащимися в рабочем каталоге. Если файлы из базы имеют дату, большую, чем файлы из рабочего каталога, то файлы в рабочем каталоге будут обновлены.

Ухудшим ситуацию — у Анки и Петьки в каталогах не только те файлы, которые необходимо включить в трансляцию проекта, но и ещё что-то личное. Вася должен знать и помнить очень точно, что брать из этих каталогов. Конечно, если он возьмёт что-то ненужное, большой беды не будет. Скорее всего. Не будет. Беды. Большой. А случаи, как я не устаю повторять, они разные бывают.

Теперь ситуация становится ещё хуже. Петя и Аня решают улучшить организацию своих данных. Говоря по-русски, они решают не валить всю фигну в одну помойку и аккуратно делят свой проект на несколько подпроектов (или подпомоек). Примерно вот так:



Теперь и без того утомлённый на голову Вася обязан помнить из каких именно подпроектов своих коллег ему надо брать исходные тексты.

А теперь у нас *внезапно* не Вася и два подцепляемых разработчика, а целый их, разработчиков, десяток. И Вася *внезапно* должен помнить, у кого что где лежит, и что оттуда надо брать, а что лишнее. И это не для двух разработчиков, а для всех десяти. Нет, так жить нельзя.

И мы встречаем волшебную команду Share. По-русски — разделить, но это согласно словарю. Разделить — это когда пирог режут на части и каждому достаётся по куску. А тут всё наоборот — раздели файл и он есть у каждого, и у хозяина файла он тоже есть. Я пошёл в угол умиляться.

Как это на практике? Создаём в базе VSS проект. Называем его нейтрально AllCommon — то есть типа всё общее. Проект будет одного уровня с проектами Vasia, Petia и Anka. На этом месте пора вмешаться администратору базы исходных текстов.

Здесь у нас будет подраздел в разделе. «Под» — потому что очень короткий, и на настоящий раздел не тянет, а всё-таки «раздел» — потому что это важно. Есть такое понятие, относящееся не только к VSS — разграничение прав доступа. В нашем случае это чёткое указание кто из пользователей может что-то делать с каким-то проектом, и что именно он может делать.

В самом начале у нас есть одинокий Василий Иванович и вопрос снимается. Василий Иванович по совместительству является также и Администратором. Администратор автоматически имеет полный и абсолютный доступ ко всему, даже в том случае, если он вообще не подключен к базе, как клиент. Запомните это на всякий случай.

Далее — подключаются Петька и Анка. Вася не хочет, чтобы они имели доступ к его данным, но очень хочет иметь возможность брать исходные тексты от П. и А. — ведь ему же надо транслировать весь проект в целом, с участием как своих текстов, так и не своих. Петька с Анкой кричат и сопротивляются — не очень умный, по их мнению, Вася, им всё испортит. Но не надо переживать за них раньше времени, не всё так однозначно.

Для каждого проекта есть четыре уровня доступа — по нарастающей. Точнее даже пять уровней. Пятый означает, что доступа нет вообще. Важно понять, что уровень доступа не является абсолютной величиной — кому нужен проект, к которому ни у кого доступа нет? Естественно, что уровень доступа — один из пяти — назначается конкретно для Васи. Упростим картину. Из пяти уровней оставим три. Полный доступ — делаю с проектом что хочу, это всегда уровень хозяина проекта. Противоположный уровень — нельзя ничего. Это уровень доступа к тем подпроектам проекта, которые никого интересовать, кроме их хозяина не должны — пресловутые тестовые программы, например. Это два уровня абсолютно понятны и очевидны. Ещё один уровень промежуточный — читать (и копировать, само собой) можно, а больше ничего.

И не забывайте об администраторе, у которого есть права на всё. Помните старый стих? Нет, конечно, не помните. Потому что и не знали.

*Только злобный Супервизор
Лезет нагло и свободно
В директории чужие
Скоро! Скоро все повиснут!*
© Народное, скорее всего

Теперь применяем эти три уровня к нашим трём пользователям (клиентам) VSS (базы). Слова в скобках здесь исключительно для того, чтобы напомнить что слова перед скобками и слова в скобках обозначают одно и то же.

Для тех каталогов (подпроектов), которые нужны только Пете и Ане, мы выставляем доступ для владельцев — полный, а для всех других пользователей — никакой. Здесь всё понятно. Вася, естественно, запрещает доступ к своим проектам всем, кроме себя, он же главный.

Теперь о проекте Petia. У Пети к нему, само собой, полный доступ. У Анки доступа к Петюне нет никакого — а действительно, зачем ей доступ? У Васи доступ только на чтение — ему ведь нужны Петины исходные тексты для трансляции. Аналогичная ситуация с проектом Anka. Для Ани — всё, для Пети — ничего, для Васи — только для чтения.

В результате все счастливы — Вася имеет всё от всех, Петя и Аня уверены, что шаловливые Васиные ручонки не залезут в их исходные тексты с нелепыми исправлениями.

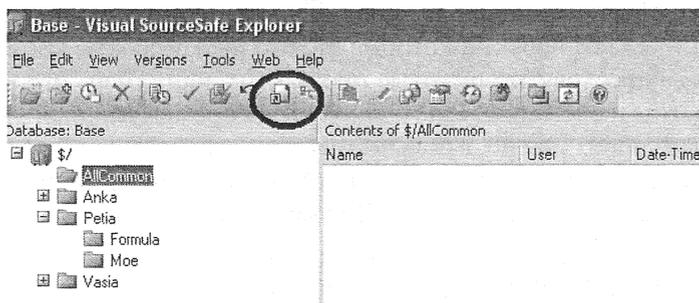
Как всё сделать хорошо

А теперь о главном. Мы хотим, чтобы Вася мог транслировать свой продукт без необходимости долго и мучительно выбирать руками необходимые ему фай-

лы из каталогов, в которых они лежат у его соратников. Здесь надо действовать очень аккуратно, потому что дело действительно серьёзное.

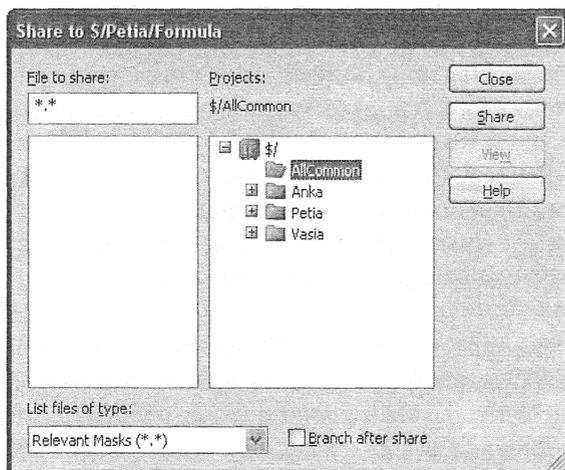
Сначала техническая и алгоритмическая сторона вопроса — как это делается. Затем — что это значит и для чего.

Петья идёт в базу, но не в свой проект Petia, а в только что созданный, но пока ещё пустой проект AllCommon:

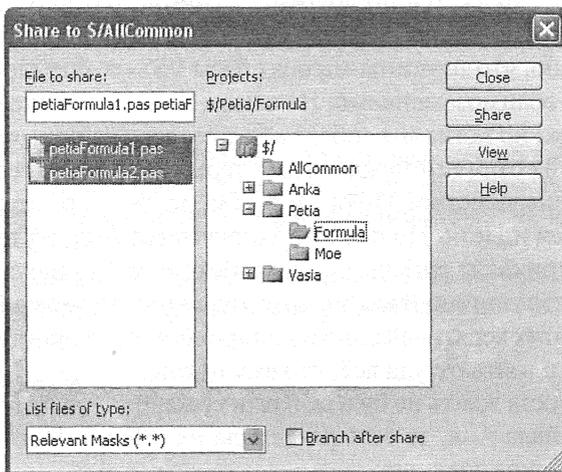


Чтобы картинка выглядела достаточно читаемой и масштаб был покрупнее, я урезал её по горизонтали. Если посмотреть на исходный образ, далеко справа маячит надпись «No Working Folder». Этим мы займёмся попозже, для наглядности от имени Васи. Напоминаю, сейчас мы в базу вошли как Петя.

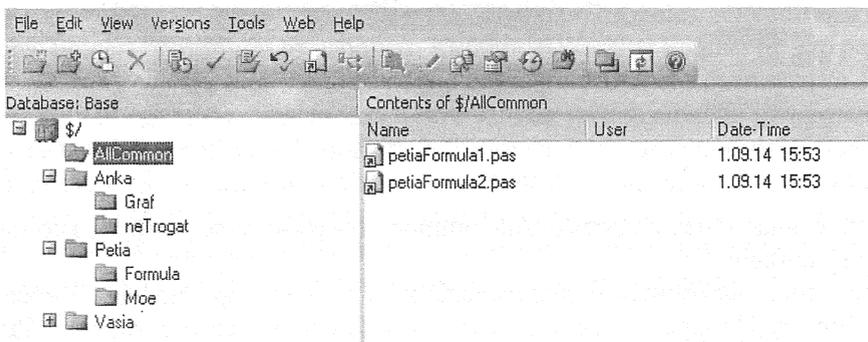
Кружком обведена целебная кнопочка «Share», которую мы и нажмём.



В появившемся окне выбираем проект (каталог) Petia\Formula\ и отмечаем оба находящихся там файла.



Нажимаем справа кнопку с неожиданным именем «Share». Затем нажимаем тоже справа кнопку «Close». После этого форма исчезает, а основная форма приобретает вот такой вид:



Обратите внимание, что значок слева от имени файла выглядит по-другому, не так, как он выглядел раньше у всех файлов. Загогулина слева внизу намекает, что файл является разделённым или расшаренным (жарг.) или как вам больше нравится это назвать.

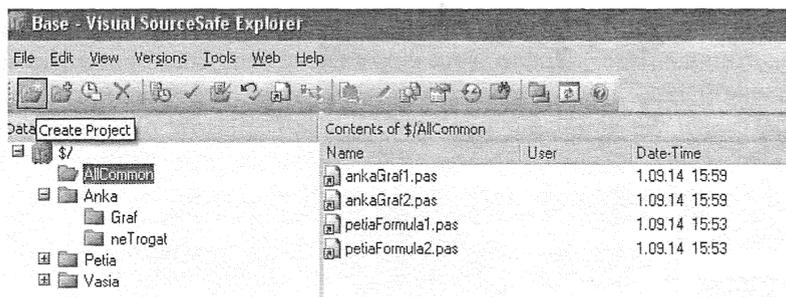
Что получили в результате? Обсудим подробнее. Тот файл (petiaFormula1.pas), который находился в каталоге у Пети, там и остался. Это естественно. Но, одновременно с этим, он появился в каталоге AllCommon. С этого момента, все изменения, которые Петя произведёт в своём файле (в каталоге Petia\Formula), автоматически будут отзеркалены и в том экземпляре (копии) файла, которая находится в каталоге AllCommon. А если наоборот? А наоборот никак, потому что каталог AllCommon доступен всем (абсолютно всем) клиентам базы только на

чтение. Если это вдруг окажется по другому, администратор базы должен быть публично наказан.

Ещё раз напоминаю, что термины «проект базы VSS» и «каталог» используются здесь совершенно взаимозаменяемо. На самом деле, с технической точки зрения, это не совсем так, но когда мы доберёмся до освоения применения теневого каталога, ни малейших отличий между этими терминами не останется.

И в чём здесь преимущество? Пока ни в чём, до тех пор, пока в разработке участвуют только Вася и Петя. Но потом подключается Аня, на тех же условиях, что и Петя. Потом приходят ещё десять разработчиков, но нам это уже безразлично. Точнее, это безразлично Васе, потому что с этого момента Вася не должен бегать по каталогам этих десяти-двадцати кодировщиков, а просто может зайти в каталог AllCommon, и взять оттуда всё, что ему нужно.

Десяти разработчиков ждать не будем, пусть ту же процедуру по отношению к своим файлам проделала Аня, то есть разделила их (Share). Поглядев на проект AllCommon, увидим такое зрелище:



То есть, сейчас у нас в проекте AllCommon содержится всё, что только надо Василию Ивановичу.

И здесь пора вспомнить о неопределённом до сих пор Рабочем Каталоге (Working Folder). Неопределённом только для AllCommon, для всех (трёх) остальных личных проектов они, рабочие каталоги, уже определены.

Не боясь быть занудным, повторю ещё раз. Для каждого проекта в базе указывается (ставится в соответствие) Рабочий Каталог. Это тот каталог, который используется при работе с этим проектом, используется на чтение, в большей степени, но не только. Когда вы командуете Check Out, то есть извлекаете файл для редактирования, то он помещается в Рабочий Каталог. Когда вы командуете Get Latest Version, то все файлы из проекта помещаются в Рабочий Каталог (но не помечаются как извлеченные из базы). Небольшая разница заключается в том, что в случае Get Latest Version файлы не ваши. То есть, они не становятся доступными для редактирования на вашем компьютере, и в протоколах базы не остаётся никаких записей об этом вашем действии.

Для проекта Anka\Graf рабочим каталогом будет скорее всего D:\Anka\Graf или что-то очень похожее. Можно, само собой, выбрать и другой каталог. Но это

уже будет создание проблем на свой собственный организм. Аналогично, для проекта Vasia\ProgOne рабочим каталогом будет D:\vasia\ProgOne/

//Разъяснение

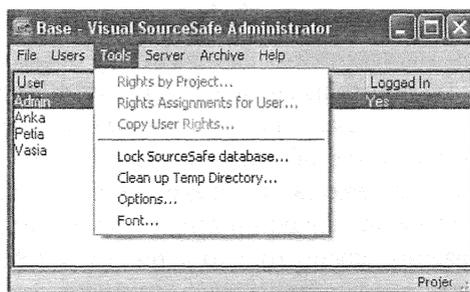
Имя логического диска D:\ употребляется здесь повсеместно только потому, что я воспитан в строгой дисциплине — размещать на диске C:\ сам Windows и все программы, на диске D:\ свои данные, а на диске E:\ игры, фильмы, музыку и порнуху. В основном, конечно, порнуху.

И запомните — никогда не скачивайте с рабочего компьютера игры, фильмы, музыку и порнуху. В основном, конечно, порнуху. Скачивайте всё это дома. И вы будете жить долго и счастливо.

//конец Разъяснения

Про Одну Кнопку

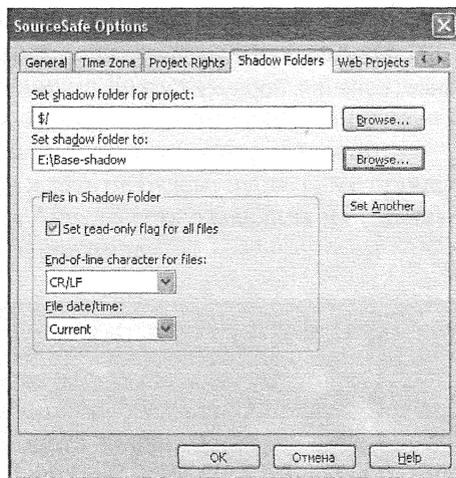
Страшное слово — теневой каталог. Первая ассоциация — *«Призрак, летящий на крыльях ночи»*. Ничего, привыкнете. Хорошая сторона теневого каталога (Shadow Folder), в отличие от рабочего Каталога (Working Folder) та, что рабочих каталогов много, а теневого каталог ровно один. Поэтому его очень легко задать. Единственная маленькая трудность — его задают не с помощью программы работы с базой (Microsoft Visual SourceSafe Explorer), а вызывая программу администрирования базы — (Microsoft Visual SourceSafe Administrator/ Это естественно, ведь в базу каждый пользователь заходит от себя лично, а теневого каталог — предмет глобальный, один на всех Итак, вызываем программу администрирования, вводим пароль (логин не нужен, он всегда один — Admin) и вызываем вот такое меню:



Далее вызываем пункт меню «Options,,,», в появившейся форме выбираем закладку «Shadow Folders». В верхней строчке нажимаем кнопку «Browse», выбираем там нашу базу, и только её, в этом месте думать категорически не надо. Во

второй строке вводим руками или выбираем каталог, в котором будет размещён наш Теневой Каталог. Я предпочитаю создать его заранее и на этом этапе просто выбрать.

Потом нажимаем кнопку «ОК». Если база реальная и большая, то реакция будет ощутимо медленной. Для нашей игрушечной базы всё будет исполнено мгновенно.



Что мы получим в результате? Самый простой путь узнать это — пойти и посмотреть, не изнутри VSS конечно, а из какой-нибудь внешней независимой программы. Слабые духом используют Explorer, правильные пацаны — FAR, те, кто не определились — какой-нибудь Total Commander. В выбранном каталоге мы увидим структуру каталогов, в точности соответствующую структуре проектов базы. Внутри каталогов лежат файлы, имена которых совпадают с именами файлов, которые мы видим обращаясь к базе. Ещё лежит какой-то псевдоневидимый служебный файл, но мы его игнорируем.

А в чём преимущество, для чего мы всё это делали?

Прежде чем ответить, нам потребуется прояснить несколько технических деталей. Как хранятся в базе текстовые файлы? Текстовые файлы это не те файлы, которые содержат внутри себя текст. Файлы с расширениями DOC, DJVU и PDF текстовыми ни разу не являются. Текстовые файлы — это то, что называется Plain Textю Они содержат только коды символов и пару-тройку управляющих кодов, причем коды символов традиционные, однобайтовые, а не модный UNICODE. Но это неважно, важно то, что к текстовым файлам относятся исходные тексты программ.

А как хранятся эти файлы в базе? Мы можем посмотреть различия между последней, 225-й версией файла и версией, к примеру, 17-й. Отсюда закрадывается подозрение, что в базе хранятся все версии файлов, от первой до двести двадцать пятой. Это действительно так, но хранятся они очень специальным образом.

Первая версия хранится целиком. А для всех последующих в хитром виде содержатся только внесённые изменения. По нынешним временам выглядит это довольно глупо, проще и надёжнее было бы хранить просто все версии файлов, как они есть. Но напоминаю, даже последняя версия VSS вышла в мохнатом 2006-м году. А про первую даже и вспомнить трудно. Тогда идея хранить кучу почти одинаковых файлов на маленьком жёстком диске (40 мегабайт) казалась нелепой и кощунственной. Но, что выросло, то и выросло.

Что ещё хуже, имена файлов в базе очень мало, а точнее, совсем никак не напоминают исходные имена тех же файлов до помещения в базу. Отыскать файл в базе могут только очень-очень умные. Наверное, в этом есть какой-то смысл, мне, к сожалению, недоступный. В дополнение, файлы хранятся в каком-то нелепом формате. Нет, я абсолютно уверен, что даже я, в конце концов, с этим форматом разобрался бы. Но зачем мне это нужно?

Теневой каталог оставляет от всего богатства, хранящегося в базе, со всем его неисчислимым количеством версий файлов только последние их версии. Структура его каталогов в точности совпадает со структурой проектов базы, но теперь мы видим простые и понятные имена файлов — именно те, которые мы туда и клали, и именно в том формате, в котором мы их туда и клали. То, что мы можем пойти и посмотреть, что же у нас в базе лежит, не пользуясь средствами базы — это, разумеется, преимущество, но небольшое. Это преимущество только для тех, кто в надёжности VSS как-то немного сомневается и хочет пощупать файлы руками.

Главное, что теперь мы можем весь наш проект изготовить то, что называется *одной кнопкой*, точнее нажатием этой самоq одной кнопки. Чтобы наглядно показать, как это делается, надо немного усложнить ситуацию. Сейчас наш виртуальный Василий Иванович живёт в полной гармонии с самим собою — то есть, пользуется только своими файлами. Сделаем ситуацию более сложной, такой, как мы и обещали сделать её с самого начала — В. в своей программе вызывает процедуры (функции), реализованные в модулях от А. и П. Пример будет игрушечным и простым, само собой.

Кроме того, чтобы не отвлекаться позже, отработаем несложную манипуляцию с базой. А, возможно, и познакомимся с новым понятием — консольное приложение. В каком смысле — новом?

- *«Попутная песня». Исполняется впервые.*
- *Да какое впервые, её сто пятьдесят лет поют!*
- *Мною — впервые.*

© Винокур

То есть, это не я впервые пишу консольное приложение, а кто-то из вас, возможно, впервые о нём услышит. И увидит. Впервые.

На каком языке сейчас написана Васина программа? Можно сказать, что на псевдокоде, а можно сказать проще — на (Турбо) Паскале. На Турбо Паскале, к

сожалению, уже не программируют, поэтому меняем всё, ничего по существу не меняя. Переносим программу из Турбо Паскаля под Delphi, но делаем её консольным приложением.

Быстро отвечаю на несколько несложных незаданных вопросов.

Что такое консольное приложение? Это такая программа как под DOS, но под Windows. Не поняли? Сейчас поймёте.

Откуда они берутся, консольные приложения? Можно написать его полностью руками, например в «Блокноте», и сохранить с расширением DPR, а можно в меню выбрать File\New\ Other и на первой же закладке выбрать Console Application. Получите заготовку маленького проекта.

Зачем нужны консольные приложения? Не знаю. Я за последние пятнадцать лет написал ровно три штуки — они молча чистили каталоги от ненужных файлов. Можно было бы сделать их обычными приложениями Delphi, но мне так было проще — после привычной работы с Турбо Паскалем. А вообще, по-моему, единственное назначение консольных приложений — гордиться, что вы умеете их писать.

Итак, очередная (пока ещё промежуточная) версия Васиной программы теперь выглядит вот так.

```

program progOne;
{$APPTYPE CONSOLE}

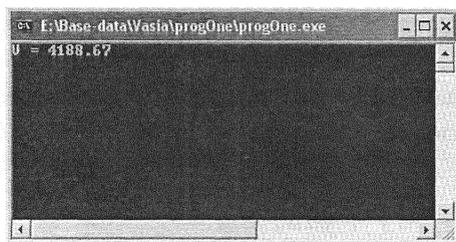
uses
  ankaGraf1, petiaFormula1;

var
  R          : single;
  V          : single;
begin
  R:=10;
  V:=(4/3) * 3.1415 * R*R*R;
  Writeln( 'V = ', V:7:2);
  Readln;
end.

```

Вторым номером идет волшебная строка, которая. Собственно, и делает приложение консольным. В остальном, это один в один паскалевская программа. Чем дольше живёшь, тем больше видишь, что ничего, в сущности, не меняется.

А что будет, если мы программу в её нынешнем состоянии запустим на выполнение? Получим мы вот такое жутковатое зрелище:

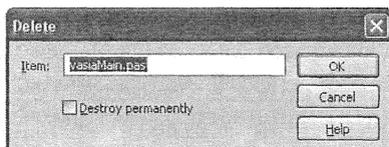


Так вот ты какое, консольное приложение! Очень похоже на древний заплесневелый MS DOS.

Возвращаемся к Петьке и Анке, они уже подключены, но ещё не реализованы. То есть, что-то там формально под этими именами в базе лежит, но немножечко пустое. Надо срочно реализовать.

Теперь несложная операция с базой. Мы помещаем (добавляем/Add) наш, только что созданный, файл в базу. Операцию Check Out мы при этом не делаем. Почему? Потому, что наш старый файл имел расширение PAS (был паскалевской программой). Наш новый файл имеет расширение DPR (является проектом Delphi). Друг друга они заменяют, но не замещают. Мы не меняем одну версию файла на другую версию того же файла, мы меняем один файл на другой. Поэтому, положив новый файл в базу, мы удаляем старый. Можно и не удалять, но когда вы засорите проекты ненужными старыми файлами, в которых невозможно разобраться, вы об этом пожалеете.

При попытке удалить файл, мы получим вот такой запрос:



Смысл вопроса в том, хотим ли мы наш файл Delete (с возможностью восстановления) или Destroy (*бритвой по горлу и в колодец* — © Джентльмены удачи). На этой мелочи можно было бы и не останавливаться, но дело в том, что две эти операции требуют разных прав доступа к VSS. Сейчас это не важно, потом разберётесь, вместе с администратором базы.

А модули Петьки и Анки реализованы следующим образом:

Первый модуль от Анки:

```
unit ankaGraf1;
interface
{-----}
}
procedure ShowV(      R,V : single);
```

```

{-----}
}
implementation
  uses
    SysUtils;
{-----}
}
procedure ShowV(      R,V : single);
begin
  Writeln( 'Volume of sphere with radius ' + FloatToStr(R) +
    ' is equal ' + FloatToStr(V));
end;
{-----}
}
end.

```

Мы чуть-чуть изменили интерфейс, теперь процедура выводит и исходный радиус тоже. Второй модуль содержит что-то настолько же полезное, но его мы здесь приводить не будем. Просто поверьте, что он есть. Он должен быть для усложнения нашей задачи изготовления Программного Продукта одной кнопкой.

Модуль от Петра:

```

unit petiaFormulal;
interface
{-----}
function VolumeOfSphere(      R : single) : single;
function AreaOfCircle(      R : single) : single;
{-----}
implementation
{-----}
function VolumeOfSphere(      R : single) : single;
begin
  result:=(4/3) * pi * R*R*R;
end;
{-----}
function AreaOfCircle(      R : single) : single;
begin
  result:=pi * R*R;
end;
{-----}
end.

```

Здесь у нас целых две функции, для объёма шара и площади круга. Пользователи Дельфи могут бросить мимолётный взгляд на идентификатор PI и задуматься, почему он нигде не объявлен и за что к нему такое уважение (он объявлен в модуле System, который подключается автоматически).

Но в целом всё понятно, в смысле понятно, какие функции эти модули реализуют.

Соответственно, корректируется и головная программа от Васи. Теперь Вася не занимается чёрной работой по формулированию формул и выводом вывода — теперь Вася только даёт команды Петьке и Анке.

```

program progOne;
  {$APPTYPE CONSOLE}

  uses
    ankaGraf1, petiaFormulal;

  var
    R           : single;
    V           : single;

begin
  R:=10;
  V:=VolumeOfSphere (R);
  ShowV (R,V);
  Readln;

end.

```

Наша технология изготовления Программного Продукта будет оформлена как файл типа BAT, называемый также «пакетный файл». Можно сделать и по другому, это всегда можно, но книга имеет конечный объём. К сожалению, только написав эти слова, я задумался — а вы знаете, что такое файл типа BAT? Сейчас это называется скриптовый язык. Как говорили в моём детстве — *«Кулер тоже вертолёт, только маленький ещё»*. То есть, файл типа BAT действительно является скриптовым языком, только очень примитивным. Настолько примитивным, что вы в нём легко интуитивно разберётесь. Напомним структуру теневого каталога нашей базы.

```

Base-Shadow
  AllCommon
  Anka
  Bat
  Petia
  Vasia

```

Проект (каталог) Bat только что появился — в нём будет храниться то, что нужно нам для изготовления Программного Продукта одной кнопкой. Проекты Anka и Petia, строго говоря, нас вообще не интересуют — их содержимое дублировано в проекте AllCommon, точнее, не всё содержимое, а то, что представляет

интерес для внешнего мира. Проект, в смысле Delphi, который мы транслируем, находится в проекте, в смысле VSS, с именем Vasia.

Здесь я не буду по шагам излагать ход мысли, который привёл меня к этому результату, а сразу покажу результат и его прокомментирую. Итак, в нашем не очень сложном случае в проекте Vat будут два файла, с вот такими именами и вот таким содержанием:

```
makeProgOne.bat
cls
copy/y dcc32_progOne.cfg c:\delphi7\bin\dcc32.cfg
rmdir /s /q e:\progOne
mkdir e:\progOne
cd e:\Base-Shadow\Vasia\progOne
call progOne e:\Base-Shadow\
rem the end -----
dcc32_progOne.cfg
-aWinTypes=Windows;WinProcs=Windows;
-uc:\delphi7\lib
-ee:\progOne
-luVcl;
-$O-
-$D-
-$L-
-$J+
```

Первый файл выглядит более пристойно со смысловой и эстетической точки зрения, что и немудрено — это его надо запустить, чтобы изготовить Программный Продукт. Файлы с расширением VAT запускаются так же легко и непринуждённо, как и файлы с расширением EXE. Полужирным шрифтом выделено то, что официально называется «внутренняя или внешняя команда». Далее построчный разбор шедевра.

```
cls
```

Всего-навсего очистка экрана, если мы запускаем VAT-файл, к примеру, из-под FAR. Необходимо на этапе отладки, потом несущественно.

```
copy/y dcc32_progOne.cfg c:\delphi7\bin\dcc32.cfg
```

Как легко догадаться, имеет какое-то отношение к нашему второму файлу, вместе с ним обсудим и эту команду.

```
rmdir /s /q e:\progOne
mkdir e:\progOne
```

Парные команды. Первая удаляет со всем содержимым и без ненужных вопросов некоторый каталог, вторая его же создаёт, пустой естественно. В ключах разберётесь. Это формально. По существу, это тот каталог, в который мы поместим наш Программный Продукт. На момент запуска, предположительно, там находится предыдущая версия ПП.

```
cd e:\Base-Shadow\Vasia\progOne
```

Переходим в каталог (проект), где лежит файл проекта (DPR) самой главной Васиной программы.

```
call progOne e:\Base-Shadow\
```

Запускаем ещё один, специфически Васин, BAT-файл. О нём позже, главное, что в его задачу входит изготовление Васиной программы. Почему нельзя было обойтись одним пакетным файлом? Потому что Программный Продукт часто состоит не из одного исполняемого файла и/или динамической библиотеки. Чаще их много. В этом случае пакетный файл верхнего уровня будет по-прежнему один, хотя и более сложной структуры, а пакетных файлов второго уровня прибавится в количестве.

```
rem the end -----
```

А это, как легко догадаться, просто комментарий.

Теперь займёмся вторым файлом, он имеет расширение CFG и совсем не исполняемый. Расширение это очень любят все разработчики, поэтому, что скрывается внутри, определить заранее невозможно. В нашем случае это конфигурационный файл проекта Delphi. Файлов таких на компьютере может одновременно существовать очень много, поэтому есть определённые правила, какой именно из них применять в данном конкретном случае. В тонкостях вы разберётесь сами, а пока о главном. При создании проекта в среде Delphi конфигурационный файл обычно автоматически создаётся в каталоге проекта, он же, естественно, и применяется.

Мы конфигурационный файл в базу не кладём, поэтому, как можно догадаться, его там и нет. Вместо этого мы помещаем наш конфигурационный файл в Самый Главный Каталог Delphi, заменяя нашим файлом тот файл, который используется по умолчанию. Обратите внимание, старый файл мы не запоминаем в начале и не восстанавливаем в конце. А зачем? Этот файл нужен только тогда, когда нет конфигурационного файла в каталоге проекта, а он есть всегда, когда мы работаем в IDE Delphi — вы ведь знаете, что это такое? Теперь построчный разбор.

```
-aWinTypes=Windows;WinProcs=Windows;
-uc:\delphi7\lib
```

Так положено. Если Delphi установлен в другом каталоге, поправьте вторую строку. Обратите внимание на отвратительный синтаксис uc:\delphi7. Поубивал бы.

```
-ee:\progOne
```

А это мы назначаем, в какой именно каталог отправятся наши оттранслированные файлы. Если вы не догадались сразу — e:\progOne.

```
-luVcl;
```

Какие мы используем пакеты (packages). В данном случае только Vcl.bpl.

Далее опции компилятора, с этим вы легко разберётесь.

Третий, и последний командный (пакетный) файл. Если вы не забыли, мы переходим в главный Васин каталог и вызываем его личный пакетный файл, вот такого содержания:

```
dcc32.exe /U%1allCommon progOne.dpr
```

Выглядит ужасно, а содержание очень незатейливо. Мы вызываем программу (dcc32.exe) и передаём ей на вход два параметра, разделённых пробелами. Сама программа представляет собой компилятор командной строки Delphi, то есть мы транслируем программу не из среды Delphi, а почти молча. Маленькая особенность — для всех версий Delphi я не проверял, но будет лучше, если в момент исполнения этого файла среда Delphi *НЕ* будет загружена. Второй параметр очевиден — это та программа, которую мы должны транслировать. Первый параметр чуть сложнее.

%1 — это то, что называется *формальный* параметр. Фактический параметр, в нашем случае — e:\Base-Shadow\, то есть местонахождение нашего теневого каталога. К нему приклеивается продолжение — allCommon. В сочетании с волшебным ключом /U получаем адрес, по которому надо искать необходимые для трансляции модули. Обдумайте.

То, что называется «Бинарные файлы»

Что такое «бинарный файл» в понимании VSS? Как это ни странно звучит, бинарный файл, это тот, который не текстовый. Какой файл текстовый, а какой

бинарный, решает лично VSS, достаточно успешно, до сих пор ошибок не наблюдалось. Два напоминания.

Первое. Файлы Microsoft Word, с точки зрения Microsoft VSS, текстовыми не являются, а являются, наоборот, бинарными.

Второе. Типичные бинарные файлы — картинки, маски, глоссарии и тому подобное — это с одной стороны. С другой стороны — это исполняемые файлы (EXE) и динамические библиотеки (DLL), которые, по каким-то интимным причинам мы имеем только в таком, ненадкусанном виде, а не как исходные тексты. Причины, в основном, имеют локальный характер. Коллеги по работе из соседней комнаты берегут своё, что называется know-how, и боятся, что вы украдёте у них интеллектуальную собственность. Но есть и промежуточные варианты — по слухам, файлы BPL, из комплектации Delphi, являются внутри себя теми же динамическими библиотеками. А если вы честно купили Delphi, за честно заработанные деньги, то имеете полное право раздавать эти библиотеки всем, кому... Вы поняли, короче.

В чём особенность хранения бинарных файлов в VSS? Главная и единственная особенность хранения бинарных файлов в VSS заключается в том, что файлы эти хранятся, не как текстовые, то есть не в инкрементном режиме. Инкрементный режим, это когда файл в самый первый раз сохраняется в базе полностью, а далее хранятся только его изменения.

Я искренне восхищаюсь программистами Фирмы, добившимися безупречной работы этого режима. И им же искренне сочувствую, когда они обнаружили, лет примерно через десять, что это никому не нужно. Объём накопителей достиг уровня, когда его, накопитель, занять практически невозможно, то есть невозможно занять этими самыми текстовыми файлами. Проще было не мучаться и хранить целиком. Хотя, с другой стороны, программисты были обеспечены интеллектуальной высокооплачиваемой работой.

С бинарными файлами пошли изначально другим путём. Бинарные файлы в первый раз сохраняются полностью, как и текстовые, а далее, в отличие от текстовых, тоже хранятся полностью. А это, не то чтобы критично, но уже немного заметно. Через два—три года будет незаметно и это, но вы когда-нибудь замечали, как быстро увеличиваются размеры исполняемых файлов?

Если у вас в проекте десять левых (не ваших) исполняемых файлов и десять левых (не ваших) динамических библиотек, и каждое из них имеет размер в десять мегабайт, и в течение года это всё обновили тридцать раз, но это уже немного заметно. Мелочь, но неприятно.

Какая из этого мораль?

Главное не в том, чтобы реже обновлять бинарные файлы в базе. Главное, чтобы класть их туда как можно меньше. Картинки, маски и глоссарии вполне допустимы. Но, если в базу приходится класть исполняемые файлы и библиотеки, причем не честно купленные один раз, и потому до следующей покупки не обновляемые, а разработки из соседней комнаты, то с соседней комнатой надо сделать что-то очень доброе и воспитательное. К примеру, залить напалмом...

Когда-то, ещё до того, как японцы взорвали Фукусиму, когда я ещё верил что японцы умнее людей из джунглей с выдвинутой далеко вперёд нижней челюстью, по другому — питекантропы, меня восхитила японская концепция «пять почему». Смысл этой гениальной, безо всяких оговорок, и настолько же примитивной, опять-таки, безо всяких оговорок, концепции, в том, что если мы хотим докопаться до причин чего-то, нельзя останавливаться на первом «Почему?».

— Почему программа рухнула? — а потому, что выдала Range Check Error.

— Почему программа выдала Range Check Error? (безобиднейшая ошибка, честно говоря, очень легко диагностируется и исправляется) — а потому, что индекс вышел за пределы массива.

— А почему индекс вышел за пределы массива? — а потому, что мы поместили в массив тридцать первый элемент, не проверив допустимость этого действия — в нашем массиве всего тридцать элементов.

— А почему мы поступили так глупо с тридцать первым элементом? — а потому, что это данные по дням месяца, мы решили, что все месяцы одинаковые и объявили массивы для них по тридцать дней.

— А почему мы так решили? — а мы не только так решили, у нас и в году ровно 365 дней, хотя случаи бывают разные.

— Так почему же мы так решили, блин? — а потому что мы сэкономили память.

К теме экономии мы ещё вернёмся.

А в остальном, работа с бинарными файлами ничем не отличается от работы с файлами текстовыми.

В качестве самостоятельного упражнения организуйте средствами нашего однокнопочного транслирующего файла добавление в Программный Продукт файлов, необходимых для автономной работы — например, Vcl70.bpl.

Как НЕ НАДО работать с VSS

Сначала о личном. Вот не далее, чем сегодня, в день, когда я это пишу, понадобилось внести относительно безболезненные изменения в два файла. В первом файле (модуле) была процедура (не метод), которая раньше там, в модуле, использовалась. Второй модуль, наоборот, содержит общеприменяемые и общепотребительные процедуры. Со временем в первом модуле эта процедура быть нужной перестала, а во втором модуле, наоборот, её присутствие стало желательным — эту процедуру стали использовать и другие разработчики, которым приходилось ссылаться на первый модуль.

Можно было оставить и так. Но я по жизни эстет и вообще очень аккуратный, поэтому я захотел удалить процедуру из первого модуля, который лично мой и перенести во второй модуль, который общий. Никаких проблем — взять первый и второй файлы из базы, выкусить процедуру из первого, воткнуть во второй и

вернуть в базу. Словосочетания Check In и Check Out я уже не пишу, полагаясь на догадливость читателя. Так вот тут-то проблема и возникла.

Когда я полез в базу, я обнаружил, что со вторым файлом всё хорошо. А вот первый файл внезапно оказался в состоянии Check Out. Причём взял я его из базы не на прошлой неделе, а в прошлом году. Какие-то смутные воспоминания об этом у меня в памяти остались, но не осталось воспоминаний о том, с какой целью я взял его из базы. Из базы просто так не берут. Точнее, это я просто так не беру, другие берут и просто так, но об этом в следующем разделе.

То, что я не помню, зачем я этот модуль взял, это, как говорится в русской народной поговорке «горе — не беда». Беда в том, что я не помню, внёс ли я в него какие-то изменения. Тут беда раздваивается. Несложно сравнить две версии файла из базы, сравнить файл из базы и файл вне базы немного труднее. Но главная печаль в эом, что даже когда я увижу эти изменения, я не пойму, скорее всего, для чего я их вносил год назад. И, вполне возможно, окружающая среда, то есть, другие модули, с которыми взаимодействует этот модуль, могли за это время решительно измениться. И, по хорошему, после этого требуется

тестирование всей системы — а это совсем не система для игры в тетрис. Так жить нельзя. А как надо?

Вначале надо определиться, файлы каких типов мы кладём в базу. То, что я говорю дальше, относится формально к Дельфи, но для других традиционных языков (включая C++) логика та же. Как делать заведомо не надо? Не надо брать всё содержание вашего рабочего каталога и кидать это всё в соответствующий проект базы. При этом в базу отправляются файлы *.DCU (объектные файлы в Delphi), *.EXE (это понятно), *.DSK, *.CFG и, особенно, *.DDP и *.DSM, так и не смог понять до конца, что это такое, а если я что-то понять не могу, скорее всего это какая-то ерунда, проверено многократно.

Попутно в базу отправляются файлы с тильдой, например *.~PAS. Тильдой в Delphi помечаются резервные копии файлов. Ещё в базу стройными колоннами следуют всякие совершенно непонятные файлы, случайно, или не случайно, оказавшиеся в рабочем каталоге — например, файлы данных, на которых производилось тестирование программы.

Так делать не надо. А как надо? Я считаю, для Delphi, хотя это, с поправками, применимо и не только к Delphi но и ко всем вменяемым языкам, включая C++, класть надо файлы типов *.DPR, *.PAS, *.DFM. Если вы уверены, то можно помещать в базу и файлы типа *.RES, но только если в них содержится хоть что-то отличающееся от содержащегося в них по умолчанию. В VSS есть возможность настроить маски (расширения) файлов при первоначальном помещении их и при других операциях. Это никого ни к чему не обязывает, всегда можно взять файл любого типа, но всё-таки как-то дисциплинирует — разумеется, если вам нравится ходить строем самому и заставлять ходить строем других.

Теперь о главном. Не надо помещать в базу одноимённые файлы, пусть даже и в разные проекты. Это не проблема базы, то есть у VSS с этим как раз никаких проблем не будет. Это ваша проблема, с системой контроля версий никак не свя-

званная. Система контроля её только делает ярче, выпуклее и нагляднее. А проблема в том, что есть две разновидности программистов. Одни правильные, вроде меня, которые считают, что недопустимо иметь разные версии одного и того же файла. Неправильные программисты полагают, что это нормально.

Как это вообще происходит? Да как и всё в программировании.

«Как оказывается, что проект отстаёт от графика на год? — постепенно». © Ф.Брукс.

Например, есть программа, пусть даже большая и сложная, называется PR-1. В программе есть модуль по имени Modul.pas. Задача модуля — обеспечить интерфейс с железным устройством, назовём это устройство «Изделие №1». Чуть позже возникает заказ на изготовление программы, назовём её PR-2. Программа делает всё то же, что и программа PR-1, *за исключением* того, что работает с железякой под названием «Изделие №2». За работу с железом отвечает модуль, напоминаю, Modul.pas. Чтобы переориентировать его на другое устройство, надо поменять в нём всего три строки из трёх тысяч. Поэтому плохой программист копирует весь проект в другой каталог, переименовывает в PR-2 и меняет в модуле Modul.pas эти самые три строки.

Со временем, само собой, появляются и другие изделия с другими номерами. Для каждого из них создается переименованная копия проекта, а в многострадальном модуле опять меняются строчки. Легко и просто. Плохой программист считает, что так и надо. Убедить его в обратном настолько же трудно, как и убедить его в том, что он плохой программист.

Я считаю, что так неправильно. А как правильно? Да как угодно, но не так. Можно сделать один универсальный Modul.pas для всех изделий с переключателем-параметром на вход. Лучше, конечно, оформить его в виде класса. Можно сделать для каждого изделия свой Modul1.pas, Modul2.pas и так далее, с изменёнными строчками в каждом. Лучше, конечно, вынести общую часть за скобки, в отдельный модуль, а в нумерованных модулях оставить только эти самые три строчки.

Это теория, а теперь практика применительно к VSS. Если работа с базой начинается с нуля, то есть одновременно с созданием нового проекта или проектов, то всё относительно хорошо. Главное, с самого начала не допускать помещения в базу файлов с дублированными именами и, наоборот, одинаковых файлов с разными именами. Решается это, к сожалению, только административными методами (= бить по голове).

Чаще переход к использованию системы контроля версий происходит, когда проект уже разрабатывается пару-тройку лет и насчитывает десятки подпроектов и несколько сотен или тысяч файлов. Среди них, в разных каталогах, наверняка окажутся:

- файлы с одинаковыми именами, но с разным текстом;
- файлы с разными именами, но с одинаковым текстом;

— файлы с одинаковыми именами и одинаковым текстом.

Последний вариант возникает обычно безо всякого злого умысла, а вследствие плохой организации — файлы использующиеся в нескольких проектах не складываются в отдельный каталог, а дублируются в каждом каталоге, который их использует. Даже в случае правильной организации — общие файлы в одном каталоге — такое часто случается. Неуверенный в себе программист меняет файл и, опасаясь кинуть его сразу в общий каталог, тестирует с одним проектом, сделав в памяти закладку — разместить его потом в общий каталог. Об этом, само собой, забывает. Результат нехороший — потом корректируется один из экземпляров файла, а все другие — нет.

Так или иначе, перед первоначальным размещением файлов в базе, дубликаты всех перечисленных выше разновидностей надо истребить. Истреблять руками — невозможно, даже если заставить сотрудников, я пробовал. Вариант номер один — взять какую-то готовую программу, предназначенную конкретно для этого, а такие есть. Например уже упомянутая древнейшая, и видимо уже не поддерживаемая Duplicate Files Finder (DFF). От того, что программа уже поросла мхом и заросла паутиной, работать она не перестала. Для личного употребления, конечно, не для коммерческого.

Если в базе нашёлся один неведомо когда и зачем взятый оттуда файл, значит, их найдётся много. Я заглянул сегодня в неё, базу. И не просто заглянул, я трудолюбиво прошёлся по всем проектам, принадлежащим мне, и обнаружил десять—двенадцать файлов, находящихся в состоянии Check Out. То есть, я когда-то взял их, чтобы внести какие-то, скорее всего, очень важные изменения и потом положить обратно. Но почему-то не положил. А почему? Просто забыл, или у меня возникли сомнения насчёт внесённых изменений? А изменения эти были сделаны? Точнее, сделаны ли изменения в их локальных копиях? Проблема в том, что некоторые их файлов взяты из базы два года назад. Ну не помню я, не помню, что там было! И что теперь делать? В данном конкретном случае я собрался с духом и отважно проделал для всех файлов операцию Undo Check Out — иначе говоря, сделал вид, что ничего и не было.

Не надо оставлять все файлы не глядя в состоянии Check Out. Точнее, вообще не надо оставлять файлы в этом состоянии, потому что потом вспомнить невозможно, зачем их брал, что хотел сделать и сделал ли.. Практические рекомендации? Для себя я решил так — файл должен быть возвращён в базу в тот же день, когда и взят. Или с внесёнными изменениями, или без них. Возражение — бывают настолько сложные изменения, что внести их за день невозможно. Встречное возражение — вы всё равно всё это забудете. Через три дня вы забудете о тех изменениях, которые хотели внести, через неделю забудете о том, ради чего нужны были эти изменения. А через две недели забудете о файле, и он будет лежать в состоянии Check Out годами. Я знаю, я много раз проверял на себе.

А если изменения действительно сложные? Банальный и неинтересный совет — сначала отработать изменения на локальной копии, и только затем брать файл из базы, вносить в него изменения и класть файл обратно. Да, это неинте-

ресно и требует дисциплины. Что хуже, снижается эффект использования VSS — фактически мы имеем две версии файла, расхождения между которыми растут с каждым днём.

Тогда другой, не такой банальный, но и не такой простой совет. Правильно проектируйте программу, правильно разбивайте программу на модули, модули правильно разбивайте на функции, или классы на методы. Устраняйте избыточные зависимости между функциями, убивайте глобальные переменные.

Там шпионки с крепким телом, ты их в дверь, они в окно © Высоцкий

В наше время, время классов и объектов, выгнанные в дверь глобальные переменные упорно пролезают обратно в окно под видом секции **private** разрабатываемого класса. Давить! И душить! Все переменные этой секции должны быть привязаны к соответствующим свойствам. Лучше прямо, не получается — косвенно.

И ещё — тщательно следите за правами на доступ. Чётко определяйте кто имеет право на чтение, кто на запись, кто на удаление и, может быть, уничтожение. И права эти, разумеется, назначать не для всей базы в целом, а для каждого проекта отдельно. Лучше дать прав меньше, чем больше. Если прав для кого-то на самом деле будет мало, потом сам придёт и попросит, а мы ещё подумаем, давать или нет. Не бойтесь кого-то обидеть. Обидчивых программистов не бывает, как и нервных. Они умирают молодыми.

ГЛАВА 6

О скучном — динамические библиотеки

Что это такое, а также историческая справка, которую можно не читать

У каждого предмета, материального и нематериального, есть своё назначение. Я в первую очередь о тех вещах, которые даны нам не от природы, а созданы руками человека. Со временем назначение предмета может меняться, причём очень сильно, меняются при этом и достоинства и недостатки. Конечно, физически предметы не меняются, просто новые поколения видят в старых вещах что-то новое, другие достоинства и другие недостатки, а старые уходят на задний план.

Героин когда-то рекламировался как лекарство от кашля, алюминий был драгоценным металлом, платину выбрасывали на помойку, а из титана делали лопаты для разгребания снега. Десять царей, цариц, императоров и императриц России под страхом каторжных работ и смертной казни запрещали купцам вывозить за границу чудодейственное лекарство от всех болезней — ревеня. Потом выяснилось, что годится он в основном для приготовления варенья. Только не надо думать, что русские были такие глупые. Ревень скупало государство в казну. А весь закупленный государством ревеня шёл на экспорт в Европу на вес золота. А европейцы лечились и радовались.

Тот же самое произошло и с динамическими библиотеками. Но сначала прочитайте о том, что это такое, если вы, конечно, этого ещё не знаете. Динамические библиотеки — взгляд снаружи — это такие файлы с расширением *.DLL (но могут быть и другие расширения). Внутри они содержат программный код, точно такой же, как и исполняемые модули (программы), с расширениями *.EXE. Разница в том, что если мы запустим на выполнение программу, то она, извините за тавтологию, выполнится. Динамическую библиотеку запустить не удастся. Почему? Для этого надо заглянуть внутрь.

А что у неё внутри, с точки зрения программиста, не процессора? Внутри у неё набор процедур и функций. В этом отношении библиотека очень похожа на программный модуль (**unit** — в терминологии Delphi). Только, в отличие от обычной программы, в библиотеке, как и в модуле, нет самой программы. Никто эти процедуры и функции не вызывает, поэтому библиотека, сама по себе, исполнению не подлежит. Кто-то должен эти процедуры вызвать. Вызывает их обычная про-

грамма, разрабатываемая независимо от библиотеки. Программа могла бы просто прикомпоновать к себе исходный текст этой библиотеки, как модуль. Тогда никакой библиотеки и не было бы, и главы этой в книге не было бы тоже. А так у нас получается по сути одна программа, искусственно разорванная на две части или больше, если библиотека не одна. Зачем это надо?

А вот здесь вспомним казалось бы бессмысленные рассуждения о первоначальном назначении абстрактного предмета и чем всё это для предмета кончилось. Всё это (я о библиотеках) началось в те времена, когда место на жёстком диске было дорого, а оперативная память запредельно дорога. Более-менее серьёзная программа никак не хотела лезть в память целиком, приходилось разбивать её на отдельные мелкие программы, которые загружались по мере надобности, а ненужные при этом из памяти выгружались. Или программа была одна, но в память грузилась только её верхушка, а прочее подгружалось по мере надобности. Звучит похоже, но на самом деле здесь речь идёт о двух принципиально разных способах применения динамических библиотек.

Первый вариант. Наша программа разбита на десять исполняемых модулей. Каждый из этих модулей нуждается в процедурах расчёта синуса, косинуса, быстрого преобразования Фурье, далее добавить по вкусу. В итоге этот, условно говоря, математический модуль, может быть очень значительным по размеру. Если прицепить его в виде исходного текста к каждой программе, то он будет дублирован десять раз, и общий размер Программного Продукта заметно увеличится. Плюс увеличится время загрузки каждой программы в память, тогда это было ощутимо. И, главное, если в памяти одновременно будут находиться все десять программ, памяти банально не хватит.

Решение этой проблемы таково — вся наша математика находится в библиотеке, библиотека грузится в память только один раз, а все программы компонируются без математического модуля. Вместо этого все программы обращаются к одной единственной библиотеке. Сплошная экономия. При этом, с точки зрения программиста, вызов процедуры из библиотеки почти ничем не отличается от вызова процедуры, прикомпонованного к программе в виде модуля.

Есть такое советское кино — «Семь стариков и одна девушка». Здесь мы имеем именно этот случай — семь программ и одна библиотека. Теперь подумаем о другом, неснятом, кино — «Семь девушек и один старик», иначе говоря — одна программа и семь библиотек. Зачем это было нужно? Да всё по той же причине — от бедности, от недостатка аппаратных, железных ресурсов. Памяти нет. Программа не помещается в памяти, которой нет.

Когда я был маленький, и даже динамических библиотек ещё не было, я программировал для советского клона машины PDP-1, фирмы DEC. Очень хорошая была машина, между прочим, не сравнить с так называемой IBM PC. Так вот, у машины этой адресное пространство составляло 64К (прописью — шестьдесят четыре килобайта оперативной памяти). Причем физической памяти было 56К, остальное выделялось под виртуальные адреса подключаемых внешних

устройств, и для размещения программ и данных эта область была недоступна. Операционная система размещалась там же, в этих 56К.

Это я не жалею, никаких особых мучений и ущемлений мы не испытывали но серьёзная программа в память, конечно, не лезла. Для решения этой проблемы программисты изобрели технологию оверлеев (overlay). Смысл в том, что в память грузится головной модуль, который там и сидит постоянно. Вся остальная программа делится на оверлеи, которые подгружаются по очереди — первый загружается, первый выгружается, второй загружается, второй выгружается... В целом понятно. Обязательное требование — чтобы оверлеи ни в коем случае не обращались друг к другу.

Легко видеть, что динамические библиотеки обеспечивают те же возможности, но лучше и гибче. При таком подходе программа сама даёт команду, когда загрузить в память ту или иную библиотеку, а когда её из памяти выгрузить. Вроде бы тоже самое, но всё совсем по другому. Но с тех пор всё изменилось.

А как с этим теперь?

Теперь всё, конечно, совсем не так. Оперативной памяти по-прежнему не хватает, её всегда не хватает, но уже совсем по другим причинам. Память на жёстком диске вообще никого давно не волнует. Волнует конечно, и очень, но только не программистов. Экономить на подцеплении одной библиотеки к десяти программам нет ни малейшего смысла. Ещё меньше смысла в загрузке десяти библиотек поочередно для использования одной программой. Но динамические библиотеки живут и применяются. Говоря метафорически, раньше в микроскоп рассматривали микробов, теперь микроскопом забивают гвозди.

И динамические библиотеки используются теперь, и используются часто, совсем для других целей. Во-первых, для стыковки с программами и модулями, написанными на других языках программирования. Правильно написанная сорок лет назад динамическая библиотека вполне пригодится в хозяйстве и сегодня. Во вторых, для расширения функциональности уже написанных программ — если они написаны, опять-таки, правильно. Если это звучит непонятно, я объясню, но позже.

Ещё раз о динамических библиотеках, немного под другим углом зрения. Динамические библиотеки обычно имеют расширение *.DLL. Это в Windows, динамические библиотеки существуют и в других операционных системах, с другими расширениями. Сокращение это расшифровывается как dynamically loadable libraries — динамически загружаемые библиотеки. Как это ни странно звучит, динамически загружаемые библиотеки могут загружаться динамически или статически, обычно перечисление идёт в обратном порядке — статически или динамически. Статическая загрузка применяются чаще и реализуется она проще, чем динамическая.

Динамическая загрузка библиотеки является исторически традиционной, приносит больше пользы, но немного сложнее. Статическая загрузка проще и, можно сказать, незаметнее. Программист иной раз и не замечает, что статически грузит библиотеку.

В случае, если мы хотим объединить модули, написанные на разных языках программирования, используется обычно статическая компоновка. Используется она потому, что задачу она решает и при этом проще в реализации — а что ещё надо? И ползут они за нами, и ползут, динамические библиотеки пятидесятилетней давности... Что, столько не живут? Живут!

Динамическая загрузка предоставляет гораздо больше возможностей и всего немного сложнее. Как и всегда в этой жизни, именно поэтому используют её заметно реже.

Сначала мы её создадим!

Что неудивительно, прежде чем использовать динамическую библиотеку, её надо создать. Библиотека, с точки зрения Delphi, и других систем программирования — это то, что называется «проект». То есть, программа, которая библиотеку использует — проект, и библиотека — тоже проект, они совершенно равноправны. Говоря совсем простым языком, совершенно неважно, кого создавать раньше, главное иметь в голове чёткий план использования библиотеки. Абсолютно обязательно сразу обдумать, с какой целью мы библиотеку создаём. Но эта тема постепенно раскроется в процессе дальнейшего изложения. Что важно сейчас запомнить — это то, что неважно, как мы будем загружать в дальнейшем нашу библиотеку, динамически или статически, библиотека в обоих случаях одна и та же.

Теперь я покажу, как создаётся библиотека в среде Delphi, именно в среде, хотя, разумеется и на языке Delphi. Можно просто взять и написать библиотеку руками, например в блокноте, но Delphi предоставляет для этого некоторый, совсем небольшой, сервис. Даже крошечный сервис.

Выбираем в главном меню File\New\Other... В появившемся окошке, на первой странице выбираем DLL Wizard. Получаем заготовку библиотеки, в значительной части состоящую из объёмного комментария. Комментарий пока что удаляем, не читая. Прочитаем позже. Сохраняем проект под именем statDLL — это как бы символизирует, что наша библиотека будет загружаться статически. Напоминаю, самой библиотеке абсолютно всё равно, как её будут грузить. Не забываем добавить дату создания файла и дату последней модификации. В результате получаем вот это:

```
library statDLL; // 17.10.2014
                // 17.10.2014
```

```

uses
    SysUtils,
    Classes;

{$R *.res}
begin
end.

```

Что делает эта библиотека? Как легко догадаться, ничего! Напомню, занудства ради, что библиотека сама по себе не делает ничего, она только предоставляет набор возможностей. Наша библиотека пока никаких возможностей и сервисов не имеет, но сейчас мы их добавим. Возможностей будет только две — рассчитать площадь круга и объём куба, для начала достаточно. Вот результат и комментарии:

```

library statDLL; // 17.10.2014
                // 17.10.2014

uses
    SysUtils,
    Classes;

{$R *.res}
{-----}
procedure circleS(    R : single;
                    var S : single); cdecl;

begin
    S:=pi*R*R;
end;
{-----}
procedure cubeV(    A : single;
                   var V : single); cdecl;

begin
    V:=A*A*A;
end;
{-----}
exports
    circleS,
    cubeV;

begin
end.

```

Появилась секция **exports**. В ней перечислены все наши процедуры, но без параметров, только имена. Точнее, перечислены те процедуры, которые мы экспортируем, то есть, делаем их доступными для внешнего мира. Если у нас есть служебные процедуры или функции, которые используются нами внутри библи-

отеки, но пользователю мы доступа к ним не предоставляем, то в секцию **exports** включать их не надо.

Секция **begin end** выполняет функции секции инициализации обычного программного модуля, нам инициализация пока не нужна. Остаются два вопроса. Первый — почему мы написали процедуры, а не функции, что было бы естественнее? Второй — что за волшебное слово **cdecl**? Ответ будет, но в следующем разделе.

В результате компиляции проекта мы получим файл `statDLL.dll`. Как легко догадаться, сам по себе он совершенно бесполезен, необходима программа, которая его вызовет. Но, прежде чем заняться этой программой, желательно проделать некоторую подготовительную работу. Можно, конечно, и не делать, но если сделать, то в дальнейшем жизнь станет легче и проще.

Изготовим модуль. Не проект, просто модуль. Изготовим его по возможности аккуратнее обычного, потому что немедленно, транслируя, проверить его правильность не получится. Не получится потому, что на данном этапе он, сам по себе, не включен ни в какой проект и, стало быть, транслироваться не будет вообще. Разумеется, если в нём есть ошибки, мы об этом узнаем, но позже, когда попытаемся этот модуль использовать.

```

unit statDLExt; // 17.10.2014
                // 17.10.2014

interface
  const
    DLLName = 'statDLL';
  {-----}
procedure circleS(      R : single;
                      var S : single); cdecl;
    external DLLName name 'circleS';

procedure cubeV(      A : single;
                      var V : single); cdecl;
    external DLLName name 'cubeV';
  {-----}
implementation
end.

```

Думать над содержимым модуля сейчас не надо, надо просто понять алгоритм его создания. В начале объявлена строковая константа с именем нашей библиотеки. Это как бы даже и очевидно. В основной части идут объявления процедур, экспортируемых библиотекой, в той виде, как они там объявлены, в точности — это я намекаю на слово **cdecl**. После описания интерфейса процедуры идёт ещё одна строка, структура её понятна, или должна быть понятна любому сколько-то разумному программисту. В строке две величины, которые можно изменить, но лучше не менять. Это всё тяжёлое наследие тех времён, когда библиотеки имели совсем другой смысл, чем сейчас.

Повторяю, модуль можно не писать, но с ним проще, потому что текст этот всё равно написать придётся, и, возможно, не один раз.

Статическая загрузка и как это может пригодиться

Сначала освоим статическую загрузку. Это не просто, а очень просто.

```
uses
    statDLLExt;
.....
var
    S                : single;
    V                : single;
begin
    circleS( 10, S);
    ShowMessage( 'S = ' + FloatToStr(S));

    cubeV( 3, V);
    ShowMessage( 'V = ' + FloatToStr(V));
end;
```

Никакой разницы по сравнению с вызовом обычного, не библиотечного, модуля. Так и напрашивается вопрос — и ради чего все эти нелепые телодвижения?

То, что мы сейчас наблюдали — это стандартное поведение динамической библиотеки, то применение, ради которого она и была задумана. Сейчас в таком использовании библиотек очень мало смысла, если он вообще есть.

Для чего это нужно? Да для того, о чём я уже говорил — стыковки программных модулей, написанных на разных языках программирования. Когда-то, когда я был маленький, такой проблемы не было — ну, или почти не было. *«В те времена былинные, теперь почти забытые»* © Высоцкий, языки программирования являлись частью операционной системы. Соответственно гарантировалось, что все их объектные коды между собой полностью совместимы на уровне типов — если, разумеется два типа вообще существовали в этих двух языках. С другой стороны, никто не обещал, что будут между собой совместимы два объектных модуля, написанные на одном и том же языке программирования, но в разных операционных системах.

На практике, то есть в нашей Российской практике, объединять приходится Delphi и C++, реже C#. Теме не менее, динозавры живут среди нас. Живут библиотеки написанные на Фортране неизвестно какой версии в неизвестно каком году. И хорошо живут.

Сначала рассмотрим ситуацию, когда есть некоторый код на Delphi, а его предполагается использовать в программе, написанной на C++, Вопросов два. Первый — что передавать, то есть, какие типы? Второй вопрос — как передавать?

Сначала первая проблема — совместимость типов данных. Понятно, что о передаче объектов в качестве параметров и речи быть не может.

Приглядимся к строкам, здесь ситуация чуть лучше, но ненамного. В Си при-знаком конца строки является нулевой байт, который не может в строке содержаться, что очевидно, зато строка может иметь произвольную длину. В Delphi нулевой символ в строке никого не волнует, это хорошо. Плохо то, что такую строку передать в Си невозможно вообще. Есть короткие строки, доставшиеся по наследству от Турбо Паскаля, аналога им в Си просто нет. Есть длинные строки, являющиеся псевдоклассом, и в таком качестве аналога им тоже нет, само собой. Идеально передаются только массивы символов:

```
const
```

```
  N = 128;
```

```
type
```

```
  TSarray = array[1..N] of char;
```

Маленький нюанс заключается в том, что программы (или программисты) должны заранее договориться между собою о том, чему равно это самое N, Программно это не решается, только административно.

Экзотические типы забываем — например, множества. С булевскими переменными тоже какая-то ерунда, то есть несовместимость. Если уж обязательно надо что-то передать булевское (да/нет), придется заменить целочисленным типом (1/0).

А что у нас остаётся хорошего? Точнее, что можно передать безопасно? Общий принцип — чем примитивнее тип, тем легче его передать. В стыковке с Фортраном никаких проблем никогда ни у какого цивилизованного языка не возникало, ввиду его, Фортрана, неискоренимой незатейливости. Пристально рассматривая Delphi, находим самые в нём незамутнённые типы — `integer` и `single`. Разумеется, и в Delphi и в C++ есть аналогичные типы, отличающиеся количеством байт, разрядностью и знаковостью — но зачем? С ними тоже всё будет в полном порядке при использовании — но, опять-таки, зачем? Для всех разумных целей нам хватит `integer` и `single`.

Теперь ещё раз и коротко — что можно использовать при работе с динамическими библиотеками и гарантированно не иметь проблем.

Любые целые и плавающие типы — разумеется, если они имеют соответствия в языке, на котором библиотека написана и в языке, который её использует.

Символы (`char`) — это ведь просто байт.

Указатели — вопрос сложный и философский. С одной стороны, указатель — это всего лишь число, с другой стороны в языке, из которого библиотека вызыва-

ется, указателей может не быть вообще. С третьей стороны — а на что этот указатель после передачи будет указывать.

Булевские — можно, но есть нюансы.

Длинные строки (**string**) — тоже можно, но тоже есть нюансы. В первую очередь, конечно, речь идёт о C++, остальные языки представляют не такой практический интерес. Причём при вызове в ту или в другую сторону (из Delphi в C++ или наоборот) нюансы будут разные.

Что-то другое использовать тоже можно, но не говорите, что я вас не предупреждал.

Замечание как бы сбоку. Очень неплохо своими собственными руками написать несложную программу, переводящую интерфейсный дельфийский файл в соответствующий файл заголовка в C++. И не только интерфейсный файл, но и все соответствующие объявления типов и констант. Это несложно, я пробовал.

Динамическая загрузка и расширение возможностей

Библиотека у нас пока остаётся прежней — statDLL.dll. Модуль statDLLExt.pas нам не нужен вообще (пока не нужен). Где-то в секции интерфейса объявляем процедурный тип:

```
type
  TCircleProc = procedure (      R : single;
                             var S : single); cdecl;
```

Что такое процедурный тип, я объяснять не буду, а если кто-то забыл, то об этом написано в Приложении. Как прямо-таки бросается в глаза, тип в точности повторяет заголовок процедуры расчёта площади круга из нашей библиотеки. Обратите внимание, что волшебное слово **cdecl** перекочевало в объявление типа.

А теперь вызов соответствующей процедуры из библиотеки, ставший заметно более сложным:

```
var
  Calculate           : TCircleProc;
  DLLName            : string;
  procName           : string;
  handle             : THandle;
  Ok                 : boolean;
  S                 : single;
begin
  Ok:=true;
  Calculate:=nil;
```

```

DLLName:='statDLL.dll';
handle:=LoadLibrary(PChar(DLLName));
if handle <> 0 then begin
    procName:='circleS';
    @Calculate:=GetProcAddress( handle, PChar(procName));
    if @Calculate = nil
        then Ok:=false;
end
else begin
    Ok:=false;
end;

if Ok
    then Calculate( 10, S);

ShowMessage( 'S = ' + FloatToStr(S));

if Ok
    then FreeLibrary(handle);
end;

```

Сначала разьясню тёмные моменты, а потом отвечу на вопрос — и ради чего такие мучения? Работа с библиотекой симметрична — загрузка (LoadLibrary) и выгрузка (FreeLibrary). Выгружать библиотеку можно и нужно только в том случае, когда загрузка прошла успешно, то есть handle <> 0. После загрузки библиотеки мы ищем в ней нужную нам функцию, по имени функции. Для этого, как легко догадаться, нужно знать имя этой функции. В нашей библиотеке, напоминаю, две функции — circleS и cubeV. Мы вспомнили имя только первой и её вызвали, а про вторую забыли. Можно ли узнать, какие функции содержит библиотека? Можно, конечно, только зачем? Подумайте об этом. Задайте вопрос *об этом* на дельфийском или ином форуме. Прочувствуйте ответы. Насладитесь. Оно вам надо?

Это я ни в коем случае не хочу вас обидеть. Это я показываю вам непростую и нелёгкую жизнь молодого программиста.

Функция PChar делает из длинной строки Delphi как бы сишную строку. Это необходимо потому, что LoadLibrary и GetProcAddress являются функциями Windows API с соответствующим интерфейсом, Delphi предоставляет только интерфейс-обёртку для их вызова. По этой же причине тщательно следите за регистром символов в строке — здесь вам не тут!

Оператор Calculate:=nil; присутствует исключительно для избавления от назойливого сообщения *Warning: Variable 'Calculate' might not be initialized*. Я, повторюсь, перфекционист, и не терплю в результатах трансляции никаких Hint и Warning. Только, я таки вас умоляю, не надо объяснять мне за то, что булевскую переменную Ok можно было в таком случае и не вводить. В данном конкретном случае я совершенно не согласен с Керниганом (и Плоджером), категорически

требующим избавлять программу от промежуточных и вспомогательных переменных.

Напоследок, в двадцать шестой раз, напомню, что и при статическом и при динамическом вызове (точнее, загрузке) мы обращаемся к одной и той же библиотеке, то есть к одному и тому же физическому файлу.

Польза от динамического вызова

Мы вызвали процедуру из библиотеки статически. Это было просто. Мы вызвали процедуру из библиотеки динамически. Это было сложно. Естественный вопрос — а зачем? То, что мы можем узнать, какие в незнакомой нам библиотеке есть функции — преимущество сомнительное. Если мы заранее не знаем, что содержится в библиотеке, зачем нам такая библиотека, и как вообще мы её собираемся использовать?

Нет, давайте исходить из того, что мы прекрасно знаем, какие функции и с каким интерфейсом содержатся в нашей библиотеке, и покажем, какой от этой библиотеки может быть толк. Создадим новую библиотеку, пустую, но теперь внимательно ознакомимся с её содержимым. Такой комментарий встречает нас при входе:

Important note about DLL memory management: ShareMem must be the first unit in your library's USES clause AND your project's (select Project-View Source) USES clause if your DLL exports any procedures or functions that pass strings as parameters or function results. This applies to all strings passed to and from your DLL—even those that are nested in records and classes. ShareMem is the interface unit to the BORLNDMM.DLL shared memory manager, which must be deployed along with your DLL. To avoid using BORLNDMM.DLL, pass string information using PChar or ShortString parameters.

Объясняю на пальцах. Если в нашей библиотеке есть хоть одна процедура, которая получает или возвращает в качестве параметра строку, даже если эта строка не сама по себе, а внутри, к примеру, записи, то мы просто так этого сделать не можем. Мы должны проделать три телодвижения.

Первое. В исходный текст нашей библиотеки, в раздел (clause) **uses**, причём в тот **uses**, который наверху, в секции **interface**, вписать модуль **ShareMem**, причём обязательно первым в списке. Исходный текст библиотеки я приведу чуть дальше, целиком.

Второе. Аналогичную манипуляцию надо проделать с исходным файлом проекта. Обратите внимание — не с модулем, в котором мы вызываем библиотечную функцию, а именно с исходным файлом проекта, тот, который *.DPR. Чтобы до

него добраться, надо из головного меню выбрать Project\View Source. Обычно туда не ходят, а если пошли — значит, что-то пошло не так, извините за каламбур. После нашего вмешательства исходный текст проекта будет выглядеть приблизительно так:

```
program _UsProgr;

uses
  ShareMem,
  Forms,
  UsProgr in 'UsProgr.pas' {Form1},
  Prime in 'Prime.pas',
  UsSort in 'UsSort.pas';

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Как легко догадаться, весь текст программы, кроме одной строки сгенерирован автоматически.

Третье. Мы должны гарантировать, что динамическая библиотека BORLNDMM.DLL всегда будет находиться в том же каталоге, что и наша программа. Это я упрощаю и примитивизирую. Разумеется, её можно положить и в какое-нибудь общедоступное место, например C:\Windows. Перечитайте предыдущую главу о VSS и, в особенности, раздел об изготовлении Программного Продукта одной кнопкой.

Четвёртое, дополнительное. Первые три пункта можно забыть, если передавать строки с помощью функции PChar. Говоря иначе, все предыдущие заклипания необходимы только в том случае, если мы вызываем из Delphi библиотеку, на Delphi и написанную. Напоминаю, что в этом случае **cdecl** можно убрать. Убрать, разумеется, с обеих сторон — и в программе, и в библиотеке.

А теперь наша новая библиотека:

```
library fSin; // 23.10.2014
              // 24.10.2014

uses
  ShareMem,
  SysUtils,
  Classes;
```

```

{$R *.res}
{-----}
procedure funcName( var name : string); cdecl;
begin
    name:='Синус';
end;
{-----}
procedure funcValue(      x : single;
                        var y : single); cdecl;
begin
    y:=Sin(x);
end;
{-----}
exports
    funcName,
    funcValue;
begin
end.

```

Сначала о формальной стороне. Имя библиотеки прямо-таки кричит о том, что она может сделать — а может она сделать синус. Имена процедур, напротив, совершенно обезличены, и только намекают, что одна процедура возвращает имя рассчитываемой функции, а другая — значение функции. Но ни слова о синусе! И это не случайно.

Само собой разумеется, оставим в стороне вопрос о том, зачем мы запрограммировали процедуру для вычисления синуса, которая обращается к функции вычисления синуса, которая уже запрограммирована задолго до нас. Так надо. На её месте мог бы оказаться каждый.

На всякий случай, приведу и вызывающую программу, но, для лучшего понимания и лаконичности, без контроля загрузки и наличия процедур в библиотеке. Это неправильно! Так не надо! Это только для того, чтобы лучше было видно голые сучья деревьев за листьями. А вы так не делайте:

```

var
    funcNameProc      : TFuncNameProc;
    funcValueProc     : TFuncValueProc;
    DLLName           : string;
    procName          : string;
    handle            : THandle;
    Y                 : single;
    funcName          : string;
begin
    DLLName:='fSin.dll';
    handle:=LoadLibrary(PChar(DLLName));

```

```

procName:=' funcName' ;
@funcNameProc:=GetProcAddress( handle, PChar(procName));

procName:=' funcValue' ;
@funcValueProc:=GetProcAddress( handle, PChar(procName));

funcName:=' None' ;
funcNameProc( funcName);

Y:=0;
funcValueProc( pi/2, Y);

ShowMessage( 'Name = ' + funcName + '   Y = ' +
FloatToStr(Y));

FreeLibrary(handle);
end;

```

Тем не менее, за деревьями по-прежнему не видно леса, то есть пользы от динамической загрузки библиотеки. Абстрактное замечание — через одну точку можно провести бесконечное количество прямых, а через две точки — только одну. Говоря иначе, смастерим ещё одну библиотеку и посмотрим, к чему это всё клонится.

```

library fSquare6; // 25.10.2014
                // 25.10.2014

uses
    ShareMem,
    SysUtils,
    Classes;

{$R *.res}
{-----}
procedure funcName( var name : string); cdecl;
begin
    name:='Площадь шестиугольника';
end;
{-----}
procedure funcValue(      x : single;
                        var y : single); cdecl;
begin
    y:=((Sqrt(3))/2) * x*x;
end;
{-----}

exports
    funcName,

```

```
funcValue;
```

```
begin
end.
```

Библиотека не очень полезна, зато не имеет замены среди стандартных функций — библиотека вычисляет площадь правильного шестиугольника по его стороне, или по большому радиусу. В качестве несложного упражнения рекомендую вам изготовить по такому же рецепту библиотеки для вычисления косинуса, факториала, синуса и пресловутой функции Дирихле.

А мы пока обратим внимание на то общее, что есть у этих двух библиотек. Сначала о внешнем сходстве. У библиотек похожие имена — сначала идёт символ “f”, а потом всё остальное, указывающее назначение библиотеки, но это сейчас значения не имеет. Внутри библиотек, что важнее, ровно по две процедуры в каждой, а имена у них стандартные — funcName и funcValue. И что мы можем из этого извлечь?

Я предлагаю написать программу — Универсальный Вычислитель Функций. А если не абсолютно универсальный, то по крайней мере — расширяемый. Но только не вообще всех возможных функций, а только тех, которые получают на вход один аргумент типа single и возвращают значение тоже типа single. Расширяемость программы означает, что добавить в неё вычисление ещё одной функции может не только её разработчик, но и совершенно посторонний программист, причём, даже не имеющий доступа к исходному коду головной программы.

Замысел такой — мы, именно мы, пишем головную программу. Программа выводит на экран красиво оформленный список доступных функций с кнопками. Красиво оформленный — надо понимать, красиво в моём представлении. При нажатии на кнопку вылезает окошко с запросом аргумента X, затем выводится ответ — тот самый Y для выбранной функции. А те, кто захотят расширить функциональность нашей программы, вообще не обязаны заглядывать в её исходный текст, который им в общем случае недоступен.

Всё, что требуется от внешнего разработчика — это создать библиотеку, имя которой имеет символ “f” в начале. Это разумеется, не главное требование, но самое простое и необходимое. Главное то, что эта библиотека должна иметь такую секцию экспорта:

```
exports
  funcName,
  funcValue;
```

Интерфейс процедур должен быть таким:

```
procedure funcName( var name : string); cdecl;
procedure funcValue( x : single;
                    var y : single); cdecl;
```

Теперь о том, как мы собираемся действовать дальше. Мы ищем в том же каталоге, где находится наша головная программа, все файлы, имя которых начинается с символа `f`, и которые при этом имеют расширение `*.DLL`. Если вы считаете, что одного `f` недостаточно для уверенной идентификации какой-то библиотеки именно в качестве нашей особенной библиотеки, то вы совершенно правы. Я тоже так думаю, но что сделано, то сделано, а вы можете исправить ситуацию, сделайте, как лучше.

Далее мы создаём кнопки, в количестве, равном количеству наших библиотек, и на каждую кнопку выводим текст (Caption), Текст, как нетрудно догадаться, мы получим от функции `funcName`. А когда на кнопку нажмут, вызываем соответствующую ей, кнопке и математической функции, программную функцию `funcValue`.

Разумеется, никаких проверок, что в библиотеках присутствуют нужные нам функции, мы проводить не будем — а зачем? Ведь мы совершенно уверены, что у нас всё в полном порядке и никаких несоответствий быть не может. Шучу. То есть проверять мы действительно ничего не будем — наша цель просто и кратко показать принцип работы программы. Но в реальной жизни, всё, конечно, будет не так. Всё, что может сломаться, сломается, не сомневайтесь. Проверять, в наличии ли функции, которые мы собираемся вызывать, необходимо. А пока необходимо знать, что проверка возможна.

Далее я предполагаю, что любезный читатель знаком с устройством компонентной системы Delphi — в частности, что если у нас есть кнопка, то у этой кнопки есть обработчик, который отвечает за то, что делает эта кнопка при её нажатии. Подключение обработчика очень просто, если заранее известно количество кнопок и количество разработчиков и их не очень много.

Следующий уровень знания — как подключить к нескольким кнопкам один и тот же обработчик. Это некоторым программистам уже недоступно. Ещё полступени вверх — обработчик один на всех, но он должен выяснить, какая именно кнопка его вызвала и реагировать адекватно, соответственно обстановке, то есть нажатой кнопке.

Говоря проще, обработчик у нас будет один, на все кнопки сразу. С этим понятно. Остаётся вопрос с кнопками — сколько их будет, мы заранее не знаем. Был бы я моложе, предложил бы хранить их в массиве. Теперь я поумнел и предпочитаю списки.

Библиотеки – технические подробности

Первая проблема с библиотеками — а как их отлаживать? Здесь и далее я говорю о Delphi, но проблема везде та же. Если мы хотим выполнить программу в пошаговом режиме, то над вызовом функции из библиотеки мы стремительно

пролетаём, в буквальном смысле. А если мы загружаем в качестве проекта библиотеку, то запустить её, ясное дело, невозможно.

Проблема решается очень просто, надо только знать, где искать. Загружаем библиотеку как проект. В главном меню выбираем Run\Parameters... На первой странице (Local) в строке Host Application путём нажатия кнопки Browse выбираем нашу программу, ту, которая вызовет нашу библиотеку. Естественно, что программа к этому моменту должна быть уже скомпилирована, и присутствовать на диске.

Теперь, когда мы находимся в проекте нашей библиотеки и перед нами на экране её текст, мы устанавливаем Breakpoint в исходном коде библиотеки. Затем нажимаем кнопку Run, запустится та программа, которую мы выбрали — та, что называется Host Application, то есть «Хозяйское приложение» для нашей библиотеки. Проводим необходимые манипуляции с программой, чтобы обеспечить вызов библиотеки, попадаем на точку останова. Далее как обычно.

Реализация проекта

Динамические библиотеки для опытов у нас уже есть — целых две, fSin.dll и fSquare6.dll. Слона будем есть по частям. Часть первая, к библиотекам и кнопкам не относящаяся:

```
var
    SR                : TSearchRec;
    rez               : integer;
    DLLName           : string;
begin
    ChDir(ExtractFilePath(ParamStr(0)));

    rez:=FindFirst( 'f*.dll', faAnyFile, SR);
    nomer:=0;

    while (rez = 0) do begin
        DLLName:=SR.Name;

        //  загрузка библиотеки и поиск в ней функций
        //  здесь создаём кнопку и добавляем её в список

        rez:=FindNext(SR);
    end;

    FindClose(SR);
end;
```

Самого главного пока нет, есть только очевидное. Обратите внимание на первый оператор. Мы требуем перехода в каталог. Имя каталога мы получаем, отбрасывая имя некоторого файла и оставляя путь к нему. Имя и путь мы получаем, запрашивая параметр с которым вызвана наша программа. Никаких параметров при вызове нашей программы нет, но мы требуем параметр номер ноль. Параметр с таким номером всегда есть. В этом случае функция возвращает полное имя нашего исполняемого модуля. Здорово, правда? С первого взгляда видно, что вся эта фигня сводится к переходу в тот каталог, из которого запущена наша программа. Нам это надо — ведь там лежат, скорее всего, наши библиотеки. Манипуляция эта выполняется настолько часто, что я очень советую немедленно оформить этот оператор в виде процедуры и сохранить в надёжном сухом месте. Пригодится ещё много раз.

Загрузку библиотеки по её имени мы уже отработывали, повторяться не будем. Имена функций, извлекаемых из каждой библиотеки нам заранее известны — `funcName` и `funcValue`. Обратите внимание, что пока нам нужна только первая функция. Мы создаём кнопку, на которой написано имя функции. При нажатии на кнопку будет вызвана сама функция, в обработчике кнопки, который, напоминаю, один для всех кнопок. Технология поиска файлов по маске имени вам должна быть уже знакома. По крайней мере, я на это надеюсь, этот нехитрый метод знать обязательно надо, независимо от применяемого языка программирования.

Объявляем ранее недообъявленные переменные. Сначала, где-то внутри нашего класса мы объявляем переменную типа список строк:

```
private
    bSL                                : TStringList;
```

Затем, в самом начале исполнения нашей программы мы её инициализируем:

```
procedure TfmUniversal.FormCreate(      Sender : TObject);
begin
    bSL:=TStringList.Create;
end;
```

В симметричном месте нашей программы она уничтожается, но это неинтересно, потому что очевидно.

А вот промежуточный результат:

```
var
    funcNameProc                : TFuncNameProc;
    SR                          : TSearchRec;
    rez                          : integer;
    DLLName                      : string;
    handle                       : THandle;
```

```
    funcName           : string;
    procName           : string;
    btn                : TButton;
    nomer              : integer;

begin
    ChDir (ExtractFilePath(ParamStr(0)));

    rez:=FindFirst( 'f*.dll', faAnyFile, SR);
    nomer:=0;

    while (rez = 0) do begin
        DLLName:=SR.Name;
        handle:=LoadLibrary(PChar(DLLName));

        procName:='funcName';
        @funcNameProc:=GetProcAddress( handle, PChar(procName));

        funcName:='None';
        funcNameProc( funcName);

        btn:=TButton.Create(Self);
        btn.Parent:=Self;
        nomer:=nomer + 1;
        btn.Tag:=nomer;
        btn.Width:=200;
        btn.Left:=(nomer-1)*(btn.Width+10) + 30;
        btn.Top:=30;
        btn.Caption:=funcName;
        btn.OnClick:=myBtnClick;

        bSL.AddObject( DLLName, btn);

        FreeLibrary(handle);

        rez:=FindNext(SR);
    end;

    FindClose(SR);
end;
```

Кнопки располагаются в ряд в левом верхнем углу. Обработчик у всех кнопок общий, задаётся оператором `btn.OnClick:=myBtnClick`; Обработчик нами пока ещё не написан. Что он должен содержать, в целом понятно — загрузка библиотеки по имени, поиск в ней функции `funcValue` и вызов её с правильным аргументом. Надо обратить внимание на кучерявый оператор `bSL.AddObject(DLLName, btn)`; У списка строк есть несколько искусственная

возможность, позволяющая добавить в список не только строку, но и связанный с ней тем же индексом указатель на что-то. По сути, список строк неявно содержит в себе ещё и традиционный классический список TList. Мы отправляем туда экземпляр класса TButton, который, не устаю напоминать, является формально тем же указателем.

Ещё заметьте применение абсолютно бесцельного свойства Tag. Почему бесцельного? Потому, что в описании его так и сказано — никакого специального назначения это свойство не имеет и может быть использовано программистом так, как ему, программисту, захочется. Все четыре байта этого свойства. Вот мы его и используем для хранения номера записи в списке bSL, который, напоминаю, уже содержит полный список созданных нами кнопок.

Осталось создать обработчик нажатия на кнопку. Это формально, а в реальности всё немного сложнее — ведь мы не знаем заранее, нажатие какой именно кнопки мы должны обработать. По сути, там, в обработчике, мы должны определить кнопку и по имени добавленного в список объекта (TButton) определить, какую именно библиотеку загружать, запросить аргумент функции и вывести значение функции.

```

procedure TfmUniversal.myBtnClick(      Sender : TObject);
var
    funcValueProc      : TFuncValueProc;
    DLLName            : string;
    handle             : THandle;
    procName           : string;
    xStr               : string;
    X, Y               : single;
begin
    DLLName:=bSL[TButton(Sender).tag-1];
    handle:=LoadLibrary(PChar(DLLName));
    procName:='funcValue';
    @funcValueProc:=GetProcAddress( handle, PChar(procName));

    xStr:='0';
    InputQuery( TButton(Sender).Caption, 'X = ', xStr);
    X:=StrToFloat(xStr);

    funcValueProc( X, Y);
    ShowMessage( 'Y = ' + FloatToStr(Y));

    FreeLibrary(handle);
end;

```

Параметр обработчика чего бы то ни было по имени Sender на неподготовленного программиста и меня тоже производит впечатление абсолютно бесполезного предмета. Но нашлось и ему применение. Остальное должно быть понятно.

ГЛАВА 7

Мелкие технические подробности

Легко ли быть жадным?

А главное, нужно ли быть жадным. Нужно ли быть жадным программистом, само собой. Если слов «жадный» кажется вам обидным, то заменим его на «бережливый» или «экономный». Это слова не обидные, а, наоборот, используются исключительно в позитивном смысле.

Я уже писал о разработчиках VSS, которые угробили огромные количества ума и смекалки, не говоря о времени, на отслеживание изменений в текстовых файлах — лишь бы не хранить их в базе полностью. Я восхищаюсь проделанной ими работой, но вся эта работа печально проследовала в задний проход.

Какая из этого следует мораль? Не знаю, я же не работаю в Microsoft, точнее, работаю не в Microsoft, а в совсем другой организации.

Как-то сколько-то очень много лет назад, я написал объект — да, уже тогда существовало ООП. Этот объект представлял собой агрегат данных, каждый из которых хранился в виде файла на жёстком диске. Я приложил максимальные усилия, чтобы при хранении файла на жёстком диске минимизировать перемещения по нему. Это было в высшей степени разумно, ведь максимальное количество файлов в агрегате я установил в целых 128. Типичный объем памяти составлял тогда мегабайт, роскошный — восемь мегабайт. Я был прав. На всякий случай, из последних сил, я добавил возможность хранить все данные в оперативной памяти. Для этого требовалось при создании объекта задать в конструкторе специальный параметр, что выглядело очень коряво, по моим нынешним взглядам.

Через два года про режим по умолчанию (всё на диске) забыли, и все пользователи объекта пользовались только режимом — в оперативной памяти. При этом, поскольку я и не думал, что этот режим будет основным, для него происходят все те же мучительные проверки, не сместились ли мы с нашей позиции в файле, и не пора ли нам переписать изменённые в памяти данные на реальный носитель. И это для оперативной памяти, напоминаю.

Какая из этого следует мораль? Морали сколько угодно. Я был неправ, не оценив скорость развития железа. Те ограничения, которые казались мне непреодолимыми, через два года растаяли. Я был неправ, сделав режим, рассчитанный на богатое будущее, альтернативным и привинченным сбоку, вместо того, чтобы сделать его основным. Я делал его исходя из минимума затрат, считая, что он

вряд ли понадобится. В результате в плане реализации он оказался таким же, как и режим работы с жёстким диском при дефиците оперативной памяти, только хуже. В итоге при режиме (теперь основном) размещения данных в оперативной памяти происходят огромные непроизводительные затраты, связанные с бессмысленными проверками и перемещениями/неперемещениями по потоку.

Единственная причина, почему это никого сейчас не волнует — то, что возможности железа развились до такой степени, что и эти затраты никого уже не волнуют.

Во главное, в чём я тогда ошибся и чего я тогда понять не мог — то, что с тех пор прошло восемнадцать (!) лет и всё это до сих пор работает, и трудолюбивые молодые программисты пишут свои современные программы под эти древние объекты. Это не потому, что я такой очень умный — это потому, что никому это заново переписывать не хочется — работает, и ладно.

Но проблемы не ходят в одиночку. Максимальное количество данных (файлов) в агрегате я определил, волевым порядком, в 128. В каждом из них, в заголовке, то есть в середине файла, я определил резерв в один килобайт. По тем временам это казалось непозволительной роскошью — $128 \cdot 1024 =$ бешеные деньги. Через два года ко мне пришли хмурые потребители, и сказали, что 128 — маловато будет, надо бы 512. И, плюс к этому, необходимые данные, которые, во чтобы то ни стало надо дописать в каждый файл с опасной скоростью стали приближаться к этому злосчастному килобайту.

Печаль в том, что эти 128 элементов (точнее, ссылки на них) были заложены в самом главном файле и увеличить их количество было конечно же можно, но как-то только через какое-то место.

Не говоря о том, что сама идея хранить каждую часть данных в отдельном файле была глупой, жадной и порочной. Но никто же теперь переделывать не будет!

Выводы будут потом. Пока я объясняю ситуацию.

Ещё в моей практике была структура данных чудовищно разветвлённого дерева (в смысле графа) с абсолютно непредсказуемым количеством ветвей. На конце каждой ветви висела запись, она же структура. На разных ветках висели разные типы структур. Количество ветвей с записями исчислялось тысячами, естественно, я старался экономить, и не выделил в записях резерва. Ни одного байта. А потом внезапно ко мне пришли пользователи и ...

А что делать?

Если бы я был таким умным сейчас, как моя жена потом © Одесская шутка

Потом мы все умные. Но какие-то выводы я смог для себя сделать.

Надо ли надеяться на развитие техники? Вы написали программу, она ужасно тормозит сегодня, но завтра, на новом компьютере всё будет летать. Нет, прямо завтра оно летать не будет, а программа нужна вообще-то не завтра, а сегодня. Гениального совета я дать не могу. Над этим думали лучшие программистские умы, куда мне до них. Могу дать просто умный совет. Пишите проще. Если какие-то операции ваша программа сегодня выполняет непозволительно медленно, не пытайтесь оптимизировать их хитрым через что-то способом. Или оставьте всё как есть, и пусть пользователи год-два потерпят. Если программа хоть сколько-то полезная, они потерпят. Или отключите эту функцию вообще. Если программа, опять таки, хоть сколько-то полезная, пользователи обойдутся и без этой функции. Но только не надо хитроумных способов оптимизации, умоляю вас!

Будьте проще — простые алгоритмы, простые типы, простые структуры данных. Не умничайте.

Параллельная проблема. Сейчас мы говорим о сложных алгоритмах в применении к простым данным. А если речь не об алгоритмах? У нас есть массив в 1024 элемента. А может быть завтра надо будет 2048? «Не мелочись, Наденька» © Кино. Если можно написать больше, пишите, пусть даже оно будет тормозить. Пока. Год-два. Потому что потом внести изменения будет дороже. Не рассчитывайте на то, что, изменив всего одну константу вы расширите диапазон работы с данными вашей программы во много раз. Так не бывает, останется куча неучтённых моментов, которые вы будете отлавливать с микроскопом по всей вашей программе. Я знаю, я проходил.

Очень короткие советы. Увеличить размерность, объём памяти или что-то ещё может понадобиться абсолютно в любом месте. Часть того, что вы зарезервируете на будущее, никогда не понадобится. Не слушайте тех, кто говорит об экономии.

Совершенно банальный совет — если хотите сохранить данные, храните их в одном файле, а не россыпью. Точнее, если вы хотите их надёжно сохранить. А до какой степени целесообразно утаптывать все данные в один файл? Ответ банален — крайности не очень хороши.

Закрадывается мысль, что вся эта скорость — просто ерунда. Так стоит ли об этом думать? Несколько цитат на тему скорости из моей любимой книги «Элементы стиля программирования»:

Думать надо о суммарной стоимости — разработка + машинное время

Исправьте программу, прежде чем повышать её скорость

Увеличивая скорость, сохраните программу правильной

Сделайте программу ясной, прежде чем улучшать

Пусть транслятор выполняет элементарную оптимизацию

Удостоверьтесь, что особый случай действительно особый

Сохраните программу простой

Найдите лучший алгоритм

Проведите измерения, прежде чем менять программу.

Теперь конкретно об эффективности

Как математик (в моём дипломе так написано), уверяю вас, что говорить о том, что лучше, а что хуже, можно только в том случае, если мы в состоянии внятно объяснить, как мы это сравнили. Можно сравнивать миллиметры с километрами — мы знаем, как их сравнить. Нельзя сравнивать атмосферы с вольтами — по крайней мере, я не знаю как.

Если мы хотим сравнить, какая программа лучше, мы должны задать простой и понятный метод сравнения. «Лучше» — это слишком сложно и очень расплывчато. Как принято у математиков, надо задать меру, по которой мы будем сравнивать. Общепринятой меры для «лучше», к сожалению нет. Давайте тогда хотя бы сравним, какая программа быстрее — на практике, это то, что видит пользователь в первую очередь. Так уж он, пользователь, устроен. Если программа отъедает всю память, тормозит все остальные процессы и расплзается своими файлами по всему диску, это никому не заметно. А вот задержка в три секунды при нажатии клавиши — это катастрофа.

Поэтому будем измерять время выполнения — программы, модуля, процедуры — и смотреть что лучше. А как измерить время? Способов много. Отличаются они своей точностью, отличаются удобством применения. Отличаются они и тем, что какие-то из них являются специфическими для Delphi, а другие принадлежат Windows API, и, следовательно более, но не абсолютно, универсальны. Из функций API я рекомендую GetTickCount, но, поскольку мы не гонимся за точностью, а желаем лишь удобства, используем взамен технологии Delphi.

Для начала померяем пузырьковую сортировку. Но, как нас учат Керниган и Плотджер в своём бессмертном труде (см. список литературы), *«прежде чем оптимизировать программу, убедитесь, что она правильна»*. Если в массиве пять элементов, можно проверить правильность сортировки визуально. Если их пятьсот, то это становится затруднительным, так что придётся и проверку доверить компьютеру.

```

const
    Nbig = 100000;
var
    a           : array[1..Nbig] of single;
    time0,time1 : TDateTime;
    i           : integer;
{.....}
function IsSorted(    a: array of single) : boolean;
var
    i           : integer;
begin
    result:=true;

```

```

    for i:=Low(a) to High(a)-1 do begin
        if a[i] > a[i+1] then begin
            result:=false;
            Break;
        end;
    end;
end;
{.....}
begin
// !!!      Randomize;  !!!

    for i:=1 to Nbig do
        a[i]:=Random(Nbig);

    ShowMessage( BoolToStr(IsSorted(a),true));

    time0:=Now;

    BubbleSort(a);

    time1:=Now;

    ShowMessage( BoolToStr(IsSorted(a),true));
    ShowMessage( TimeToStr(time1-time0));
end;

```

Всё очень просто. Функция Now, если вы с ней ещё не знакомы, возвращает текущее время, а заодно и дату, в формате TDateTime. Формат этот чудесным образом совпадает с форматом single, и по форме и по содержанию, — целая часть числа это дни от 2/30/1899, дробная часть — ну, в общем тоже дни. Если дробная часть равна 0.1, это соответствует 24 часа * 0.1 = 2 часа 24 минуты. Если из одной переменной этого типа вычесть другую, то мы получим разницу во времени, что нам и требуется.

Массив, предназначенный для сортировки, мы заполняем случайными числами функцией Random. Обратите внимание, что вызов процедуры Randomize демонстративно закомментирован. Таким образом мы получаем массив случайным образом, но только один раз. При каждом новом запуске программы он будет одним и тем же. Причина понятна — не совсем спортивно сравнивать разные методы сортировок на разных массивах — вдруг один из них окажется изначально более упорядоченным, чем другой? Функция IsSorted отвечает на одноимённый вопрос — отсортирован ли массив. Обратите внимание на загадочно-нелепый вызов BoolToStr, это не я такое придумал, это так надо.

Константа N, задающая количество элементов сортируемого массива, прошла долгий путь. При значениях 10, 100, 1000 сортировка происходила мгновенно. При значении 10000 программа немного задумалась, но результат всё равно

был ноль секунд. А вот при 100000 программа начала реально работать. Результат — 37 секунд. Теперь нам есть с чем сравнивать.

А сравнивать мы будем с другой, улучшенной и быстрой сортировкой. А откуда мы её возьмём? Прошу обратить самое серьёзное внимание, что слово «быстрая» подразумевает здесь «работающая быстро». Потому что есть ещё и так называемая Быстрая Сортировка (Quick Sort).

Оптимизируем на самом нижнем уровне

Первый путь ускорения программы — то, что называется «ловля блох». Приглядимся к тому, что у нас находится внутри двух вложенных циклов, то есть, повторяется очень большое количество раз. Точнее, кроме этого наша программа почти ничего и не делает. Наверное, если бы нам удалось ускорить этот фрагмент кода, всё заработало бы гораздо быстрее. Посмотрим внимательно:

```
if a[i] > a[i+1] then begin
    tmp:=a[i];
    a[i]:=a[i+1];
    a[i+1]:=tmp;
    gamover:=false;
end;
```

Когда я был молодой и весёлый (и, как поётся в песне, «и голодный и злой») я программировал на Паскале на клоне PDP-1, у которого было честных 64 килобайта оперативной памяти, я уже говорил об этом. Транслятору банально не хватало памяти для работы. Поэтому транслятор Паскаля был двухпроходной. Сначала транслятор переводил исходный паскалевский код в исходный ассемблерный код, затем ассемблер переводил свой код в объектный. Это, конечно, тормозило, требовало написания BAT-файлов для управления процессом, зато предоставляло чудесную возможность посмотреть, как выглядит Паскалевская программа изнутри после трансляции. Машинные команды всё-таки мало кому дано читать, а ассемблер гораздо понятнее.

Так вот, в нашей программе мы три раза обращаемся к элементу массива $a[i]$. Индекс массива i при этом инвариантен. Вы не поняли? Это я показал свою образованность. На протяжении этого куска программы переменная i не меняется, это и называется словом *инвариантно*. А вы представляете сколько машинных команд требуется для расчета индекса, определения адреса в памяти с учетом размера элемента массива и доступа к нему? Конечно, я немного запугиваю читателя, но команд требуется действительно далеко не одна. Хотя один раз из трёх $a[i]$ стоит у нас слева от оператора присваивания, но адрес то у неё/него тот же самый. А именно на определение адреса уходят основные усилия процессора, в этом куске кода, само собой.

//Кстати

Лет приблизительно двадцать назад предметом гордости была программа, написанная на чистом ассемблере, независимо от её прочих качеств. Когда однажды появился вирус, написанный на Турбо Паскале, его автора все заочно презирали.

//конец Кстати

А ещё мы три раза обращаемся к $a[i+1]$. Тут всё ещё печальнее. Ведь на самом деле нам не надо определять адрес элемента $a[i+1]$. Он равен адресу элемента массива $a[i]$, со смещением четыре байта. И не забывайте, если вы заглянете в пункт меню Project\Options и выберете страницу Compiler, то, скорее всего, обнаружите установленным пункт Runtime errors\Range Checking. Смысл его понятен — если вследствие нашей же ошибки индекс массива выходит за его границы, программа генерирует соответствующее сообщение. Это очень полезно, не спорю, но если бы вы узнали, сколько совершенно ненужного бесполезного кода добавляется при каждом обращении к элементу массива! Разумеется, эту опцию надо немедленно отключить.

Наверное, проще было бы просто двигаться по переменной, соответствующей массиву, с помощью увеличения указателя, извлекать из него четырёхбайтовые куски, сравнивать их как плавающие числа и менять местами, командой System.Move. Обратите внимание, я даже не предлагаю написать весь этот кусок на ассемблере, ведь я же очень добрый!

Если мы всё это сделаем, будет ли это правильно? Вряд ли. Программа будет работать быстрее, в этом я уверен (хотя лучше всё-таки проверить). Но у нас уйдёт заметно больше времени на её написание, тут никаких сомнений быть не может. Легко и непринуждённо обращаться с индексами массива может каждый. А возиться со смещениями нравится не всем, даже мне не нравится. А главное, что этот полуассемблерный код не только придётся дольше писать, но и отлаживать его будем гораздо дольше. И нет никакой гарантии, что после отладки всё будет в полном порядке, код такого типа отлаживается трудно.

Но я очень любознательный. Одновременно с этим я очень трудолюбивый, и смастерил вот такую штуковину, обратите внимание, никаких индексов массивов:

```

procedure BubbleSortBad( var a : array of single);
var
    gamover           : boolean;
    p, pNext         : pointer;
    tmp              : single;
    i                : integer;
{.....}
function IncP(      p   : pointer;
                 add : integer) : pointer;
begin
    result:=Pointer(Integer(p)+add);

```

```

end;
{.....}
begin
  repeat
    gamover:=true;
    p:=@a;
    pNext:=IncP( p, 4);

    for i:=Low(a) to High(a)-1 do begin
      if Single(p^) > Single(pNext^) then begin
        tmp:=Single(p^);
        Single(p^):=Single(pNext^);
        Single(pNext^):=tmp;
        gamover:=false;
      end;

      p:=IncP( p, 4);
      pNext:=IncP( pNext, 4);
    end;
  until gamover;
end;

```

Замерим время. Понравился ли вам результат? И мне тоже. Удивил ли он вас? Меня — удивил. Что же, получается, что мои глубокие познания в работе паскалевского транслятора для PDP-1 уже не имеют никакой практической ценности, и теперь всё по другому?

Но, мои маленькие друзья, на этом я не успокоился, и решил оптимизировать дальше. У меня возникло подозрение, что преобразование типов, сочетаясь с оператором присваивания дают какую-то дурно пахнущую смесь. Я решил оптимизировать их тоже — заменить на простой System.Move.

То есть, вот эти три строчки:

```

tmp:=Single(p^);
Single(p^):=Single(pNext^);
Single(pNext^):=tmp;

```

я заменил на вот эти три строчки:

```

System.Move( p^, tmp, 4);
System.Move( pNext^, p^, 4);
System.Move( tmp, pNext^, 4);

```

После чего замерил время. Я, честно говоря, уже был готов к чему угодно, но не к этому.

Я знал, что будет плохо, но не знал, что так скоро © «Кино»

То есть, получается, что я уже вообще ничего не понимаю в трансляторах? Не радуйтесь, и с вами то же будет, и довольно скоро.

В поисках ценной литературы по информационным технологиям я только что прочитал книгу о сортировке и поиске на графах и в массивах. Как называется и кто автор не скажу, из чувства солидарности (с автором). Этот автор доказывает, что эффективную программу сортировки невозможно написать без тщательного отслеживания и сокращения ходов штанги магнитных головок жёсткого диска, но признаётся, что есть засекреченные системы сортировки, но он нам о них не расскажет. Разумеется, мне очень понравилось, что все программы написаны на Турбо Паскале, люблю я это дело. Основная проблема в программах — ограничение объёма памяти в 64К (опять!), поэтому рекомендуется выгрузить среду Турбо Паскаля перед запуском программы. Все программы написаны так, как будто не то что объектного, но и структурного программирования никогда не существовало, в теле программ торчат пупырчатые метки.

Печаль в том, что это не книга для студентов, а все программы совершенно серьёзно рекомендуются для практического применения. А другая печаль, что всё это издано уже в нынешнем тысячелетии, после проблемы двухтысячного года (если кто-то ещё о ней помнит, конечно).

Всю эту гнусность и злобность я написал не для того, чтобы показать какой я умный, на примере автора книги, который совсем наоборот. Просто я только что убедился, что знания и опыт, которые казались мне ценными и нужными, незаметно превратились в мусор и труху. Повторюсь, и с вами то же будет, во всех отношениях. Какая мораль? Вы в чём-то уверены, вы знаете, что лучше и что хуже. Например, выгоднее рассматривать массив как просто область памяти и пользоваться указателями для перемещения по нему, чем работать с индексом массива. Будьте готовы обнаружить, что всё уже давно не так, и транслятор справляется лучше. Вы в совершенстве освоили какую-то технологию, научились преодолевать какие-то ограничения — будьте готовы к тому, что технология эта никому больше не нужна, а ограничений этих давно и близко нет.

Меня привело в гармонию со вселенной только перечитывание книги Кернигана и Плотжера «*Элементы стиля программирования*». Оттуда я снова узнал, что транслятор оптимизирует лучше всех, и что

Запись программы на языке ассемблера является последним средством повышения её быстродействия, к которому прибегают либо слишком рано, либо слишком поздно © я уже сказал, кто

У меня конечно, до ассемблера ещё было далеко, но насчёт «слишком рано» всё верно.

Улучшаем алгоритм

В Delphi реализована Быстрая Сортировка (Quick Sort). Немного технических подробностей. Когда я был маленьким и программировал на отечественном клоне IBM/360, сортировка была реализована в виде процедуры, с точки зрения программиста, а физически в виде отдельной внешней программы. Процедура была как бы сверху, а на вход к ней подавались данные для сортировки. Это не сентиментальные воспоминания, это напоминание о двух подходах к обработке данных вообще и везде. Что первично, программа или данные?

Так вот, в Delphi первичны данные. Сортировка существует не как отдельная процедура, на вход которой задаются данные, а как метод класса TList и его наследников. Чтобы что-то отсортировать, надо это что-то поместить в список — или сами данные, если они не больше четырёх байт, или ссылки на них, и написать процедуру сравнения двух элементов. Процедура передаётся на вход метода сортировки. К моему стыду за Delphi, в соответствующем месте справочной системы у них ошибка, и поступать следует строго наоборот написанному там. Это плохое.

А хорошее — то, что Delphi поставляется вместе с исходными текстами, и мы можем посмотреть, как же там оно всё реализовано. Я примерно догадывался, что я там увижу, потому что я уже смотрел, как реализован в Delphi сам TList, который, напоминая, является классическим списком. Как должны реализовываться списки, нетрудно узнать, заглянув в элементарный учебник по информационным технологиям.

Списки бывают однонаправленные, это когда узел списка хранит собственно данные и указатель на следующий элемент списка, память для которого, естественно, тоже выделена динамически. По такому списку можно продвигаться только в одну сторону, от предыдущего элемента к следующему. Узнать для элемента, кто ему предшествовал, нельзя. Бывают списки двунаправленные, устроенные сложнее, по ним можно двигаться в обе стороны. Всё это не то, чтобы сложно, но для понимания требует некоторого напряжения интеллекта головного мозга.

Так вот, в Delphi решили не заморачиваться и реализовали список в виде огромного массива указателей. Главное, всем легко понятно, даже мне. Так вот, заглянув внутрь метода QuickSort, я несколько не удивился, что внутри никакого ассемблера нет. Кое-где в исходных кодах его очень много, особенно в тех местах, которые относятся к самым базовым операциям работы с объектами. Там, в сортировке, не было не только ассемблера, но и никаких System.Move и тому подобного. Более того, там присутствовала рекурсия, совсем как в учебнике по информатике с описанием алгоритма Быстрой Сортировки.

Так что, про эти глупости мы благополучно забываем и идём другим путем. Как учат нас буквально все наставления по программированию — не занимайтесь мелочной оптимизацией, улучшите алгоритм. Попробуем улучшить алгоритм, а сначала к нему, пузырьковому алгоритму, приглядимся. Наша задача —

отсортировать элементы массива по возрастанию. Алгоритм, напоминаю, очень простой — сравниваем первый элемент со вторым, если они следуют в неправильном порядке, то меняем их местами. Потом делаем то же самое со вторым и третьим элементами. Тривиальное наблюдение — если при первом сравнении мы поменяли первый и второй элементы, то при втором сравнении в роли второго элемента выступает бывший первый.

Отсюда нетрудно заметить, что если самый большой элемент массива находится у нас в самой плохой (удалённой от правильного места) первой позиции, то за всего один проход по массиву он доберётся до своего правильного, последнего места. Если не верите, то проверьте на бумажке. Если же самый маленький элемент массива оказался в самой плохой для него последней позиции, то ему потребуется $N-1$ проходов по массиву, чтобы попасть на законную первую позицию. Здесь N — количество элементов.

Впрочем, всё уже придумано до нас. В улучшенном алгоритме сначала мы идём по массиву от первого элемента до последнего, а затем от последнего элемента до первого. У такого алгоритма уже есть и своё название — Cocktail Sort. При пузырьковой сортировки пузырьки всплывают вверх, а здесь они мечутся туда-сюда в обе стороны, что, видимо, у англоязычных программистов ассоциируется со взбиванием коктейля.

```
procedure CocktailSort ( var a : array of single );  
  var  
    gamover           : boolean;  
    tmp               : single;  
    i                 : integer;  
begin  
  repeat  
    gamover:=true;  
    for i:=Low(a) to High(a)-1 do begin  
      if a[i] > a[i+1] then begin  
        tmp:=a[i];  
        a[i]:=a[i+1];  
        a[i+1]:=tmp;  
        gamover:=false;  
      end;  
    end;  
  
    for i:=High(a) to Low(a)+1 do begin  
      if a[i] < a[i+1] then begin  
        tmp:=a[i];  
        a[i]:=a[i+1];  
        a[i+1]:=tmp;  
        gamover:=false;  
      end;  
    end;  
end;
```

```
until gamover;
```

```
end;
```

Там, откуда я взял алгоритм на псевдокоде, между двумя внутренними циклами была дополнительная проверка. Если за время первого прохода по массиву не было ни одной перестановки элементов, то работа завершалась, не входя во второй цикл. Это один из глупейших способов оптимизации среди когда-либо виденных мною.

Попробуем эту процедуру и замерим время, на том же массиве, разумеется. Интуитивно кажется, что должно работать быстрее, причём ощутимо быстрее, чуть ли не в два раза. Увы, интуиция обманывает.

Есть и другая (не моя) рекомендация по улучшению пузырьковой сортировки. Как уже замечено, после первого прохода по массиву самый большой элемент гарантированно оказывается на своём последнем месте. После второго прохода правильным элементом будет занято предпоследнее место и так далее. Напрашивается мысль — после каждого пройденного цикла уменьшать количество итераций следующего цикла на единицу. Это для натуральной пузырьковой сортировки. Для коктейльного мутанта итерации надо уменьшать и с начала массива. Запрограммируйте, замерьте время, и сделайте выводы самостоятельно. А я даже попробовать не буду.

После всего этого я заглянул в очень полезную и рекомендуемую (мною) для тщательного изучения (вами) книгу Рода Стивенса «*Delphi. Готовые алгоритмы*», в раздел о Пузырьковой Сортировке. Кроме двух только что испробованных (вами) усовершенствований я нашёл там загадочный совет хранить переставляемые на большое расстояние элементы массива во временных переменных. Во первых, мы уже почти это пробовали. Во вторых, а как догадаться заранее, что элемент будет перемещён последовательно на несколько ступеней вверх?

Короче, как сейчас в таких случаях принято говорить, *Один я умный в белом пальто стою красивый*.

Меняем алгоритм

Сделаем наконец с нашей сортировкой именно то, что все авторитеты советовали сделать с самого начала — выбрать другой алгоритм.

Возьмём то, что называется *comb sort* — сортировка расчёской. Она простая, но непонятная. Лично мне пришлось задуматься. А поскольку я человек по характеру скорее недобрый, предлагаю задуматься и вам и запрограммировать алгоритм самому. Но, поскольку во мне есть и что-то общечеловеческое, сразу предлагаю результат.

```
procedure CombSort( var a : array of single);  
  var  
    size                : integer;  
    gap                 : integer;  
    shrink              : single;  
    gamover             : boolean;  
    tmp                 : single;  
    num                 : integer;  
begin  
  size:=Length(a);  
  shrink:=1.3;  
  gap:=size;  
  
  repeat  
    gap:=Trunc( gap/shrink);  
    if gap < 1  
      then gap:=1;  
    gamover:=true;  
    num:=0;  
  
    while (num + gap < size) do begin  
      if a[num] > a[num+gap] then begin  
        tmp:=a[num];  
        a[num]:=a[num+gap];  
        a[num+gap]:=tmp;  
        gamover:=false;  
      end;  
      num:=num+1;  
    end;  
  until (gap=1) and gamover;  
end;
```

Замерьте время. Подумайте. Ещё раз замерьте. Ещё раз подумайте. Хорошо ли это? А хорошо ли, что двадцать предыдущих страниц мы занимались ерундой?

А вообще, как заметил вечно правый Керниган, если какая-то фигня занимает один процент времени от выполнения всей программы, нафига её оптимизировать? То есть, конечно, он использовал немного другие слова, но смысл был тот же.

ГЛАВА 8

Техника программирования, или Это вы обязательно должны знать

Абстрактные рассуждения или а оно нам надо?

В основном эта глава должна была рассказывать об указателях, объяснять указатели, показывать применение указателей. Потому что без этого полноценно программировать нельзя. Но, по мере написания текста, у меня стали возникать кошунственные мысли — а действительно ли указатели так уж необходимы? Может, ну их? Ведь есть же языки программирования без указателей и на некоторых из этих языков даже можно программировать. Поэтому вместо вдалбливания в голову читателя мысль о необходимости указателей я решил немного поразмышлять. Вслух. Сам с собой. На примерах, само собой.

Но всё же двадцать шестой раз повторяю — каждый *настоящий* программист должен понимать указатели! Я специально не говорю — знать указатели, потому что знать там нечего, почти. Указатель — это очень просто. Указатель — это, всего-навсего, переменная, которая указывает на другую переменную. Всё, больше ничего знать не надо. Это даже проще, чем концепция параметров процедуры, которые передаются, с незапамятных времен, или по значению или по ссылке, а теперь ещё и по умолчанию, не говоря уже про вариант `const`. Это даже не безобидные целые и плавающие переменные, которые, несмотря на свою внешнюю безобидность, могут быть знаковыми и беззнаковыми, иметь разную разрядность, да ещё и сопровождаться для плавающих чудесными ошибки округления и приведения (смотри термин «*сдвинуть кучу песка*»).

А вот указатели всегда имеют один и тот же тип. Все указатели, говоря другими словами, абсолютно одинаковы, по крайней мере, для данного конкретного языка программирования, они одинаковы.

А теперь я повторяюсь, где-то я уже эту мысль излагал, но мне кажется в книге, которая так и не была опубликована.

Первое, что должен понять и понимает обязательно каждый программист — понятие переменной. Есть переменная, и есть её значение. Всё. Это понимают если не все, то почти все.

Второй уровень понимания — массив. Есть массивы, есть индекс массива, указывающий на элемент массива, есть значение элемента массива, на который указывает индекс массива. Если будущий программист понял про переменную, то он обычно понимает и про массив и с этого момента считается не будущим программистом, а программистом настоящим. Большинство на этом месте останавливаются в своем развитии и программируют долго и счастливо.

Третий уровень — указатель. Указатель указывает на другую переменную. Эта другая переменная может быть, к примеру, массивом, со своими индексами. Проблема скорее не в том, что появляется ещё один уровень ссылок, проблема в том, что возникают некоторые труднопонимаемые моменты, специфические именно для указателей. Изначально указатель не указывает никуда. Неинициализированная переменная тоже не содержит ничего, но этим «ничего» очень часто оказывается ноль, и, по этой счастливой случайности, всё благополучно работает. Ноль в указателе, напротив, гарантированно рушит всю программу. Неизбежное при работе с указателями выделение и освобождение памяти добавляет остроты.

Есть общее правило — программа должна быть законченной и симметричной, причём на любом этапе её написания. Законченность программы обозначает, что она транслируется без ошибок. Этого мало. Если в программе вызывается конструктор класса, но не вызывается деструктор, то такая программа транслируется, но это очень подозрительная программа. Если программа имеет и конструктор и деструктор, но они вызываются в разных процедурах, и, хуже того, на разных уровнях вложенности — это даже опаснее.

Симметричность означает, что если написал **begin**, тут же напиши **end**, пусть даже между ними пока ничего не будет. Если вызвал конструктор, тут же вызови и деструктор, причём на том же уровне, то есть в той же процедуре. Вас это не устраивает? Именно для вашего случая это не подходит? Скромнее надо быть, девушки! Симметрия превыше всего.

Указатели очень часто эту симметрию нарушают — память выделяется где-то снаружи, созданный объект (в широком смысле) передаётся в объект-контейнер, который и распоряжается дальнейшей судьбой созданного объекта, в частности, освобождает его память.

Кроме того, некоторые сущности, назовем их так, в Дельфи являются на самом деле замаскированными указателями — например, экземпляры классов и строки, которые в глубине души являются псевдообъектами. Не беда, когда взрослые люди упорно передают указатели на некоторую переменную не по значению, а по ссылке, мотивируя это тем, что значение переменной, на которую указывает указатель, будет меняться! Горе, когда они передают по ссылке объект потому, что изменится одно из его полей (свойств)...

Кратко о предмете обсуждения

Указатели — это просто. Но, поскольку некоторые упорно их не понимают, я объясню. Потом ещё раз. И ещё. Я вообще ужасно терпеливый и добрый.

*Жалкие трусы придумали тормоз,
То же скажу о гаишниках в форме...* © ОСП

Указатели, как и файлы, бывают типированные и нетипированные. Типированные — для жалких трусов. Больше о них не вспоминаем.

Уточнение. Технически указатель содержит адрес некоторой другой переменной. Это уточнение означает, что с указателями можно обращаться как с обычными целыми числами. В C++ это просто и незатейливо, в Дельфи совершенно умышленно предприняты все усилия, чтобы максимально затруднить эти манипуляции. Впрочем, написав две-три нехитрых функции, можно обращаться с указателями так же легко и просто, как и с числами целого типа.

// Можно не читать. Длинное

Можно не читать, если вы чётко понимаете что такое указатель, а для других я объясню. Когда я был не просто маленький, а был очень маленький, то я имел счастье программировать для машины M-20 (а может M-220, какая разница). Это была большая, красивая машина, она занимала целую комнату (или этаж). Программировали мы сначала не на каком-то языке программирования, не на ассемблере даже, а чисто в машинных кодах.

Сколько у этой машины было памяти, я уже точно не помню — то ли четыре килобайта, то ли все двенадцать. Числа были сорокапятиразрядные. Википедия утверждает, что числа были плавающие, но это неправда. На самом деле, это был просто набор битов (или бит).

Программа обычно располагалась начиная с ячейки 000. Программы у нас были учебные и маленькие и данные традиционно размещались с адреса 100. Чем хороша такая архитектура, так это тем, что её можно окинуть одним взглядом — вот оно всё, под руками. Да, и обратите внимание, 100 — это не адрес в смысле смещения в байтах от начала памяти, 100 это адрес в смысле номера ячейки. Как я заметил, это существенно упрощает понимание.

Итак, у нас есть ячейка номер 100. В ячейке находится число 200. Если быть точным, то мы не знаем *число* ли там находится. Находится там последовательность битов, она же машинное слово, которая (которое) может быть числом с плавающей точкой, цепочкой символов или целым числом. Ведь мы программировали в машинных кодах, а все эти сущности, целые и плавающие, возникают только на уровне языка программирования. Так что точнее было бы сказать, что в ячейке находится просто 200. Извините за философию, но таково программирование.

Мы можем смотреть на это двести, как на целое число и, в таком его качестве, увеличить его на единицу. Точнее, мы должны поместить в ячейку 101 (число) 1, а потом дать команду сложить числа в двух ячейках как целые и поместить результат в ячейку 100. Если правильно помню, команда выглядела так:

0001 0100 0101 // 0001 – код сложения, комментарии там не было

Но мы можем смотреть на это 200 не как на число, а как на номер ячейки памяти. В этой ячейке находится, разумеется тоже какое-то число, например 2000. От того, что мы считаем это число адресом, ничего, по сути, не меняется. Мы точно так же можем увеличить значение ячейки 100 на единицу, после чего она будет указывать на ячейку памяти 201, в которой, опять-таки к примеру, находится число 2100.

Всё, ушёл плакать.

// конец того что Можно не читать

При объявлении указателя, переменной традиционно даётся имя P:

```
var
  p                : pointer;
```

Чему равен уже объявленный, но ещё не инициализированный указатель? Как правило, нулю, но в нём может быть и мусор. Впрочем, это неважно, ноль в качестве значения указателя — такой же мусор.

Указатель представляет ценность, как правило, только тогда, когда указывает на какую-то область памяти. Эта память может быть распределена динамически, то есть руками программиста, в нужный момент, тогда программист должен подумать и о её, памяти, освобождении (в нужный момент). Другой типичный случай, это когда указатель ссылается на область памяти, выделенную автоматически, то есть при объявлении переменной. Извращения мы здесь не рассматриваем — при желании указатель может указывать почти на что угодно. В качестве особого случая, наверное, следует рассмотреть нетипированные параметры.

Основной метод применения указателей — это динамическое выделение памяти. Когда это бывает нужно, и почему нельзя обойтись автоматически объявленными переменными? Отчасти это правильный вопрос. Работу упростили динамические массивы — то есть массивы, длина которых может меняться в произвольный момент выполнения программы, то есть уже после их объявления. Но только отчасти, с указателями всё-таки нагляднее и, не побоюсь этого слова, проще.

Рядом лежащие вопросы

Везде написано, что так программировать нельзя. Нельзя, потому что неправильно. И действительно, это неправильно. Но придётся.

К работе с указателями примыкает непосредственная работа с памятью. Это бывает надо. Реализована эта работа через процедуру Move. Нередко случается, что в классе, в котором реализуется наш метод, уже есть свой метод Move, в таких случаях к процедуре надо обращаться квалифицированно с указанием имени мо-

дуля — System.Move. Процедура делает то же, что и одноимённая ассемблерная команда. Далее в примерах int имеет тип integer, p — pointer.

Типичные задачи. Выделить память и переслать целое в область памяти и обратно:

```
GetMem( p, 1024);
Move( int, p^, 4);
Move( p^, int, 4);
```

Чуть сложнее — не в исполнении, в понимании.

```
var
  a          : array[1..100] of integer;
  b          : array[1..100] of integer;
begin
  Move( a, b[51], 4*15);
```

Что мы тут делаем? Пересылаем первые пятнадцать элементов массива A в середину массива B. Обратите внимание, что если бы элементы второго массива имели тип single, транслятору было бы всё равно. Более того, транслятору было бы всё равно, даже если бы элементы второго массива имели тип byte, что не очень хорошо и потенциально опасно.

В комплекте идёт процедура FillChar, изначально предназначенная для заполнения строки одним повторяющимся символом. Но используется она почти исключительно для заполнения неважно чего нулями. Пусть sA — **array[1..100] of single**. Заполнить всё нулями с помощью этой процедуры проще, чем написать цикл, особенно, если массив двумерный:

```
FillChar( sA, SizeOf(sA), #0);
```

Можно применять эту процедуру и к записям, и к массивам записей, и к записям, содержащим массивы. Одно условие — надо хорошо понимать, что именно вы делаете.

Подумаем конкретно, на примерах. Первый пример

Подумаем мы о необходимости применения указателей, можно ли обойтись без них и, главное, нужно ли.

Задача первая. Есть файл. Файл здесь и далее — это последовательность байтов, записанных на каком-то реальном носителе, не в оперативной памяти. Надо

в этом файле что-то найти, например, бодрящее слово из трёх букв (**end**). *Синтаксический* поиск по образцу задача интересная сама по себе (посмотрите в книгах, на которые я уже ссылался), поэтому отвлекаться на него не будем, а упростим задачу до безобразия. Давайте найдём первый нулевой байт в файле, то есть узнаем его смещение.

Сначала решим задачу, как если бы единственным доступным программным инструментом у нас был этот самый файл, и прилагающиеся к нему процедуры, то есть в языке никаких других конструкций у нас просто нет.

```

var
  F           : File;
  b           : byte;
  whereZ      : integer;
  i           : integer;
begin
  AssignFile( F, 'gr-01.bmp' );
  ReSet( F, 1);

  whereZ:=-1;

  for i:=0 to FileSize(F)-1 do begin
    BlockRead( F, b, 1);
    if b = 0 then begin
      whereZ:=i;
      Break;
    end;
  end;

  CloseFile(F);

  ShowMessage( 'whereZ = ' + IntToStr(whereZ));
end;
```

Предполагая в читателе знакомство с файлами, хотя бы даже и самое элементарное, вынужден объяснить только о причине начала цикла от нуля, что мне, вообще говоря, совершенно несвойственно. Мы ищем *смещение*, то есть для первого байта в файле оно равно нулю. Поэтому совершенно естественно и цикл организовать с нуля. По этой же причине, если требуемый байт не обнаружен, возвращается не ноль, а минус единица.

Мы могли бы написать программу чуть иначе:

```

Seek(F, i);
BlockRead( F, b, 1);
```

То есть, получив вместо одной строчки две, одна из которых совершенно лишняя. Смысл этого, во-первых, напомнить, как можно перемещаться по файлу в произвольную позицию, а во-вторых, унифицировать нашу программу для сравнения с её дальнейшими модифицированными вариантами.

Теперь главный вопрос — хороша ли наша программа? Вопрос странный — наша программа, бесспорно, работает, и этого достаточно, чтобы заткнуть рот всем критикам. Другой вопрос — а хорошо ли она работает? В нашем случае слово *хорошо* является синонимом слова *быстро*, и вопрос в том, нельзя ли её заставить работать быстрее. Когда вам зададут такой вопрос, вы должны не раздумывая ответить — а оно надо? Насколько важно быстродействие этой процедуры, ведь это не самостоятельная программа, а маленькая окраина чего-то большого. Сколько раз она вообще вызывается, за всё время выполнения основной программы, или за сутки. Велика ли будет экономия, если процедура заработает в десять или в тысячу раз быстрее? Стоит ли этого ускорения труд программиста, или наоборот? Всё, я расстроился.

Если успокоиться и посмотреть холодным взглядом, то очевидно, что программа наша работать будет достаточно быстро. Сейчас, в наше время, и насколько я понимаю, при обращении к файлу он весь засасывается в кэш и все перемещения по нему равнозначны работе с оперативной памятью. Проблемы возникнут если файл *действительно* большой. Раньше было не то. Раньше, когда мы читали файл, то читался только один кластер — составлявший от половины килобайта до нескольких, немногих килобайт — поэтому любое движение по файлу обходилось очень дорого. Не исключаю, что при каких-то особых обстоятельствах это и сейчас так.

Если для нас важно сократить количество обращений к физическому носителю, радикальным способом решения проблемы будет прочитать весь файл в память сразу. Тогда мы будем абсолютно уверены, что обращений к носителю у нас ровно одно, меньше никак нельзя. Решим задачу традиционно, с помощью указателей.

```
var
    F                : File;
    p, pWrk         : pointer;
    skoka           : integer;
    b               : byte;
    whereZ          : integer;
    i               : integer;
```

begin

```
AssignFile( F, 'gr-01.bmp' );
ReSet( F, 1);
```

```

whereZ:=-1;

skoka:=FileSize(F);
GetMem( p, skoka);
BlockRead( F, p^, skoka);

pWrk:=p;

for i:=0 to skoka-1 do begin
  System.Move( pWrk^, b, 1);
  if b = 0 then begin
    whereZ:=i;
    Break;
  end;
  pWrk:=Pointer( Integer(pWrk) + 1);
end;

FreeMem( p, skoka);

CloseFile(F);
ShowMessage( 'whereZ = ' + IntToStr(whereZ));
end;

```

Программа написана с некоторым запасом прочности. Обратите внимание на два указателя, вначале имеющих одинаковые значения, то есть указывающих на одно и то же. Затем первый остаётся неизменным для возможности гарантированного и безопасного освобождения памяти, а второй ползёт по блоку данных. Можно было бы обойтись и одним, но с двумя надёжнее. При последнем выполнении цикла второй указатель выходит за пределы допустимого, но это уже не опасно. Если в указателе содержится ерунда, это не страшно, до тех пор, пока мы не попытаемся обратиться к памяти, на которую он (как будто) указывает. Подумайте, можно ли обойтись без байтовой переменной `b`. Испугайтесь вот этого: `pWrk:=Pointer(Integer(pWrk) +1);` Здесь вам не C++, иначе никак. Оформите этот ужас в виде функции.

Два способа мы проверили. Первый простой, но потенциально неэффективный. Второй — эффективный, но невразумительный какой-то. Нельзя ли что-то быстрое, но попроще? Повторим два предыдущих опыта на новом уровне. Сначала используем готовый класс `TFileStream` (файловый поток), затем класс `TMemoryStream` (поток в памяти). Так эти классы называются в Delphi, но и в других приличных языках есть аналоги.

```

var
  FS                : TFileStream;
  b                 : byte;
  whereZ            : integer;

```

```

i                                     : integer;
begin
  FS:=TFileStream.Create( 'gr-01.bmp', fmOpenRead);
  whereZ:=-1;

  for i:=0 to FS.Size-1 do begin
    FS.Read(b,1);
    if b = 0 then begin
      whereZ:=i;
      Break;
    end;
  end;

  FS.Free;
  ShowMessage( 'whereZ = ' + IntToStr(whereZ));
end;
```

Это почти наша ранняя программа с файлами, но оформленная аккуратнее, упакованная, можно сказать. Говоря по научному, реализована первая составляющая ООП — инкапсуляция. Тем не менее, вопросы к быстродействию, они же частота обращения к носителю, остаются теми же, что и для непосредственной работы с файлами. Поэтому предлагаю четвёртый и окончательный вариант. Файловый поток меняем на поток в памяти:

```

var
  MS                                     : TMemoryStream;
  b                                     : byte;
  whereZ                               : integer;
  i                                     : integer;
begin
  MS:=TMemoryStream.Create;
  MS.LoadFromFile( 'gr-01.bmp');
  whereZ:=-1;

  for i:=0 to MS.Size-1 do begin
    MS.Read(b,1);
    if b = 0 then begin
      whereZ:=i;
      Break;
    end;
  end;

  MS.Free;
  ShowMessage( 'whereZ = ' + IntToStr(whereZ));
end;
```

Итак, имеем все преимущества быстрой загрузки в сочетании с предельно простым программным кодом.

Вывод: для этой задачи никаких преимуществ применение указателей не даёт, только усложняет программу. Лучший выбор — использование готового класса `TMemoryStream` или его аналога в других языках.

Внутри средневекового рыцаря — наши опилки © А.Райкин

Если мы заглянем внутрь исходного кода этого класса, то для начала увидим, что загрузку из файла он сам выполнить не в состоянии, а создаёт для этой цели внутренний экземпляр класса `TFileStream`. Никакого намёка на использование указателей незаметно, зато очень даже заметно, что класс является порождённым от класса `TCustomMemoryStream`, написанный вот в таком стиле (цитата):

```
Move(Pointer(Longint(FMemory) + FPosition)^, Buffer, Result);
```

Тип `Longint` как бы намекает, что запрограммировано это во времена самого первого Delphi, ещё шестнадцатирядного. Ничего плохого в этом нет — зачем чинить то, что не сломалось?

Мораль — от указателей всё равно никуда не деться. Впрочем, вот ещё один вариант, с динамическим массивом.

Пятый способ, бессмысленный и бесполезный

Динамический массив — массив объявляемый без указания количества его элементов, указывается только тип. Фактическая длина задаётся процедурой `SetLength`. Что важно — задаётся не только лишь один раз, перед использованием, а сколько угодно. Размер массива может как увеличиваться, так и уменьшаться. Если размер увеличивается, то хвост содержит мусор. Если размер уменьшается, то данные, вообще говоря, теряются навсегда.

Это всё хорошо. Плохо то, что поведение динамических массивов отличается некоторой чудаковатостью. Самое безобидное из этих чудачеств — начальный индекс массива всегда ноль. После окончания работы с массивом хорошим тоном является оператор `A:=nil`, где `A` — наш динамический массив. Оператор `A:=B` для динамических массивов не делает их равными, как для статических. Точнее, он делает их *не только* равными. С этого момента идентификаторы `A` и `B` указывают на один и тот же массив. Любые изменения, внесенные в `A[10]` отражаются в `B[10]` и наоборот. Особо умные уже догадались, что внутри динамических массивов те же указатели.

А теперь обещанный способ. Его единственный педагогический смысл — в демонстрации базовых методов применения динамических массивов:

```

var
  F           : File;
  da         : array of byte;
  skoka      : integer;
  whereZ     : integer;
  i          : integer;

```

```

begin
  // здесь неплохо было бы перейти в текущий каталог

```

```

  AssignFile( F, 'gr-01.bmp' );
  ReSet( F, 1 );
  skoka:=FileSize(F);
  SetLength( da, skoka);
  BlockRead( F, Pointer(da)^, skoka);

```

```

  whereZ:=-1;

```

```

  for i:=0 to skoka-1 do begin
    if da[i] = 0 then begin
      whereZ:=i;
      Break;
    end;
  end;
end;

```

```

  CloseFile(F);
  da:=nil;

```

```

  ShowMessage( 'whereZ = ' + IntToStr(whereZ));

```

```

end;

```

На что обратить внимание? Вот на этот оператор

```

BlockRead( F, Pointer(da)^, skoka);

```

Так делать не рекомендуется, точнее так делать запрещено. Согласно правилам языка мы должны прочитать файл побайтно и присвоить каждый отдельный байт отдельному элементу массива отдельным оператором присваивания. Но мы то ведь знаем, что динамический массив и указатель — почти одно и тоже. Так почему бы этим не воспользоваться? Только никому не говорите.

То же самое, вид сбоку. Второй пример

В реальной жизни, не в учебной, всё точно так же, только таких маленьких программ не бывает. То, что у нас до сих пор было программой, скорее всего станет процедурой (функцией). Вряд ли на вход этой функции передадут файл. Даже если данные поступят из файла, на вход поступит не сам файл (handle), а блок данных, из него прочитанных. Но, скорее всего, никакого файла не будет — данные поступят откуда-то — из глубин космоса, из просторов океана или с ближайшего подключённого к Интернету утюга. Интерфейс этой процедуры будет не выбран её разработчиком, а задан сверху, и будет примерно таким:

```
procedure ProcessP(      p      : pointer;
                        skoka  : integer;
                        var whereZ : integer);
```

Если Тот, Кто Выбирает Интерфейс, принадлежит к множеству эстетов, то, возможно, будет так:

```
procedure ProcessW( var what;
                    skoka  : integer;
                    var whereZ : integer);
```

Это называется нетипированный параметр (untyped parameter). **Var** для нетипированного параметра обязателен. Второй вариант легко приводится к первому:

```
var
  p      : pointer;
begin
  p:=Pointer(what);
  ProcessP( p, skoka, whereZ);
```

Дальнейшая обработка в точности совпадает с вариантом номер два из предыдущего раздела, повторяться не будем. Единственное, не забудьте — не вы память выделяли, не вам и освобождать!

Если вам, опять-таки, из эстетических соображений, больше нравится работать с потоком в памяти (TMemoryStream), то задача может быть сведена к предыдущей. Пусть MS — экземпляр нашего потока. Тогда:

```
MS.SetSize(skoka);
Move( p^, MS.Memory^, skoka);
```

Первый оператор соответствует выделению памяти (GetMem). Память выделяется для свойства-указателя TMemoryStream.Memory в количестве *skoка*. Далее всё понятно.

Можно, при желании, свести решение задачи к применению динамического массива, но что от этого изменится?

Можно ли вообще здесь обойтись без указателей? По моему, нет. Даже если мы сведём задачу к применению потоков, этого сделать нельзя без формального применения аппарата указателей. И, главное, без понимания того, что это, и как оно работает.

Что толкает к использованию указателей в нашем случае? То, что приходят данные и количество их неизвестно, точнее, неизвестно заранее, до того как они придут. Можно сделать вид, что указатели нам не нужны и пользоваться потоками или динамическими массивами. Я и сам так иногда поступаю, имею в виду потоки, не массивы. Нет у меня как-то к ним доверия.

А теперь рассмотрим пример посложнее.

Всё не так и всё разное. Третий пример

Что оставалось неизменным в двух предыдущих задачах? Неизменным оставались входные данные — поток байтов. Точно так же входные данные могли бы быть и потоком четырёхбайтовых целых, восьмибайтовых плавающих или записей, с некоторыми оговорками. Главное то, что каждая наименьшая составляющая входных данных имеет одинаковый заранее известный тип. Условно назовём эту наименьшую составляющую *атом*. Слово это, к сожалению, уже использовано в Windows API — Applications can use two types of atoms: string and integer. The following sections describe these types, имея там чёткий и конкретный смысл. С другой стороны, из моих знакомых, на C++ программирующих, атомами никто не пользуется, хотя кто-то что-то как будто о них слышал. Так что назовём те элементы, на которые мы разбираем наш входной поток данных, атомами.

Всё становится гораздо сложнее, когда атом имеет переменный размер, и эти атомы следуют на вход обработчика в непредсказуемом порядке. Непредсказуемый порядок, из доступных нам средств, лучше всего умеет создавать генератор случайных чисел, точнее псевдослучайных, но какая разница? Решено — на одном конце пищевой цепочки этот самый генератор, отсылающий данные. Данные случайного типа, и, для усложнения, случайного размера. Я имею в виду, что нашу обработку приходящих данных очень упростило бы, если на вход поступали целые-integer, плавающие-single и булевские-boolean. Все эти типы в Delphi имеют одинаковый размер в четыре байта. Нет, мы ситуацию намеренно усложним. А каждое данное будет сопровождаться (спереди, разумеется) сигнатурой с указанием его типа.

Приводить исходный код я не буду, только объясню идею. Первый параметр типированный, это сигнатура. Сигнатура — страшное умное слово. В нашем случае за этим словом скрывается обычное целое число — четырёхбайтовое, а какое же ещё? — указывающее на тип входящих данных. Например, 1 — целое четыре байта, 2 — плавающее восемь байт. Второй параметр нетипированный. Зная сигнатуру, мы знаем, сколько байт необходимо извлечь из второго параметра, который и есть атом.

А куда всё это богатство отправить? Два очевидных варианта.

Первый — массив байт, туда и отправляем. Плюс — очень просто. Минус — извлечение произвольного атома потребует определённых, даже выше среднего, усилий.

Второй вариант — отправляем данные в список. Плюс — это очень просто. Минус — стандартный список не подойдёт, придётся написать свой, порождённый.

Извините, что коротко, иначе никогда не кончим. Тема безграничная.

ГЛАВА 9

Что должен знать программист? Как он должен себя вести?

Нет, не всё

Программист не должен знать всё. Нет, даже не так — программист должен знать не всё. Ум человеческий ограничен, имеет конечную ёмкость и влезает в него мало. Одно влезло, другое вылезло.

Программист должен знать математику, это само собой разумеется. Двадцатого января 1714-го года Пётр Первый издал указа о запрещении дворянам жениться, пока не выучат геометрию. На всякий случай исходный текст:

Послать во все губернии по несколько человек из школ математических, чтобы учить дворянских детей, кроме однодворцов, приказного чина цифири и геометрии, и положить штраф такой, что невольню будет жениться, пока сего выучится. И для того о том к архиереям о сем, дабы памятей венчальных не давали без соизволения тех, которым школы приказаны.

Пётр был гуманист и верил в человечество. Штрафа было явно мало, надо было сразу кастрировать.

Ещё цитата, и, кстати, очень верно подмечено:

Математику уже затем учить надо, что она ум в порядок приводит.

© М.В. Ломоносов

Кроме того, программист обязан уметь логически мыслить. Не в смысле математической логики, а вообще. Почему начальник от вас чего-то хочет?

Что бы вы не ответили, в любом случае, ответ неправильный.

Потому что начальник лично вас ненавидит? Это практически исключено, что в вас такого особенного? Потому что начальник по натуре злой? Бывает. Потому что начальник дурак? Тоже бывает, но злые начальники встречаются чаще, чем начальники-дураки. Начальник, он обычно умнее вас, иначе он не был бы начальником. Или, хотя бы опытнее.

— *А может, Господь Бог не так уж и всемогущ? Просто он долго прожил, и всё знает...* © День сурка

Вот и начальник так же, он дольше прожил и много знает.

О чём я вообще?

В этой главе будет много о моём личном опыте, поэтому главу можно пропустить и немедленно перейти к следующей.

Здесь речь пойдёт о том, про что надо прочитать, что надо выучить, что попробовать на компьютере, чтобы стать хорошим (которому платят деньги) программистом.

Начну с печального — пацаны, для начала выучите русский язык! Эта безграмотность достаёт. Кто-то очень, или слишком, умный сказал, что у американцев цивилизация ориентирована на слово, а у нас на число. У американцев главный предмет — английский язык, а у нас — не русский язык, а математика. Или, по крайней мере, раньше так было. Только не говорите мне, что у нас теперь тоже главный предмет английский язык — всё равно никто его не то чтобы кое-как, но вообще никак не знает. Я, разумеется, говорю о программистах — ведь эта книга написана для них.

И, по слухам, у американцев претендент, написавший резюме с американскими орфографическими ошибками, рискует обломиться в получении вакансии. Не знаю, не проверял, надо спросить. Хотя, как учит нас Задорнов — «они тупые».

Это я к тому, что наша родная математическая ориентированность ни одного раза не может служить оправданием безграмотности в родном русском языке.

И не рассчитывайте, как дети малые, на проверку орфографии в Microsoft Word. Если вы напишете слово «файл» без буквы «й», а слово «лес» наоборот, с двумя буквами «с», Word всё это скушает. Про окончания «-тся» и «-ться» я даже не заикаюсь. Хуже того, предыдущие примеры недокоррекции со стороны Офиса я могу легко объяснить и оправдать. Но двумя абзацами выше я, не нарочно, вместо «для начала», написал «для начла». Word и это скушал за милую душу! Ну не знаю я такого слова, не знаю, а ОН знает.

// Update: до меня наконец дошло, что имело в виду дитя Билла Гейтса. // А до вас?

То есть на компьютерную проверку орфографии рассчитывать не приходится.

Продолжу печальным — пацаны, выучите английский язык! Я сам патриот, мы их всех закопаем, аннигилируем и отправим сажать ягель. Потом. Но программист без английского языка в данный исторический момент — не программист, а недоразумение.

Только не надо отвлекаться на ерунду. Ерундой в данном случае являются и все курсы английского языка. Они не плохие, нет, там вас направляют к вполне разумной и правильной цели — иметь возможность объясняться на английском с официантом, или, наоборот, в качестве официанта с клиентом, или, если очень повезёт, в той же Америке обговорить проект как один программист с другим

программистом. Тут без хоть сколько-то правильного произношения никак, будь вы хоть официант, хотя программист.

Но вам это не нужно, ваша цель другая. Ваша цель — читать справочную систему, читать книжки по языкам программирования (обратите внимание, я даже не говорю — книги по специальности) и ползать по англоязычным сайтам в поисках полезной информации. Вы не поверите, но бывает ещё Интернет на английском языке! Хотя на русском всё, конечно, гораздо интереснее и понятнее.

Суммируем — что нам надо от английского языка:

Произношение — не надо.

Говорить вообще — не надо.

Аудирование (понимание на слух) — не надо.

Читать и понимать — обязательно.

Не так уж и много. Другие языки для программиста излишни и бесполезны. Хотя некоторые уже учат китайский.

Математика

О себе, любимом. Я учился на факультете прикладной математики. В дипломе моём было написано, к моему огорчению, не «прикладной математик», а просто «математик». Хотя сейчас я уже не думаю, что стоило огорчаться по этому поводу, скорее наоборот. Скорее огорчаться надо было тем, кто закончил математический факультет — у них в дипломах через запятую написали позорное «Математик, преподаватель математики».

Образование наше было слегка перекошено — в отличие от нынешних времен, когда учебные часы примерно поровну распределены между математикой и Computer Science, тогда восемьдесят процентов времени приходилось на математику, десять процентов на физкультуру, эстетику и политическую экономию, а остальные проценты преподавали преподаватели, преподавать преподаваемые предметы которым лучше бы не стоило. Они откровенно не понимали, что это, зачем это надо, да и, откровенно говоря, были теми, кого попросили на выход из программирующих компаний за профессиональную непригодность. Кроме того, невозможно было представить, как сейчас, работающего (по специальности) с третьего курса где-то на половине ставки студента. В крайнем случае, на пятом, да и то через так называемый научно-исследовательский сектор Университета. Хотя я провёл три месяца на практике в достаточно солидной фирме, существующей и поныне.

Это я всё к тому, что математике я учился заведомо больше, чем любой нынешний студент прикладной математики. И, обладая бесценным опытом, с вершины горы, могу дать совет, что пригодилось, а что — увы нет. Ещё раз напомню глубоко выстраданную мысль — если вы чего-то не знаете, то вам это никогда и не пригодится, потому что вы даже и не догадаетесь, что вам это нужно. В полной мере это относится и к моему случаю.

Первые два года нас обучали основам математики, потом началась специализация, причём конкретно у нашей группы она почему-то происходила два раза — что-то там реорганизовалось на кафедрах, нас кидало туда-сюда, так что я приобрёл двойной опыт. Надо добавить, что специализация тоже была исключительно в математических дисциплинах, ничего специфически компьютерного.

Об основах, или что там было на первом курсе. Математический анализ. Мне нравилось тогда, и сейчас нравится, люблю время от времени прочитать ещё какой-нибудь нечитанный учебник. Матанализ — вещь абсолютно необходимая, хотя практически почти не использовалась в моей практике. С печалью признаюсь, после окончания Университета я не взял ни одного интеграла, хотя производные встречались. Несмотря на это, нисколько не жалею о потраченном времени — матанализ очень способствует пониманию всего вообще.

Аналитическая геометрия для среднего программиста в среднем необходима. Я имею в виду, что большинству программистов её знание потребуется, в первые же двадцать лет работы. Третьим главным предметом было что-то, как будто называющееся «Высшая алгебра». Мне это никак не давалось, но там хорошо научили, что такое матрица и что с ней можно сделать. К сожалению, для узко специализированного мозга программиста — матрица всего лишь двумерный массив.

Преподавание математической логики на первом курсе мне кажется преждевременным. Кантор и Гипотеза Континуума! Или наоборот, Континуум и Гипотеза Кантора? Или опять я всё перепутал? Ну зачем?! Всё что там было действительно необходимо для дальнейшего — множество, квантор существования и квантор общности. А ещё девушек очень возбуждало, что молодой лектор, по фамилии то ли Кантор, то ли Квантор, то ли вообще Континуум, приходил каждый день в разных носках. Не в смысле в понедельник в одних, а во вторник в других. В смысле на левой ноге жёлтый, а на правой — чёрный.

Ещё был очаровательный, хотя и недолгий, курс под названием «Введение в специальность», не знаю, есть ли он сейчас. Мне кажется, именно сейчас он просто обязан быть. Там учили тому, чему должны были научить в школе, но, как показывал опыт, обычно не научили. По хорошему, именно там и надо было учить основам логики, в лёгкой, упрощённой и доступной форме.

Теория вероятностей — очень полезная вещь. И не потому, что я запомнил из неё много формул — я запомнил мало формул, хотя те, что запомнил, я запомнил очень хорошо. А потому, что я просто на физически-интуитивном уровне начал чувствовать, что вероятно, а что — нет, что бывает, а что — вряд ли. Когда я пришёл работать на свою первую работу, то обнаружил там коллегу, на год или два меня старше. Он окончил Политехнический Институт, ныне, само собой, Университет, по самой престижной специальности АСУ (Автоматизированные Системы Управления).

// Демонстрация эрудиции

А знаете ли вы, что Молотов, сталинский народный комиссар иностранных дел, член политбюро и так далее, в молодости поступил в Санкт-Петербургский Политех-

нический Институт на самый престижный факультет? Но через год понял, что не поймет, и перевёлся на факультет попроще — экономический. А тот, самый престижный и глянцевоый, был судостроительным.

// конец Демонстрации эрудиции

Их (не будущего наркома Молотова, а коллегу) там тоже учили теории вероятностей, в рамках общего курса «Высшая математика». В те времена была популярна лотерея «Спортлото» — сам зачёркиваешь шесть номеров на бланке и кидаешь бланк в специальный ящик. Если угадал все шесть чисел — то очень большие деньги. Числа в диапазоне 01-49. Коллега вёл годами статистику всех выпавших чисел, чтобы точно выявить, на кого в следующий раз ставить. Вёл статистику серьёзно, на компьютере ЕС-1022. Это не тот компьютер, который на столе стоит, это тот, который занимает две очень больших комнаты с кондиционерами. Его ведь учили, что выпадения всех чисел абсолютно равновероятно, поэтому если номер 13 за тринадцать предыдущих лет выпадал меньше других номеров, то ставить нужно на него. В те времена я был глуп, наивен и пытался указывать людям на их ошибки. Но в этом случае у меня хватило ума затихариться и промолчать.

Это я всё к тому, что очень неплохо прочитать хороший и простой учебник по теории вероятностей. Лучше старый, чем новый. Из старого рекомендую учебник Вентцель и популярную книгу — Г.Кимбл «Как правильно пользоваться статистикой».

Кстати, много-много лет спустя я узнал, что в теории вероятностей борются два подхода к этому делу. Так вот то, чему учили нас, теперь осуждается и называется Байесовский подход. Никогда не поздно узнать что-то новенькое.

Физике нас учили всего полгода, о чём я очень сожалею, потому что физика в дальнейшем мне очень понадобилась. Само собой разумеется, что всё преподавание было чисто теоретическим, ни о каких практических экспериментах и речи не было. Как результат, всему, что я знаю из физики, я обязан исключительно средней школе. А физика, повторюсь, мне понадобилась и очень. А экзамен я сдал на «хорошо» и совершенно честно. Это мне в плюс, а Университету в минус.

Зато нас долго учили дифференциальными уравнениями и уравнениями математической физики. Математическая физика к физике имеет отношение примерно такое же, как морские свинки имеют к морю и к свиньям. Не то, что это бесполезно — это полезно, и лишних знаний не бывает, но лучше бы меня обучили хоть одному дополнительному языку программирования, или какой-нибудь модной и новой на тот момент математической дисциплине.

Волею пославшего меня деканата и силою вещей я посвятил часть своего студенческого времени теории игр. Я не могу жаловаться, что время это ушло напрасно и в никуда. Да, можно было бы провести это время с большей пользой. Ещё большую часть моего студенческого времени я посвятил обучению игры на гитаре, от этого пользы не было никакой вообще ввиду отсутствия у меня слуха. Но ведь я не жалуясь!

Нужно ли знать теорию игр? А почему нет, лишних знаний не бывает, но лучше потратить время на что-то другое. Теория графов была немного полезнее, но совсем немного.

Методы оптимизации во всех своих разновидностях — совсем другое дело. Где бы я ни работал, в какой сфере индустрии и науки, неизбежно всплывала на горизонте потребность в знании методов оптимизации. Честно говоря, за всю свою жизнь я программировал только в двух отраслях — МВД и геология, но, поскольку области эти находятся как бы на разных полюсах планеты, можно считать то, что верно для них, верным и для всех промежуточных точек (см. теорема Коши о среднем). И там, и там, потребовались методы оптимизации. В МВД — линейное программирование, в МинГео — нелинейное. Можете, если хотите, делать из этого какие-то многозначительные выводы.

А почему мне так часто приходилось применять методы оптимизации? А потому, что я их знал! Сделайте из этого выводы.

Информатика (Computer Science)

Информатика стоит на третьем месте, а не на первом. Почему? В любом случае, тот, кто программирует, что-то об информатике знает и может быть, даже какие-то книги по ней прочёл, не считая учебников. Здесь я должен честно признаться, что в моё время всё было гораздо хуже. Никаких учебников по информационным технологиям не было вообще, так что никто их и читать не мог. Не было даже учебников по языкам программирования. Допускаю, что где-то в МГУ, МФТИ или в оптико-механическом институте они были, но в нашем скромном провинциальном университете они отсутствовали как класс. То есть книги по языкам программирования были, и много, но не учебники, и не в библиотеке Университета, а в книжном магазине. Всё изучалось по конспектам, которые записывались по лекциям, которые читали уважаемые доктора наук, которые в этом были... Не очень они были. Сам я был очень вдумчивый. Когда я на первой лекции второго курса услышал, что «PL/I — язык оболочка», я над этим задумался и думал приблизительно две-три следующих лекции, не будучи в состоянии воспринимать поступающую информацию. На пятом курсе в качестве приходящих преподавателей появились ребята помоложе и не состоящие в штате университета, но было уже поздно — кого вообще что-то интересует на пятом курсе?

Так вот, если студент хотел что-то прочитать, кроме лекций, он должен был пойти в магазин и за свои деньги купить книгу, настоящую книгу, не учебник. Книги были недорогие и хорошие, это плюс. Минус, что никто не шёл и не покупал, кроме меня, и ещё одного или двух студентов. Соответственно, программирую со всего своего бывшего курса сейчас только я и ещё один или два человека.

А надо ли знать другие языки программирования? То есть, надо ли знать много языков программирования?

А надо ли знать другие человеческие языки, те, на которых разговаривают и на которых пишут? Да, безусловно, надо. Каждый новый выученный язык поднимает уровень вашего интеллекта, или, в моём случае, предотвращает преждевременную деградацию нейронов старческого мозга. И это не говоря о прямой выгоде — вы сможете читать ихние новости и узнавать, что думают о нас плохие американцы. И даже, что думают хорошие.

А надо ли знать другие языки программирования? Нет, не другие. Совсем другие. Понятно, что если вы пишете на Delphi, надо знать C++ и C#. Бывает, что надо знать и Фортран. Но всё это знания необходимые профессионально, можно сказать, меркантильно. Или Ассемблер — для растопыривания пальцев.

А вот *совсем* другие языки? Например Haskell или SmallTalk? Я привожу в пример именно два этих языка, потому что я их немного изучал. Да, авторы компиляторов и сред программирования Сами Себе Злобные Буратины, всё несовместимо со всем — а сколько человек реально пользуются этими языками? Пользуются — значит, зарабатывают на них деньги. Двадцать, или даже тридцать? Хотя, о чём это я? Возможно, я подумаю об этом завтра и напишу об этом книгу. Завтра.

На мой взгляд, надо, обязательно надо. Все мы с возрастом тупеем в области головного мозга. Разница в том, что одни тупеют начиная с восьмидесяти, другие тупеют начиная с тридцати. И чтобы хоть как-то задержать этот неумолимый процесс, человек должен заниматься неприятными вещами. Неприятными для его головного мозга. Заставлять его думать. Заставлять мозг делать то, чего бы он для своего удовольствия делать не стал.

Например, в музыке, прослушать подряд симфонию Хиндемита «Гармония мира» и оперу Шостаковича «Нос». А кто говорил, что будет легко? Просмотреть подряд шесть раз в цикле фильмы «Диллинджер мёртв» и «Человек с киноаппаратом». Многим нравится, если что. Найти в интернетах мои неопубликованные книги и произвести фрейдистский анализ ментального состояния автора. Выучить *другой* язык программирования.

И чем дальше этот язык программирования от вашего основного, тем лучше.

Немного о прочих науках, типа физики и филателии.

О физике я уже говорил, потому что нам её не то, чтобы преподавали, но делали вид, что преподавали. Реальная физика, в моем представлении, это когда летят два кирпича, или два астероида, встретились, и что получилось? Или сжимают, сжимают, сжимают газ. И когда оно рванёт?

Вместо этого физику нам излагали в виде сложных формул. Сложных по понятиям студента-математика второго курса. И кому это было надо? Да я в школе больше узнал из физики, чем в университете. В школе даже были настоящие Опыты!

Об обучении другим наукам и речи даже не шло, например о химии. Хотя нет, вру. Вследствие недостатка аудиторий, иногда занятия у нас проходили в корпусе Химико-Биологического факультета. Вокруг стояли шкафы с заспиртованными гомункулусами, а на шкафах чучела бобров. Так вот ты какая, Химия! Так вот ты какой, Северный Олень!

ГЛАВА 10

Для дома, для семьи

Зачем это надо?

В жизни каждого настоящего мужчины наступает момент, когда чистые носки проще купить © Народная мудрость

В жизни каждого настоящего программиста наступает момент, когда нужную программу проще написать самому. А если не наступает, то, значит, мужчина не настоящий. Или программист не настоящий. Или носки не очень-то и нужны.

А какие бывают нужные программы? Я перебрал в памяти то, что программировал лично для себя, а не ради денег.

Программа для заучивания иностранных слов. Такой программе задаётся в какой-то форме словарь, обычно на одной строке слово на импортном языке и его аналог на русском. Программа поступает с ними в зависимости от выбранной технологии обучения. Например, показывает случайное слово на иностранном, а через десять секунд его перевод. Возможны и более сложные методы дрессировки. Такие программы есть, и наверняка среди них есть хорошие, но хорошие мне не попадались. А те, что попадались, к употреблению были непригодны, я студентом лучше программировал. А студентом я программировал плохо, это я сейчас уже могу признать. Поэтому я запрограммировал свою программу, и вам советую. Если, конечно, вас вообще интересуют иностранные языки.

Если всё-таки они вас интересуют, то продолжаем разговор. Когда-то я прочитал статью в журнале (бумажном) о несложной методике пополнения словарного запаса на иностранном языке. На самом деле методика пополнения самая простая и доисторическая — учить, учить и учить! Элемент новизны заключается в критерии отбора слов для заучивания.

Вы читаете книгу на нужном языке, то есть том языке, который вы не знаете. Вы встречаете незнакомое слово. Вы выписываете его на карточку, вместе с переводом. Потом сортируете карточки по алфавиту, если слово уже есть, откладываем карточку в отдельную пачку, для тщательного заучивания. Если слово есть уже и во второй пачке, то неплохо написать его на стене, можно в туалете. Чем пачкать стены в туалете, проще написать небольшую программу. Я её написал, причём написал я её на языке Turbo Pascal for Windows 1.0! Нет, вам не понять всей глубины взлёта моего подвига. Подойдите к книжному шкафу, или залезьте вороватыми ручонками в Интернет и прочтите вот это:

Н.Рубенкинг «Турбо Паскаль для Windows», М. «Мир», 1994

Neil J/Rubenking “Turbo Pascal for Windows. Techniques and Utilities”, 1992

Если вы прочтёте это краткое двухтомное введение в систему программирования и всё, совершенно всё, поймёте и хорошенько проработаете, то, со временем, возможно, сможете самостоятельно запрограммировать крестики-нолики. Хреноватые, конечно. Но это я отвлёкся.

Ещё из полезного — программа поиска дубликатов файлов. Если вы думаете, что это не надо, так это надо. Если вы думаете, что в Интернетах их много, так их мало и все они не совсем то, что надо. Или за деньги. Критерий поиска может быть очень разным — имя файла, имя плюс размер, плюс дата, плюс даже контрольная сумма. Наступит момент, когда такая программа вам очень понадобится.

Запрограммируйте таймер. Понадобится. Но чтобы не один таймер, а произвольное количество тикающих таймеров параллельно. Не знаю зачем, но понадобится.

Изготовьте генератор неповторяющихся случайных чисел. Например надо получить десять случайных чисел в диапазоне от единицы до трёхсот, но чтобы все разные. Редко, но нужно. Зачем? Я перечитал рукопись этой книги несколько раз и каждый раз находил новые ошибки. Сейчас мне кажется, что уже всё. Я хочу выбрать совершенно случайным образом десять *разных* страниц и проверить их. Если опечаток нет, то действительно уже всё.

То, что я перечислил — это хорошие, нужные, правильные, полезные программы. Но немного скучные. А какие бывают ненужные и бесполезные программы? Что-нибудь такое, бесполезное, но забавное? Но поскольку в любую бочку мёда необходимо добавить ведро дёгтя, то все наши следующие программы будут объединены одной темой — темой работы со строками.

Несложный бредогенератор. Постановка задачи

Название раздела как бы намекает, что бывают и сложные бредогенераторы. О чём вообще речь? Для начала можно ознакомиться вот с этим:

lurkmore.to/Корчеватель,

впрочем, оно много где ещё есть, в несколько разных трактовках. Вкратце:

««Корчеватель: алгоритм типичной унификации точек доступа и избыточности». Статья с таким названием была опубликована на сайте «Журнала научных публикаций аспирантов и докторантов» в августе 2008 года и переведена на русский. Эта статья сгенерирована генератором англоязычного наукообразного бреда SCIGen. Программа SCIGen, генерирующая псевдонаучные тексты, была написана за несколько лет до этого группой студентов Массачусетского техноло-

гического института с целью проверки качества рецензирования на (псевдо) научных конференциях».

На самом деле в оригинале статья называется “Rooter”, что звучит гораздо благороднее, чем нелепый «Корчеватель». О чём была статья? Вот краткое содержание на английском языке:

ABSTRACT

Many physicists would agree that, had it not been for congestion control, the evaluation of web browsers might never have occurred. In fact, few hackers worldwide would disagree with the essential unification of vice-over-IP and public private key pair. In order to solve this riddle, we confirm that SMPs can be made stochastic, cacheable, and interposable.

В целом, безусловно, присутствует смысл, только смысл этот всё время ускользает. И, главное, звучит всё очень убедительно.

А нам зачем это надо? Просто для развлечения. Это же интересно, почему бы и нам не попробовать. Вы набрасываете конспект для бредогенератора и он работает — выдаёт правдоподобный умный текст. Конечно, если нам нужен только один как бы умный текст, то проще написать его самому, но что, если мы хотим эту процедуру поставить на поток? Ведь интересно же, правда?

А как оно будет работать? Очень сложный генератор, нам, конечно, так сразу не написать, цепей Маркова в нашем продукте не будет. Наш генератор будет очень простенький, и очень несложный. Зато и написать его будет очень легко.

Замысел такой — есть входной текст с инструкциями интерпретатору и есть выходной текст с конечным результатом. Наша задача разработать язык инструкции (для входного текста) и запрограммировать интерпретатор.

Входной текст будет состоять из фиксированной части, которая просто переносится в выходной текст, и из вариабельной части, с которой при переносе что-то происходит. Например, если на входе у нас текст

Aaa bbb ccc

то на выходе мы получаем именно его, символ в символ. Это банально и неинтересно. А теперь давайте договоримся, что если несколько слов (во входном тексте) написаны в круглых скобках, то из них в выходной текст случайным образом выбирается только одно. То есть, если на входе у нас:

aaa (b1 b2 b3) ccc

на выходе, с равной вероятностью, может получиться:

aaa b1 ccc

aaa b2 ccc

aaa b3 ccc

Или, переходя на более русский язык, если у нас на входе:

Я помню (чудное милое гадкое сладкое) (мгновенье виденье варенье)

то варианты выходного текста вы должны сгенерировать сами и обдумать результат. Побудьте немного центральным процессором.

Разумеется, скобки могут быть и вложенными. На входе:

Aaa (b1 b2 (b31 b32 b33)) ccc

На выходе — почти что угодно:

Aaa b1 b31 ccc

Aaa b2 b33 ccc

Aaa b1 b33 ccc

Возникает легкомысленный вопрос — разве вышеприведённый текст не равнозначен вот такому:

Aaa (b1 b2 b31 b32 b33) ccc

Ответ — не совсем. В первом варианте, с одной парой скобок, вероятности выбора вариантов следующие

B1 *1/6*

B2 *1/6*

B31 *1/6*

B32 *1/6*

B33 *1/6*

Во втором, более изощрённом случае, когда мы имеем две пары вложенных скобок, вероятности заметно иные:

B1 *1/3*

B2 *1/3*

B31 *1/9*

B32 *1/9*

B33 *1/9*

Почему так, я объяснять не буду, вы ведь и сами понимаете. А если не понимаете, ждите одну из моих следующих книг под названием «*Полное введение в теорию вероятностей на Дельфи за пятнадцать минут, с цветными пошлыми картинками*».

Но это не главная причина задействовать вложенный выбор. Путь у нас есть вот такой текст:

Я вас любил. Любовь ещё (быть может не сможет поможет)

Что будет на выходе? Ответ правильный — полная ерунда. А почему? А потому, что выбирать мы должны не из пяти слов, а из трёх. Разумеется, здесь уже пора заменить обыденное слово «слово» на интеллектуальное «лексема». Слов у нас, действительно пять.

*быть
может
не
сможет
поможет*

А лексем у нас только три

*быть может
не сможет
поможет*

А как объяснить нашему маленькому компилятору, что два слова, разделённых пробелом, на самом деле единое целое, и разбирать это единое целое на части не надо? Можно, как в канонических языках программирования, взять всё это в кавычки. Но, для наших ограниченных целей, подойдёт и вариант попроще.

С кавычками, хоть и немного, но придётся повозиться — потому что их две. У нас уже есть скобки, с которыми тоже придётся немного повозиться, потому что их тоже две. Но скобки, по крайней мере, разные — есть левая, а есть правая. Одна начинается, другая заканчивает, а кавычка — увы, одна. Поэтому решаем проблему некрасиво, но дёшево. Лексемы разделяются пробелами. Если несколько слов принадлежат одной лексеме, то они объединяются символом подчёркивания. То есть, текст, приведённый выше, будет записан теперь вот таким образом:

Я вас любил. Любовь ещё (быть_может не_сможет поможет)

Это всё абсолютно необходимо, и очень полезно. Но при чём здесь вложенные скобки?

Вот опять абстрактный алгебраический пример:

xxx (a_(b c) (d e)) yyy

Что будет, если ликвидировать внутренние скобки заранее? А ничего хорошего. Мы получим следующее:

xxx (a_b c d e) yyy

С одной стороны, мы получаем возможность вфбора в скобках лексемы “с”. Жтого не могло бы быть, если бы мы раскрывалаи скобки последовательно изнутри. С другой стороны, мы не получаем лексемы “а_с”, которая вполне допустима.

А теерь возьмёмся за Михаила Юрьевича, Лермонтова, в смысле, исключительно для примера:

С печалью я гляжу на ((наше ваше в_каше)_ (поколенье чудное_виденье абсолютное_неуменьье_распределять_динамическую_память))

В этом случае уничтожение внутренних скобок *заранее* приведёт к совершенно неправильному результату. Если внутренние скобки не раскрыть заранее, или если бы их не было, то процесс бы пошёл неправильно.

Это формулировка нашей задачи в самом, что ни на есть, минимальном варианте. Потом мы её, формулировку задачи, расширим и усложним, а пока хватит и этого. Разумеется, программу с расширенной функциональностью вы реализуете сами. Единственное, чего всё-таки недостаёт в нашем первоначальном варианте — учета того, что сочинять мы будем чаще не в стихах, а в прозе. Как говорит народная мудрость — «Из песни слова не выкинешь». А вот из сказки, а тем более из статьи в научный журнал, можно выкинуть что угодно. Следствие этого рассуждения таково — как в арифметике необходим ноль, как в математической логике необходимо пустое множество, так и нам необходимо некоторое обозначение отсутствия слова. То есть, получив на вход исходный текст:

Скажи-ка (дядя_Петя Вася АВОТТУТНИЧЕГОНЕНАДО), ведь недаром

На выходе мы бы хотели равновероятно получить:

Скажи-ка дядя Петя, ведь недаром

Скажи-ка Вася, ведь недаром

Скажи-ка, ведь недаром

Напрашивается мысль обозначить отсутствие слова пустой строкой — то есть двумя кавычками. Но две кавычки, как ни крути, это два символа. А два символа, увы, обработать в два раза сложнее, чем один. Поэтому предлагаю и настаиваю на обозначении отсутствия слова одним символом, а каким именно будет этот символ, неважно. Я буду использовать знак равенства, таким образом, предыдущий текст будет выглядеть так:

Скажи-ка (дядя дядя_Петя Вася =), ведь недаром

Заметим (в скобках), что скобки (круглые) с этого момента в нашем исходном тексте неприменимы. Можно было бы заменить для целей нашего синтаксиса

круглые скобки на квадратные, но мне круглые кажутся более наглядными, по сугубо sentimentalным соображениям. Об этом — в следующем разделе.

Бредогенератор. Сначала немного о ЛИСПе

Первое, что приходит на ум — а зачем мы это вообще делаем? Точнее, зачем мы это делаем, понятно — просто от безделья. Но зачем мы это делаем своими собственными руками?

Всё уже украдено до нас! © Операция Ы

Действительно, куда ни плюнь, всё уже украдено до нас. Точнее, всё уже придумано до нас, и запрограммировано тоже.

Давным-давно, точнее в пятьдесят каком-то позднем году, уже был создан язык обработки списков LISP (LISt Processing language). Список, в понимании этого языка — то, что в круглых скобках. Типичная лисповская программа просто кишит и рябит в глазах круглыми скобками.

Вот образчик. Хотя это извлечено из абсурдопедии <http://absurdopedia.wikia.com/wiki/ЛИСП>, но в чём-то они правы.

//Цитата .

Типичный текст на лиспе:

(((((0(((0)))(0))((0))((0))(((0(00(000(0))((0))))))))))

Некоторые отступники пытаются испортить простоту и изящность лиспа символами пробела и переноса строки. Текст на лиспе, написанный отступником:

*((0(((
0 0 ((0 (0)) (0))
(0))
) (0)))*

//конец Цитаты

Приглядевшись под микроскопом, замечаем типичную конструкцию — скобка открывается, имя функции, параметры, скобка закрывается. Любой из параметров может выглядеть точно так же, то есть — скобка открывается... Сразу надо предупредить, что настоящий, не фальшивый ЛИСП не разделяет программу и данные, поступающие на вход программы. Это одно и то же.

Пример. Поскольку нас интересуют только строки, то все примеры будут только со строками. Хотя в чистом ЛИСПе кроме строк ничего нет, да и строк нет, честно говоря.

(StrCat "a" "b" "c")

На выходе

"abc"

Что и понятно, функция StrCat склеивает несколько строк в одну. Нам, для наших целей, это не нужно и неинтересно. Но скобки могут быть и вложенными, а это уже обещает быть более полезным. Например, вот такой программный код:

```
(StrCat "a", "b", (StrCat "c","d"))
```

Результат

"abcd"

Здесь о ЛИСПе пока всё, а дополнительно почитать можно вот что:

Альфред Ахо, Рави Сети, Джеффри Ульман, Моника Лам
«Компиляторы. Принципы, технологии, инструменты» 2008.

Это второе издание, первое вышло ещё в 1988-м, но с тех пор ничего не изменилось. Читайте всё, когда-нибудь пригодится. А ЛИСП я изучал вот по этой книге:

С.С. Лавров, Г.С.Силагадзе «Автоматическая обработка данных. Язык лисп и его реализация», М.»Наука», 1978

//Теоретическое рассуждение

Если поглядеть на нашу задачу немного под другим углом, то мы увидим вполне типичную работу для программиста — простую или сложную, как мы выберем. В случае простой задачи мы кодируем сейчас несложный программируемый калькулятор. В случае сложной — мы создаём компилятор с неведомого, но вместе с тем, несложного, языка.

//конец Теоретического рассуждения и раздела тоже

Бредогенератор. Реализация

Для достижения нашей цели неплохо было бы реализовать заранее несколько простых, но, в то же время, универсальных функций работы со строками. Таких, которые нам понадобятся не только конкретно для этой разработки, но и, скорее

всего, позже. А почему они не написаны раньше разработчиками уважаемых, всемирно известных фирм, как-то — Borland, Microsoft, Твёрдторф? Не знаю, наверное, у них были более другие важные занятия. Функций в языке вроде бы и много, но или все не те, или работают не так. Приходится всё делать самому, вот этими самыми руками. Минимальный набор функций размещён в приложениях, возможно, в процессе разработки какие-то из них окажутся лишними. Книгу я пишу по народному чукотскому принципу — «*Что вижу, то пою*», то есть программы я пишу параллельно с написанием текста главы про эту программу, поэтому время от времени в повествовании образуются зигзаги и петли.

Посмотрите на эти функции. Подумайте. Если они покажутся вам полезными, подумайте ещё раз, возможно, вам понравится. Почему функция подсчёта слов в строке разделителями считает только пробелы? Почему функция устранения повторяющихся пробелов устраняет только пробелы? Нельзя ли обобщить? Да, сегодня это ещё не надо. А завтра будет надо, но уже поздно.

Возвращаемся к задаче. Задача это, напоминаю, не только и не просто учебная, но пишется в основном для нашего развлечения. А это означает, что требования к ней, её функциональности и помехоустойчивости заведомо понижены. Разумеется, как происходит со всеми временными программами, если только вы начнёте использовать её чаще трёх раз в неделю, ваше отношение к её надёжности станет намного жёстче.

Обработка входного текста будет производиться построчно. Для нашего случая это означает, что если левая скобка открывается на второй строке, а соответствующая правая закрывается на следующей третьей строке, то это плохая, негодная скобка. Все скобки должны замыкаться в пределах одной строки. Да, мы упрощаем себе жизнь.

Контроль входных данных у нас будет минимальный. Говоря по-русски — минимальные требования к программе, это чтобы она не выдавала “Access Violation Егг” и тому подобные вызывающие злость сообщения. Тем не менее, надо озаботиться хотя бы тем, что каждой открывающей скобке соответствует закрывающая скобка. Будем считать сумму открывающих скобок (плюс один) и закрывающих (минус один). Если в итоге сумма не равна нулю — то увы. Но этого мало — входной текст “aaa)123(bbb” прекрасно такую проверку пройдёт. Поэтому, состояние «увы» должно немедленно наступить и в случае, если сумма становится отрицательной:

```
function CheckStr(      stroka : string) : boolean;
var
    parCount            : integer;
    i                   : integer;
begin
    result:=true;
    parCount:=0;
```

```

for i:=1 to Length(stroka) do begin
  if stroka[i] = '(' then parCount:=parCount + 1 else
  if stroka[i] = ')' then parCount:=parCount - 1;
  if parCount < 0 then begin
    result:=false;
    Break;
  end;
end;

if parCount <> 0 then result:=false;
end;

```

А теперь формулируем алгоритм в целом на псевдоязыке:

ЦИКЛ ПО ВСЕМ СТРОКАМ

ЦИКЛ ДО ТЕХ ПОР, ПОКА В СТРОКЕ ЕСТЬ ЛЕВАЯ СКОБКА
 ВЫДЕЛИТЬ ПОДСТРОКУ С САМЫМИ ВНЕШНИМИ СКОБКАМИ
 ОБРАБОТАТЬ ПОДСТРОКУ И ПОЛУЧИТЬ РЕЗУЛЬТАТ - **А КАК???**
 ЗАМЕНИТЬ ВЫДЕЛЕННУЮ ПОДСТРОКУ ОБРАБОТАННОЙ

КОНЕЦ ЦИКЛА ПО СКОВКЕ

КОНЕЦ ЦИКЛА ПО ВСЕМ СТРОКАМ

Пояснение по поводу поиска подстроки — имеется в виду поиск пары самых внешних из вложенных скобок. То есть, для строки “aa (bb(cc) dd)” мы должны получить в ответ, в какой именно форме ответ — уточним позднее, что открывающая скобка находится в позиции 4, и вся лексема имеет длину 8. Это раздел программы достаточно специфичен и не универсален, поэтому включать его в нашу маленькую библиотеку обработки строк нет смысла. Включаем её в основной текст программы. Обратите сочувственное внимание на отчаянные вопросительные знаки — я действительно не знаю, а как?

Здесь был исходный текст функции с вот таким интерфейсом:

```

function FindIntParStr(   inS       : string;
                        var posFr, len : integer) : string;

```

но я его прибил по причинам, которые станут понятными далее

Зато я с самого начала написал великолепный и исчерпывающий все потенциальные будущие проблемы набор тестов. Так что все ошибки выявились просто с полпинка, после написания основной программы конечно. На всякий случай, что мы должны протестировать? Какие случаи? Во-первых, когда скобок нет вообще, а в строке или одно слово, или несколько. Во-вторых, когда скобки есть, но не вложенные, как одна пара, так и несколько. Сначала, разумеется, рассмотреть примитивный вариант, когда внутри скобок только одно слово или вообще ничего нет, потом случай, когда перед и после самых внешних скобок есть ещё что-то. И вариант с вложенными скобками. Вот такой набор:

```

abc
abc def
(a)
aaa () bbb
123 (a) 456
(ab) (cd) (ef)
(ab cd)
123 (ab cd)
(ab c def) (12 34)
((ab c def) (12 34))
(ab cd (12 34))

```

И это мы ещё не обращаем внимания на случаи лишних, избыточных пробелов — потому что мы их потом приберём. Попробуйте свой вариант программы, на каком по счёту тесте он сломается? Первая версия моей программы сломалась на последнем тесте, и я этим горжусь.

Однако, мы увлеклись подробностями, и забыли о главном. Откуда вообще возьмётся у нас текст для интерпретации? Возможно из файла, возможно его наберут руками в каком-нибудь поле ввода, предоставленном библиотеками языка программирования. Возможно вообще из космоса телепатическим способом. Это значит, что здесь и сейчас мы должны подумать о тестовой программе, которая берёт на себя обязанность обеспечить требуемый текст, и об интерфейсе главной процедуры, реализующей наш алгоритм.

Тестовая программа будет брать данные из файла. Что до главного интерфейса, то данные будут передаваться через список строк (TStringList), мне так больше нравится. Никакого сужения общности при этом не происходит — список строк легко и непринуждённо заменяется на массив или на то, что ещё в состоянии придумать извращённый ум:

```

var
    inSL          : TStringList;
    fName        : string;
begin
    inSL:=TStringList.Create;

    fName:='test-004.brd';
    if FileExists(fName) then begin
        inSL.LoadFromFile(fName);
        GenerateFrom( inSL);
    end
    else ShowMessage( fName + ` no hai`);

    // сохранить куда-то результат, ведь он нам нужен?
    // здесь мы его куда-то как бы сохраняем
    inSL.Free;
end;

```

Проверку на присутствие файла я добавил только после нескольких ошибок в написании его, файла, имени. Теперь у нас уже кое-что есть и мы в состоянии частично реализовать наш псевдокод.

```

var
    inS           : string;
    stroka        : string;
    posFr         : integer;
    len           : integer;
    i             : integer;
begin
    for i:=1 to inSL.Count do begin
        inS:=inSL[i-1];
        inS:=CompressSpaces(inS);
        while (Pos('\',inS) <> 0) do begin
            stroka:=FindParStr( inS, posFr, len);

            ОБРАБОТАТЬ ПОДСТРОКУ И ПОЛУЧИТЬ РЕЗУЛЬТАТ - А КАК???

            Delete( inS, posFr, len);
            Insert( stroka, inS, posFr);
        end;
        inSL[i-1]:=inS;
    end;
end;
end;

```

Ранее неупомянутая функция `CompressSpaces` занимается именно тем, о чём говорит её имя — сжимает последовательность пробелов до только одного пробела. Можно и без неё, но после обработки ею текст выглядит немного аккуратнее. Теперь осталась главная мелочь — реализовать функцию, находящую внутри себя скобки, внутри которых тоже скобки и так далее, до тех пор пока внутри скобок совсем не останется. В этот момент наша функция выбирает случайным образом одну из лексем. Если вам кажется, что это подозрительно напоминает рекурсию, то вы правы. Хотя и не хочется, но основная часть реализации алгоритма напрашивается быть рекурсивной. Просматриваем строку, находим скобки. Вызываем процедуру с параметром в виде строки, содержащим самые внешние вложенные скобки. Процедура просматривает строку, если есть скобки, то вызывает рекурсивно сама себя, но уже с параметром первой встреченной подстроки со скобками. Если скобок нет, то процедура выполняет свою основную задачу — случайным образом выбирает одно слово из нескольких, содержащихся в скобках.

Всё отлично, всё должно работать! В чём засада? Засада в том, что те слова в строке, которые находятся снаружи самых внешних скобок имеют особый статус. Они просто копируются, потому, что нам, писателям бредового текста, так удобнее. Поэтому никак не получается обработать первоначальную строку на об-

щих основаниях — если мы передадим её в нашу рекурсивную процедуру, то из неё случайным образом будет извлечено одно слово, а нам этого не надо. Вы знаете лёгкий способ справиться с этой проблемой? Я знаю — написать программу без рекурсии. Она будет простая и незатейливая и сразу заработает. Но я пошёл на принцип и решил домучить рекурсивный вариант. А что если раскрывать скобки начиная не снаружи, а изнутри?

И вдруг мне говорит служанка

Моей гостиницы, гречанка:

«А что, как этот самый мальчик -

Сыночек графа Лудовико?» © Собака на сене

Получается неплохо — извлекаем самую внутреннюю пару скобок, если их несколько, неважно какую именно, нам подойдёт любая. Отбрасываем наружные скобки — вспомните, ведь внутри этой подстроки других скобок по определению нет. Выбираем одну лексему (слово) из нескольких и вставляем на место выкушенной подстроки. Когда мы покончим с самыми внешними скобками, никаких проблем не возникнет, в отличие от предыдущего варианта. Точнее, проблема одна — а зачем нам теперь рекурсия? Вспоминается давний персонаж артиста Геннадия Хазанова. Он (персонаж, не Хазанов) всю жизнь мечтал попасть в метро, но никак не получалось. Максимум куда ему удалось войти — мужской туалет, потому что над входом тоже светила буква «М»... Вот и я, всю жизнь мечтаю написать рекурсивную программу, и не такую, что рекурсия просто нужна в образовательных целях, а такую, что без рекурсии никак не обойтись. И опять облом.

Наш общий алгоритм выглядит теперь так:

ЦИКЛ ПО ВСЕМ СТРОКАМ

ЦИКЛ ДО ТЕХ ПОР, ПОКА В СТРОКЕ ЕСТЬ ЛЕВАЯ СКОБКА

 ВЫДЕЛИТЬ ПОДСТРОКУ С САМЫМИ ВНУТРЕННИМИ СКОБКАМИ

 УДАЛИТЬ В ПОДСТРОКЕ СКОБКУ И ВЫБРАТЬ СЛОВО

 ЗАМЕНИТЬ ВЫДЕЛЕННУЮ ПОДСТРОКУ СЛОВОМ

КОНЕЦ ЦИКЛА ПО СКОБКЕ

КОНЕЦ ЦИКЛА ПО ВСЕМ СТРОКАМ

А вот функция извлечения самой внутренней подстроки внутри скобок. Немного коряво, но зато работает.

```
function FindIntParStr(      inS      : string;
                           var posFr, len : integer) : string;
var
    parCount, maxParCount   : integer;
    i                       : integer;
begin
```

```

posFr:=0;
len:=0;
parCount:=0;
maxParCount:=0;

for i:=1 to Length(inS) do begin
  if inS[i] = '(' then begin
    parCount:=parCount + 1;
    if parCount > maxParCount then begin
      maxParCount:=parCount;
      posFr:=i;
    end;
  end else
  if inS[i] = ')' then begin
    parCount:=parCount - 1;
  end;
end;

if posFr >=1 then begin
  result:=Copy( inS, posFr, Length(inS));
  result:=Copy( result, 1, Pos(')',result) );
  len:=Length(result);
end
else result:=inS;
end;

```

Текст основной процедуры после этого кажется до смешного простым:

```

var
  inS          : string;
  i            : integer;
begin
  Randomize;

  ShowSL(inSL, ,До того'); // закомментировать потом

  for i:=1 to inSL.Count do begin
    inS:=inSL[i-1];

    inS:=CompressSpaces(inS);

    while (Pos('(',inS) <> 0) do begin
      if CheckStr( inS) then begin
        ProcessParString( inS);
      end
      else begin

```

```

        inS:='ERROR !!!' , + inS;
    end;
end;

// ДОРАБОТКА НАПИЛЬНИКОМ
// PostProcessing( inS); - раскомментировать потом

inSL[i-1]:=inS;
end;

ShowSL(inSL, 'После того'); // закомментировать потом
end;
```

PostProcessing — та, ещё не написанная, процедура, наводящая окончательный блеск на текст, полученный в результате. ShowSL — незатейливая, но очень полезная процедура для вывода списка строк через стандартный вывод сообщения.

Бредогенератор. Доработка напильником и планы на будущее

Осталось немного — заменить подчёркивания на пробелы и удалить наш логический ноль, то есть знак равенства. Если в случае с подчёркиваниями с удобством можно воспользоваться функцией из библиотеки Delphi (SysUtils. StringReplace), то для равенства ситуация чуть сложнее — удалить надо не только равенство, но и пробелы вокруг него. А пробелы могут быть не обязательно с обеих сторон но и только слева или только справа (когда?). Обратите внимание на не совсем тривиальный интерфейс предоставленной нам системой функции. Сможете ли вы написать её по памяти, если вас разбудят среди ночи? Мне кажется, если функция несложная, в пределах десяти строк, лучше написать её под себя с правильным интерфейсом. Это будет быстрее, чем заучивать волшебное слово [rfReplaceAll]. Да я знаю, это не слово, а целое множество.

В любом случае, полностью обойтись казёнными или нашими средствами замены символов не удастся и придётся потом вызвать опять-таки нами сочинённую функцию CompressSpaces. Вот результат, вставьте вызов функции туда, куда надо.

```

function PostProcessing( var stroka : string) : string;
begin
    stroka:=StringReplace( stroka, '\_', '\ ', [rfReplaceAll]);
    stroka:=StringReplace( stroka, '\=', '\ ', [rfReplaceAll]);
```

```
stroka:=CompressSpaces( stroka);
end;
```

Поскольку процедура ShowSL для трансляции и работы требует в секции **uses** указать модуль Dialogs, не включая эту процедуру в свою микробиблиотеку, а привожу её здесь. Обратите внимание, что второй параметр, это параметр по умолчанию (default parameter).

```
procedure ShowSL(      SL      : TStringList;
                    caption : string = '');
var
  stroka      : string;
  i           : integer;
begin
  stroka:=caption + #13#10;

  for i:=1 to SL.Count do begin
    stroka:=stroka + SL[i-1];
  end;

  ShowMessage(stroka);
end;
```

В качестве заключительного штриха, для придания уверенности, неплохо было бы проверить программу на всех представленных в предыдущем тексте примерах. Вы проверили? Нет? А ведь это даже не ваша программа, вам её не жалко. А мне жалко, но я проверил — программа работает. Наверное, случайно.

Написав программу, любую, или модуль, любой, обязательно подумайте — как можно расширить его возможности. Для чего? Причин поступить так множество. От вас могут попросить или потребовать реализовать новую возможность и будет лучше, если эта возможность уже реализована. Если вы попытаетесь расширить функциональность, то, почти наверняка, какое-то место программы не выдержит и лопнет.

Сейчас у нас все варианты выбора лексемы имеют одинаковый шанс реализоваться. Можно дать им разную вероятность, хотя стоит ли? Можно ввести переменные, задающие настройку на тему текста — пишем ли мы сейчас о политике, метафизике или о китайской эротике в трактовке Ван Гулика. И вообще, у меня много и других интересных идей.

Искатель тайного смысла. Обсуждение

Это хорошая, годная, бесполезная программа. Идите в Гугль и погуглите по словам “Bible Code”.

Почему я не предлагаю поискать русские слова «Библейский код»? Потому что первая же ссылка будет на русскую Википедию, в который вы найдёте вот такую ахиною:

Библейский код (ивр. כּוּד בַּיְבְרָה), также известный, как код Торы — последовательность букв, якобы существующая в тексте Библии, которая при декодировании показывает предвидения и пророчества.

Вы что-то поняли? Понять можно одно — в тексте Библии есть какой-то секретный код, и этот секретный код содержит много секретного и очень интересного. Английская Википедия в этом отношении значительно более информативна и вразумительна. То есть в русской Википедии примерно три бессмысленных строки, а в английской несколько осмысленных страниц. Так же неплохо заглянуть сюда <http://skepdic.com/bibcode.html>.

Теперь подробнее, с цитатами и примерами и предложением купить книгу. Вот реклама. Кстати, и недорого, и непосредственно от авторов, от библейских первоотцов, в смысле:

Bible Codes 2000

Based on the algorithms developed by pioneering code researcher Yoav Rosenberg (of WRR), this Israeli software gives you the ability to search for codes confidently throughout the Tanakh. Bible Codes 2000 generates reports on finds that include skips, books, chapters, verses and letter locations, as well as letter numbers. Includes Hebrew font. Creates matrices. Dictionary, concordance, name databases, more. Basic system requirements. Runs on PCs with Windows 95, through Vista and Windows 7. **\$49.95 plus \$10 priority shipping direct from Israel.**

Это только один экземпляр, таких программ море.

Теперь посмотрим на примере, пример, кажется, из американской Википедии. Что мы здесь имеем? А имеем мы текст Библии, на английском, естественно, языке. Текст расположен по горизонтали. В каждой следующей строке текст сдвигается на четыре символа налево. При этом четыре символа слева исчезают, а четыре справа появляются. По вертикали мы пытаемся увидеть какие-то слова, которых в тексте Библии нет. То, что мы увидели, выделено полужирным шрифтом. Увидели мы два слова. Roswell — деревушка в США, где потерпела катастрофу летающая тарелка с пупырчатыми человечками из другой галактики. Это было в 1947-м году. UFO — то же самое, что по-русски НЛО (неопознанный летающий объект). Обратите внимание, что UFO мы находим не просто по вертикали, а по вертикали через два символа. Мне кажется, что это нечестно, мы так искать не будем.

S A N D M Y D A U G H T E R S T H O U H
M Y D A U G H T E R S T H O U H A S T N
U G H T E R S T H O U H A S T N O W D O
E R S T H O U H A S T N O W D O N E F O
H O U H A S T N O W D O N E F O O L I S
A S T N O W D O N E F O O L I S H L Y I
O W D O N E F O O L I S H L Y I N S O D
N E F O O L I S H L Y I N S O D O I N G
O L I S H K Y I N S O D O I N G I T I S
H L Y I N S O D O I N G I T I S I N T H

А сейчас ещё раз, но другими словами.

В чём состоит процедура с технической, и даже физической, точки зрения? Список (не в виде книги, а в виде длинного рулона) Библии, Ветхого завета, Торы, или Капитала Карла Маркса, наматывается на деревянный цилиндр. Наматывается аккуратно, и так, чтобы буквы в строках оказывались строго вертикально друг над другом. Затем мы ищем по вертикали интересующее нас слово, например, «Вася». Откуда в Библии Вася? В Библии нет Васи, И даже в Капитале нет Васи, хотя я могу и ошибаться. Но тех слов, что мы ищем по вертикали, и находим, в исходном тексте и нет. Эти слова искусственным и случайным образом составились из букв слов, принадлежащих к разным строкам исходного текста.

Задача упрощается (только для евреев) тем, что в древнееврейском (не путать с нынешним ивритом) больших и маленьких букв не было. Да и гласных не было. Поэтому искать скрытый смысл было гораздо легче.

Пока я всё это писал, в новостях обнаружилось вот такое, обратите внимание на цену:

Президент США Барак Обама скрывает от американцев библейское пророчество, согласно которому Россия нападет на США в январе 2017 года, считает бывший кандидат в президенты от Республиканской партии Герман Кейн

В сообщении уточняется, что Россия использует против США электромагнитную бомбу. Однако американцы смогут спастись, если купят за 49 долларов программу «Пережить конец времен».

Теперь постараюсь продемонстрировать на примере. Программу мы ещё не написали, поэтому придется работать головой. Какой текст взять для опытов? Да какая разница! Но, разумеется, текст должен быть оригинальным, то есть непереводаемым. Ход моей мысли очевиден — мы должны постичь бездны и глубины, таящиеся в подсознании автора, а не како-то жалкого и ничтожного переводчика.

Берём вместо Библии роман в стихах «Евгений Онегин». Тренироваться будем на нём. Он длинный и разнообразный.

Искатель тайного смысла. Постановка задачи

Задача наша такая — тест располагается в прямоугольную матрицу, то есть вот так:

мойдядясамыхчестныхправилкогда
невшуткузанемогонуважаатьсебяз
аставилилучшевыдуматьнемогегоп
римерддругимнауканобожемойкакая
скукасидетьсбольнымиденьночьн
еотходянишагупрочь

Обратите внимание, я изменил шрифт на моноширинный. Он не такой красивый, но идеально соответствует нашей задаче — подумайте, почему. И ещё, было очень сложно набирать этот текст без пробелов. Это первая часть программы, число символов в строке задаётся извне, само собой.

Теперь мы задаём слово, по которому хотим найти откровение в тексте — «Ленин», «Вова», «Вася», «Димон», «масоны», «баракобамы». Слова автоматически приводятся к нижнему регистру, как и сам исходный текст.

Далее в действие вступает главный механизм поиска — он пробегает по всем столбцам, и в каждом столбце, начиная с каждой строки сверху ищет заказанное слово. Если находит, то сообщает об этом и выдаёт контекст, в котором слово было найдено — например, все строки, которые пересекаются с этим словом, ну и ещё по одной строке сверху и снизу. Это на тот случай, если предложение исходного текста оказалось разорванным пополам. Мне кажется, всё честно. Я имею в виду, что на выходе мы получим абсолютно честное предсказание с того света, в смысле — от умершего классика.

А теперь краткое содержание программ с разбиением на процедуры

1. Запросить имя исходного файла и прочитать его.
2. Убедиться, что файл более-менее текстовый, это не так просто как кажется, между прочим.
3. Прочитать весь файл и превратить его в таблицу, попутно выкинув пробелы, переводы строк, и заменив большие буквы на маленькие.
4. Запросить ключевое слово для поиска и произвести собственно поиск.
5. Выдать результат.

Как видите, всё очень просто. Приступаем.

Искатель тайного смысла. Подготовка

Первое. Запросить имя текстового файла и прочитать его. Напрашивается примерно вот такая организация программы

```

var
    Text          : TextFile;
begin
    AssignFile( Text, fName);
    Reset( Text);
    // здесь наша программа засасывает весь файл куда-то
    CloseFile(Text);
end;

```

Хорошо ли это? Кажется, хорошо.

Но данные нам надо будет представить в виде прямоугольной таблицы, она же двумерный массив, ведь работать мы будем с ним. Скорее всего, в каждой строке у нас будет 32 символа — кажется, в этом был изначально некоторый сакральный смысл, а кроме того число самым удачным образом представляет собой степень двойки. Строк у нас для начала будет $16 \cdot 1024$. Почему именно столько? Потому что максимальный размер стека, то есть суммарный объём всех доступных в процедуре переменных и всех параметров процедуры равен одному мегабайту. Его можно легко увеличить, но нам, конечно же, лень Переменные можно объявить глобально — но нам опять-таки лень, ведь мы пишем очень простую программу.

То есть, мы должны прочитать в цикле по строкам наш текстовый файл и немедленно обработать каждую строку — то есть переписать её в нашу таблицу. Естественно, первая же прочитанная строка из файла в строку нашей таблицы или не поместится полностью или будет длиннее, чем строка нашей таблицы. Мы должны будем запомнить последний заполненный символ в строке нашей таблицы, и номер строки, затем прочитать следующую строку из текстового файла, записать её с нужной позиции в строку таблицы и, если не влезло, перенести невлезший остаток на следующую строку. А в чём проблема?

Да, собственно, проблемы нет. Но мозг человеческий настолько ограничен, что заниматься хотя бы и всего двумя делами параллельно — читать файл и респихивать его по таблице — для этого ограниченного мозга слишком сложно. Для моего мозга — точно. Я бы предпочёл сначала прочитать весь файл, а уже затем заняться размещением его по таблице. Не надо параллельно, надо последовательно. Всё-таки, немного проще.

Но что это значит — прочитать весь файл? Прочитать его в какой-то массив? Мы и так читаем его в какой-то массив. В чём разница? Образуется замкнутый круг. Это я медленно подвожу вас к тому, что, как обычно, все проблемы уже возникали до нас. И всё уже запрограммировано до нас. Или почти всё.

Сюрприз — в Delphi есть вполне подходящий для наших целей класс TStringList. Я о нём ещё не говорил? Если вы о нём не знали, или не знали, да ещё и забыли, перечитайте справочную систему.

Следующий очень нехитрый код считывает весь наш текстовый файл — это часть первая. Часть вторая, для демонстрации возможностей класса, выводит его построчно на экран. Подумайте, хорошо ли это, если в тексте несколько тысяч строк. Придите к выводу, что это плохо. Подумайте от том, как можно прервать

в любой момент это увлекательное занятие. Отвергните вариант перейти в среду Delphi и нажать <CTRL/F2>. Придумайте что-то лучше.

```

var
    SL                : TStringList;
    i                 : integer;
begin
    GoHome;

    SL:=TStringList.Create;
    SL.LoadFromFile( 'text.txt');

    for i:=1 to SL.Count do begin
        ShowMessage (SL[i-1]);
    end;

    SL.Free;
end;
```

Итак, весь файл мы прочитали, причём добрый класс TStringList разобрал весь текст по строкам и организовал из этих строк псевдомассив. Далее, лёгким движением рук над клавиатурой, мы приводим все эти строки к нижнему регистру — если все строки и наше слово для поиска будут в одном и том же регистре, то вероятность успешного результата значительно повышается. Попутно отсекаем начальные и конечные пробелы.

Достаточно ли этой подготовки текста? Увы, это зависит от самого текста. Если затолкать внутрь что-то длинное и унылое (по форме), например «Конька-Горбунка», то всё будет неплохо. С «Евгением Онегиным» нас ждут проблемы — роман разбит на строфы, между строфами пустые строки. Строфы пронумерованы римскими цифрами (на самом деле английскими буквами V, X и I). Само собой, беспощадному истреблению подлежат знаки препинания, цифры и... Короче, убить всех, кто не русские буквы в нижнем регистре. Удобное их расположение в таблице кодировок значительно облегчает эту задачу.

В итоге, причёсывать файл будем вот так:

```

procedure BibPrepareText(    SL : TStringList);
var
    stroka                : string;
    i,k                   : integer;
begin
    // перевести в нижний регистр
    for i:=1 to SL.Count do begin
        SL[i-1]:=AnsiLowerCase (SL[i-1]);
    end;
```

```

// убрать пробелы спереди и сзади
for i:=1 to SL.Count do begin
    SL[i-1]:=Trim(SL[i-1]);
end;

// убрать пробелы вообще
for i:=1 to SL.Count do begin
    stroka:=SL[i-1];
    for k:=Length(stroka) downto 1 do begin
        if stroka[k] = ' '
            then Delete( stroka, k, 1);
    end;
    SL[i-1]:=stroka;
end;

// убрать вообще всё, что не маленькие буквы
for i:=1 to SL.Count do begin
    stroka:=SL[i-1];
    for k:=Length(stroka) downto 1 do begin
        if (Ord(stroka[k]) < 224)
            then Delete( stroka, k, 1);
    end;
    SL[i-1]:=stroka;
end;

// убрать пустые строки
for i:=SL.Count downto 1 do begin
    if SL[i-1] = '' then SL.Delete(i-1);
end;

// убрать строки, длина которых меньше десяти символов
for i:=SL.Count downto 1 do begin
    if Length( SL[i-1]) < 10 then SL.Delete(i-1);
end;
end;

```

Убедитесь, что в секции `uses` прописан модуль `SysUtils`, обе функции, `AnsiLowerCase` и `Trim`, содержатся в нём. Почему эти две операции не объединены в один цикл? Это мне подсказывает мой богатый жизненный опыт. Такой вариант работает? Да. Такой вариант работает последовательно, разделяя две операции на два прохода списка? Да. Значит он надёжнее. Хотя чуть-чуть, но надёжнее. Разумеется, для нашей несложной программы можно их и объединить, никакого риска нет. Но если это реальная жизнь и это достаточно сложные операции, лучше пусть будет два прохода по данным, чем один, но двойной сложности. Да, накладные расходы возрастут, но важнее надёжность.

Почему тут вообще что-то повторяется, нельзя ли короче? Можно, но в-первых, здесь отражён поток моих мыслей в процессе создания кода. Во-вторых для наглядности. В третьих, для удобства внесения изменений и отката к предыдущей версии. Поясняю. Мне кажется, что удалять пустые строки надо безусловно. А вот удалять ли строки длиной меньше десяти — вопрос спорный, и особенно спорной является константа десять. Поэтому мне кажется нецелесообразным объединять обе эти манипуляции в один цикл.

Следующим шагом предполагается преобразование нашего уже голого текста в таблицу, то есть в двумерный массив символов. Делать это будем просто и без затей — объявим обычный (не динамический) двумерный массив символов. Массив по длине (строкам) достаточно длинный — чтоб влез весь Онегин, и не слишком большой по ширине (столбцам). Предлагаю вот такой тип:

```
const
  minY = 16;
  maxY = 32;
  maxX = 8*1024;
type
  TChArr = array[1..maxX,1..maxY] of char;
```

Если вам этого мало, увеличьте, заодно увеличьте размер стека — Project\Options\Linker (а вот когда я был маленький, у нас стек был... стыдно признаться, в общем, тоже маленький).

Чтобы продать что-нибудь ненужное, нужно сначала купить что-нибудь ненужное, а у нас денег нет © Трое из Простоквашино

Прежде чем что-то программировать, надо написать программу для тестирования этого чего-то. Но прежде чем написать программу для тестирования чего-то, нам надо в точности знать, что именно мы тестируем. Разорвать это бесконечное коловращение сущностей позволяет примерно тот же механизм, который разрывает цепь циклических ссылок. Сначала мы пишем интерфейс тестируемого модуля, затем программу для его тестирования, а потом и сам тестируемый модуль. В нашем случае на вход у нас обязательно должно подаваться количество символов в строке, а на выход — количество фактически заполненных строк. Можно и не возвращать, определяя количество непробельных строк на лету, но что-то мне подсказывает, что вариант с явным возвращением количества упростит дальнейшую жизнь.

```
procedure BibPrepareTable(      SL : TStringList;
                               numY : integer;
                               var ca : TChArr;
                               var numX : integer);
```

Результат выполнения процедуры (массив `ca`) необходимо как-то просмотреть. Предпочтительнее, чтобы наша тестовая программа имела возможность сохранить в текстовый файл двумерный массив с результатами, выводить его на экран было бы чрезмерно утомительно. Для этого понадобится вспомогательная процедура, далее текст. Обратите внимание, что внутри неё прячется старый знакомый — список строк. Создать объект, заполнить, сохранить в файл и уничтожить объект — проще, чем проделывать всё это руками с текстовым файлом (`TextFile`).

```

procedure BibTableToFile(      fName : string;
                               ca    : TChArr;
                               numX  : integer);

var
    SL          : TStringList;
    i           : integer;

begin
    SL:=TStringList.Create;

    for i:=1 to numX do begin
        SL.Add( ca[i]);
    end;

    SL.SaveToFile(fName);
    SL.Free;

end;

```

А вот так, по моему, должна выглядеть тестовая программа в целом:

```

var
    SL          : TStringList;
    fName       : string;
    ca         : TChArr;
    numX       : integer;

begin
    SL:=TStringList.Create;

    fName:='eo.txt';

    if FileExists( fName) then begin
        SL.LoadFromFile(fName);
        BibPrepareText(SL);
        BibPrepareTable( SL, 10, ca, numX);
        SL.SaveToFile(fName + 'bib');
        BibTableToFile( fName + 'bibtab', ca,numX);
        ShowMessage( 'Ok'#13#10 +
                     '  numX = ' + IntToStr(numX));
    end;

```

```
end
else begin
    ShowMessage( fName + ' not exists');
end;
```

```
SL.Free;
```

```
end;
```

Минимальные комментарии в вопросах и ответах. Первый вопрос — в каком каталоге эта программа будет искать текст Романа в Стихах? Ответ — а кто его знает. Предположения есть, уверенности нет. Неплохо было бы перейти в начале процедуры в тот каталог, в котором находится исполняемый файл программы, или в тот, в котором, предположительно, должны находиться наши текстовые файлы с литературными источниками. Это актуально не только для этой программы, но и всегда, во все случаях. Другой вопрос — почему всё это наше богатство не оформлено в виде класса? Ответ — хотя я очень люблю ООП, но во всём надо знать меру, здесь классы выглядят избыточной сущностью. Третий вопрос — зачем задавать как параметр количество символов в строке? Не проще ли всегда заполнять массив по всей ширине (maxY)? Потому что мы хотим немного сжульничать — если при канонической ширине строки в тридцать два символа мы ничего интересного не найдём, то переключимся на тридцать один символ, повторим поиск, и так далее, до двух символов в строке. Почему не до одного, понятно.

Теперь о теории, практике и методологии тестирования. В нашем тесте количество символов в строке задано равным десяти — потому что так легче считать и проверять. Само собой неплохо бы проверить и на другое число — например три. Но, абсолютно обязательной является и проверка на предельные случаи — только она показывает, как ни парадоксально это звучит, работоспособность нашей функции в случаях стандартных.

Для начала стоит задать требуемое число символов в строке очень большим и посмотреть, что получится. Сто символов подойдёт. Я попробовал. Катастрофы не случилось, что меня, по некоторому размышлению, удивило. Я вспомнил, что ширина массива совсем не 100, а всего 32, поэтому желательна проверка на максимально допустимое значение и принудительная замена на него. Разумеется, если бы мы писали эту программу на продажу, этого было бы недостаточно. Потребовалась бы хоть какая-то, но диагностика, или в виде явного кода возврата — нулевого или нет, или в виде возврата фактического количества символов в строке и его унылой проверки впоследствии.

Следующим номером нашей программы должна быть установка переменной в минимально осмысленное значение — единицу. А значения меньше, опять-таки, должны беспощадно отсекаются. Убедившись, что подготовительный процесс работает, приступаем к самой увлекательной части — поиску тайного смысла.

Искатель тайного смысла. Реализация в первом приближении

Как всегда — сначала определить интерфейс, потом написать тестовую программу, потом напихать в интерфейс начинку. В качестве кандидата предлагается вот такое спорное решение:

```
type
  TResArr = array[1..maxNum,1..2] of integer;

procedure BibSearch(      ca          : TChArr;
                          numX,numY : integer;
                          word       : string;
                          var numOf  : integer;
                          var rs     : TResArr);
```

Что здесь вызывает сомнения? Хорошо ли это — задавать поиск только по одному слову, или лучше было бы сразу заменить этот параметр на динамический массив:

```
words : array of string;
```

Может быть, и лучше, для использования этой процедуры, в плане эффективности и скорости. Но, во-первых мы ещё не знаем, какими будут затраты времени на работу процедуры. Во-вторых, и отлаживать вариант с динамическим массивом будет сложнее.

А как мы, собственно, возвращаем результат? Не очень хорошим способом, наши выходные параметры неочевидны. Хороший параметр в объяснениях не нуждается. Мне же придётся пояснить, что numOf — число найденных вхождений слова, из чего следует, что мы не будем останавливаться, найдя слово в первый раз, но будем упорно продолжать поиск до самого конца текста. Что хуже, мне пришлось объявить специальный, хотя и примитивный, тип для хранения собственно результатов. По длине массива (первому индексу), понятно, будут записаны сами результаты (тавтология), а вот по ширине — строка, в которой начинается слово, и столбец. Два напрашивающихся вопроса — нельзя ли заменить массив на одномерный и хранить в нём записи? И нельзя ли заменить массив на список? Ответ — можно, и даже, наверное, нужно, но мы ведь пишем *очень простую, чисто для себя и для развлечения* программу.

Ещё вопрос по интерфейсу — в классической технологии обнаружения библиейского кода ищется не просто слово, которого нет в оригинальном тексте, но которое образуется в нём путём складывания текста в свиток. Этого мало. Найдя заданное для поиска слово *вася* мы немедленно в его окрестностях ищем лексему, к примеру, *нехороший человек* или что-нибудь посильнее. Поскольку мы не

планируем сколько-нибудь коммерчески успешно продавать нашу программу хотя бы за семьдесят долларов американских денег, приходится с огромным сожалением от этих планов отказаться. Мы ведь не знаем заранее, что там найдётся около *вася* — то ли *нехороший человек*, то ли *плохой танцор*. Программа должна располагать огромным словарём потенциально возможных разумных не то что словосочетаний, что желательного, но хотя бы слов. Об этом придётся забыть. Пока.

Впрочем, основная претензия к интерфейсу в том, что он, как ни банально это звучит, вообще существует. Чем подробнее интерфейс процедуры — тем хуже. В нашем случае интерфейс слишком длинный и громоздкий. У нас целых шесть параметров в списке, это очень много. Мы ведь не фирма Microsoft, чтобы позволять себе такое. Оформление в виде класса всё-таки было бы хорошим решением — количество параметров немедленно сокращается почти до нуля.

Покончив с интерфейсом, задумаемся, что у процедуры, им описанной, будет внутри. Только задумаемся, но программировать пока не будем, потому что сначала надо написать тестовую программу. А писать тестовую программу невозможно без того, чтобы знать, чего ожидать от выполнения нашей процедуры. Так вот, в стандартном каноническом варианте поиска библейского кода слова ищутся сверху вниз, снизу вверх, по правильной диагонали (сорок пять градусов направо и столько же налево) и, что по-моему неспортивно, по наклонной под другими углами. Например, шаг направо по таблице, два шага вверх и так далее. Так можно найти гораздо больше. По горизонтали не ищутся, и то спасибо, по горизонтали по понятным причинам у нас и так всё есть. Мы ограничимся только поиском по вертикали и строго сверху вниз.

Займёмся тестовой программой, изготовим её, расширив предыдущую. Что она должна делать? Всё то же самое, что делала и предыдущая, плюс поиск заказанного слова. Плюс вывод результатов. Для экономии дальше не текст целиком, а только добавления:

```

numOf          : integer;
rs             : TResArr;
word           : string;
{.....}
procedure ShowResult;
  var
    stroka      : string;
    i           : integer;
begin
  stroka:=word + #13#10#13#10;
  stroka:=stroka + `numOf = ` + IntToStr(numOf) + #13#10#13#10;

  for i:=1 to numOf do begin
    stroka:=stroka + IntToStr(rs[i,1]) + ` ` +
IntToStr(rs[i,2]) + #13#10;
  end;

```

```
ShowMessage(stroka);
```

```
end;
```

```
{.....}
```

```
begin
```

```
SL.LoadFromFile(fName);
```

```
BibPrepareText(SL);
```

```
BibPrepareTable(SL, 32, ca, numX);
```

```
SL.SaveToFile(fName + 'bib');
```

```
BibTableToFile(fName + 'bibtav', ca, numX);
```

```
ShowMessage('Ok'#13#10 +
             ' numX = ' + IntToStr(numX));
```

```
BibSearch(ca, numX, 32, 'вася', numOf, rs);
```

```
ShowResult;
```

```
end;
```

Однако нужен и специальный тестовый текстовый файл, трудно тестировать сразу на Евгении Онегине, уж очень он длинный. Каковы наши требования к тестовому файлу? Чтобы файл был небольшой. Чтобы там было несколько (обязательно больше одного) вхождений искомого слова. Чтобы одно из этих вхождений обязательно было в самых последних строках файла — для проверки краевого случая (если в слове четыре буквы, то номера строк N-3, N-2, N-1, N). Чтобы было как бы слово, но не полностью, чтобы проверить на ложные срабатывания. Само собой в отдельном режиме на отдельном файле подлежит проверке случай, когда слова вообще нет.

Ещё желательно, для нашего личного комфорта, чтобы наши слова выделялись на общем фоне файла и чтобы легко можно было посчитать их позиции (строка, столбец). Лучше всего этим требованиям удовлетворяет текст примерно вот такого вида:

```
1234567890
```

```
1в34567890
```

```
1а34567890
```

```
и так далее
```

Сразу видно, что наше сакральное слово «Вася» начинается во второй строке и во втором столбце. Подумайте, хорошо ли это. То, что мы это легко определили, хорошо. Плохо то, что если наша программа путает строки и столбцы, мы этого не узнаем — и так (2,2) и так (2,2). Тем не менее, пусть такое вхождение останется, но другие должны быть разнообразнее в этом отношении.

К сожалению, как бы этого не хотелось, всего сразу иметь нельзя. Наш (ладно, мой) только что сочинённый алгоритм не желает пропускать цифры и удаляет их из строки. После чего строки становятся короче десяти символов и убиваются вообще и все. Надо быть скромнее, девушки.


```

var
    nasli                : boolean;
    len                  : integer;
    i,j,k                : integer;
begin
    word:=Flatten(word);
    len:=Length(word);
    numOf:=0;

    for i:=1 to numX - len+1 do begin
        for j:=1 to numY do begin
            nasli:=true;
            for k:=1 to len do begin
                if ca[i+k-1,j] <> word[k] then begin
                    nasli:=false;
                    Break;
                end;
            end;
        end;

        if nasli then begin
            numOf:=numOf + 1;
            rs[numOf,1]:=i;
            rs[numOf,2]:=j;
        end;
    end;
end;
end;

```

Функция Flatten вызывается из задекларированного в предыдущих разделах модуля UsDtrUtils (означает Полезные Строковые Утилиты, если что). Как-то меня огорчала необходимость писать невразумительное *вся* вместо гордого *Вася*. Не забудьте вписать модуль в секцию **uses**, и, если понадобится, в Project\Options\Directories\Conditionals.

Искатель тайного смысла. Развитие темы

Теперь опробуем наш шедевр, естественно на Евгении Онегине и слове *вся*. Увы, результат отрицательный, во всём Онегине Васи нет. По крайней мере, вертикального Васи, возможно диагональный Вася повёл бы себя достойнее и нашёлся в романе. Поиски Пети дают лучший результат:

Кушать подали

Хоть понятно, мой знакомый Петя примерно это больше всего и любил. А вот и Коля:

Замерло в устах

Невразумительно. В целом, итог небогатый — ноль вхождений для первого слова и по одному вхождению для второго и третьего. Это для стандартной развёртки по 32 символа в строке. Поскольку для нас главное не следование канонам, а результат, то напишем несложную надстройку над нашей нехитрой процедурой.

Надстроечная процедура будет в цикле поочередно формировать таблицу с разным количеством строк, на вход процедуры будет задаваться минимальное и максимальное их, строк, количество. Ограничимся предельными значениями, которые мы уже задали для нашей таблицы. Нетрудно посчитать, что при размере спрессованного Евгения в приблизительно 135К минимально допустимое и более менее круглое число символов в строке будет 16. Если оно станет меньше, то количество строк станет больше, а строки у нас динамически свой размер не меняют. «Война и мир» в нашу схему не влезет, надо доработать программу, заменив обычный массив на динамический, или просто увеличив размер стека. Со стеком проблема та, что это решение ситуативное, временное и безграничное. Уважаемый читатель, если тебе интересно, и хочется развить программу, рекомендую предпочесть динамические массивы, хотя и их и не люблю.

Интерфейс нашей будущей процедуры, сразу назовём её BibMegaSearch, с одной стороны по количеству параметров будет меньше. На вход явно не надо передавать саму таблицу, да и количество строк в ней numX. И то и другое будет вычисляться внутри, в цикле. С другой стороны, если мы не передаём готовую таблицу, то, видимо, должны передать хотя бы список строк SL. Кроме того, надо явно указать от какого и до какого значения строк мы проводим поиск. Или не надо? Если не надо, то внутри надо организовать проверку на минимальное значение количества символов... Как всё запутано.

Тяжела и неказиста жизнь простого программиста © Просто песня

Интерфейс предлагается вот такой:

```
procedure BibMegaSearch(      SL           : TStringList;
                             numYfr,numYto : integer;
                             word          : string;
                             var numOf    : integer;
                             var rs       : TResArr);
```

Реализацию я не буду разяснять по шагам, меня ведь читают ещё начинающие, но уже профессиональные программисты.

```

procedure BibMegaSearch(      SL           : TStringList;
                             numYfr,numYto : integer;
                             word          : string;
                             var numOf    : integer;
                             var rs      : TResArr);

var
    ca           : TChArr;
    numX         : integer;
    oneNumOf     : integer;
    oneRs        : TResArr;
    i,k          : integer;

begin
    if numYfr < minY then numYfr:=minY;
    if numYto > maxY then numYto:=maxY;

    BibPrepareText (SL);

    numOf:=0;

    for i:=numYfr to numYto do begin
        BibPrepareTable( SL, i, ca, numX);
        BibSearch( ca, numX, i, word, oneNumOf, oneRs);

        if numOf+oneNumOf <= maxNum then begin
            for k:=1 to oneNumOf do begin
                rs[numOf+k]:=oneRs[k];
            end;
            numOf:=numOf + oneNumOf;
        end;
    end;
end;

```

Обратите внимание, что тестовую программу переписывать не надо, пока, по крайней мере. Так запустим её. Результат налицо. На имя *Вася*, о котором Евгений Онегин делал вид, что и не слышал о нём, мы получили целых девять ссылок. Это хорошо. Плохо то, что все эти ссылки представлены исключительно номерами строк и столбцов. По этим данным мы даже найти результат не сможем, потому что не знаем, для какой ширины таблицы они выданы. Развитие, как известно, происходит по спирали, и мы выходим на тот же вопрос, который задавали себе и раньше. А не следует ли нам хранить результаты поиска не в двумерном массиве целых чисел, а в одномерном массиве записей? Пожалуй, стоит.

В этой записи, разумеется, должно храниться то, что и так раньше хранилось в массиве — номер строки и номер столбца. Вместе с этим должно храниться и количество символов в строке, оно же количество столбцов. Эти три числа делают вполне однозначным нахождение слова в тексте. Ещё, чтобы не искать по файлу, можно записать туда и саму строку, которую пересекает по вертикали

слово. Разумеется, слово пересекает столько строк, сколько в нем символов, но я сохраню только первую. Сохранить больше было бы не трудно, даже без каких-либо ухищрений. Для слова более чем достаточно было бы сохранять шестнадцать строк, слова длиннее очень редки.

Объявляем тип(ы):

```

type
  TResRec = packed record
    str          : integer;
    sto          : integer;
    numY         : integer;
    stroka       : array[1..maxY] of char;
  end;

  TResArr = array[1..maxNum] of TResRec;

```

Незатейливый массив символов вместо типа **string** использован потому, что массив символов сохранить в поток или файл легко, просто и приятно, а вот для **string** замените все эти слова на прямо противоположные. Теперь осталось немного подкрутить то место, в котором мы запоминаем результаты и ту несложную процедуру, которая их выводит:

```

if nasli then begin
  numOf:=numOf + 1;
  rs[numOf].str:=i;
  rs[numOf].sto:=j;
  rs[numOf].numY:=numY;
  Move( ca[i], rs[numOf].stroka, SizeOf(ca[i]));
end;

```

procedure ShowResult;

var

i : integer;

begin

stroka:=word + #13#10#13#10;

stroka:=stroka + `numOf = ` + IntToStr(numOf) + #13#10#13#10;

for i:=1 **to** numOf **do begin**

stroka:=stroka + IntToStr(rs[i].numY) + ` . `;

stroka:=stroka + IntToStr(rs[i].str) + ` ` +

IntToStr(rs[i].sto);

stroka:=stroka + `>> ` + rs[i].stroka;

stroka:=stroka + #13#10;

end;

```
ShowMessage(stroka);
```

```
end;
```

Обратите внимание на оператор (формально процедуру) Move в первом фрагменте. Подумайте, почему я не мог воспользоваться присваиванием (правильно, по причине несовместимости типов). Подумайте, почему я не переприсвоил строки посимвольно? — Правильно, потому что так, как я написал, короче. Мы ведь учимся писать реальные программы. Главное, чтобы работало. И чтобы потом было проще эту программу изменить.

А теперь наконец запускаем и анализируем результаты, если только такое высокопарное выражение применимо к нашей теме.

Сначала *Вася*

Считаюсь инвалидом — это понятно

Пистолет в пяти сажнях — мрачно

Зовёт кот кошку — это что, какой-то левый Онегин?

Теперь *Петя*

Постепенно жизни холод — тоже понятно

Страхаснеплачемчосуспраспл — я бы на месте Пети поостерёгся

Теперь я в вашей воле — ну и тьфу на тебя

Вот не хочешь — да мне то что

Но кушать подано — да, пожрать многие любят

А что мы всё о пацанах, да о пацанах? Давайте про девчонок.

Таню пробовать не будем, её и по горизонтали слишком много. Попробуем Люсю, мне интересно, по личным причинам. Люси нет. Попробуем Лену. Ужас какой-то, полсотни с лишним вхождений, я даже смотреть не стал. Интересно, какими особенностями русского языка объясняется переизбыток Лен над Люсями. Пусть будет Катя. Сравните. Изучите. Вспомните знакомых Люсь, Лен и Кать. Добавьте поиск по диагонали и в обратном направлении. Много думайте.

Но это всё детские игры для настоящих искателей Библейского Кода. Туда не обращаются за предсказаниями своей личной незначительной судьбы или прогноза успеха у неизвестных истории девушек.

Задания:

Поищите на слова:

Темнейший

Чернейший

БаракОбама

СарайОбама

ЧердакОбама

ВсёСлил

ВсёПропало

КакЖыдь — с вопросительным или восклицательным знаком

Снять все неснятые ограничения в программе

Оформите как класс.

А как будет работать наша программа, если текст будет, к примеру, на иврите?

Иврит — это такой странный язык, где всё наоборот — буквы как бы, настоящие, не иероглифы, но пишутся они справа налево. Как придётся изменить нашу программу? Не отмахивайтесь от этого вопроса, он относится не только к нашей не очень серьёзной программе. Возможно, вы напишите очень серьёзную программу и её придётся то, что называется, локализовать, то есть перевести на местный язык. Обычно это касается только перевода интерфейса программы — меню, поля редактирования, сообщения и прочее. Но даже в этом случае проблемы гарантированы, подумайте какие. Кстати, в Delphi есть встроенный механизм локализации, но он мне показался каким-то недоделанным, так что я пошёл своим путём. В следующей своей книге, предназначенной для маленьких начальников, я уделю этому вопросу отдельную главу.

А пока впечатление от работоспособности нашей программы в таких нетрадиционных условиях складывается не вполне однозначное. С одной стороны, если мы возьмём текст на иврите и сформируем из него нашу таблицу для последующего поиска, то результат будет эквивалентен тому, как если бы мы записали русский текст задом наперёд. На поиск в таблице это не повлияет никак, в смысле — совсем никак. Но будет ли наша программа работать, если текст записан в Unicode? И подумайте об иероглифах, китайских, конечно. Китайский минимум для поддержки компьютерами составляет 6500 иероглифов, расширенный набор добавляет столько же. В сущности, немного. С одной стороны, непосредственно наша программа смысл теряет, однозначно. С другой стороны как-то это всё очень даже в китайских традициях, так что надо идти к успеху.

Расстояние между словами. Введение в тему

А эта задача серьёзная и нужная, хотя и не всем. Есть слово, с одной стороны, условно говор, слева. Назовём его — образец. И есть много слов, с другой стороны, условно говоря, справа. Наша задача — определить, какое из слов справа больше всего похоже на слово слева, то есть на образец. Это очень и очень практическая задача. В элементарном случае пользователь набирает что-то в выпадающем списке, в Delphi — TComboBox, а мы должны догадаться, что такое именно он пытался набрать. Ошибиться не страшно, пользователь может потом вы-

брать правильное слово сам. Вспомните о проверке орфографии в Word и о всех смешных забавных шутках с этим связанных.

В Delphi есть отдалённо связанный с этим набор функций — SoundEx и сопутствующие. Soundex переводит слово в набор цифр с буквой впереди, алгоритм разработан для полицейских нужд Нью-Йорка и ориентирован в первую очередь на исконно американские фамилии. Позже была разработана специальная еврейская версия, но говорить *еврейская* теперь неполиткорректно, поэтому версию переименовали в *восточноевропейскую* — примерно то же самое, но вежливо. Это не совсем то, что нам надо — сравнение расстояния между словами является в такой трактовке совершенно неочевидным.

В сущности поставленная задача сводится к тому, чтобы найти расстояние между двумя словами. Если мы умеем найти расстояние между двумя словами, мы сумеем найти из группы слов ближайшее к заданному.

Сначала неплохо упростить слова, то есть избавиться от избыточной информации. Верхний и нижний регистр совершенно не нужны, это очевидно. Двойные согласные идут лесом, и двойные гласные туда же, хотя они и могут иметь некоторое смысловозначительное значения. Заодно избавляемся от окончаний прилагательных. Список заведомо не полный, расширьте сами. И почему ограничиваться только прилагательными? У нас и другие части речи есть. Расширьте!

Получается нечто вот такое, зато несложное:

```
function Flatten(      stroka : string) : string;
begin
    result:=Trim( AnsiLowerCase(stroka));
end;

function Simplify(    stroka : string) : string;
const
    numOkon = 8;
    okon      : array[1..numOkon] of string =
        ( 'OE', 'АЙ', 'АЯ', 'ЯЯ', 'ИЙ',
          'ОЙ', 'АЕ', 'АЯ' );
var
    len      : integer;
    k        : integer;
begin
    stroka:=Flatten(stroka);
    // повторяющиеся буквы
    for k:=Length(stroka) downto 2 do
        if stroka[k]=stroka[k-1]
            then Delete( stroka, k, 1);

    // окончания нафиг
    len:=Length(stroka);
    for k:=1 to numOkon do
```

```
if (len>2) and stroka[len]=okon[k,1])
  and(stroka[len]=okon[k,2])
  then begin
    Delete( stroka, len-1, 2);
    Break;
  end;
```

```
result:=stroka;
```

```
end;
```

Расстояние между словами. Реализация

Какие требования изначально предъявляются к нашей функции? С математической точки зрения, как любая метрика, наша функция для двух одинаковых слов обязана вернуть значения ноль. И расстояние от первого слова до второго слова должно быть в точности равно расстоянию от второго до первого. А с точки зрения программирования, нас интересует, значение какого типа наша функция должна вернуть. Чтобы не мучиться, пусть будет плавающее. Не так красиво, как целое, но не накладывает никаких ограничений на применяемые вычисления.

Получаем вот такую болванку функции:

```
function WordRawDistance(      word1, word2 : string) : single;
begin
  result:=0;
end;
```

Пока что все требования выполнены — при совпадающих словах результатом будет ноль и функция абсолютно симметрична. С другой стороны, всё же хотелось бы, чтобы при отличающихся словах результат был ненулевым, и чем больше слова отличаются, тем больше ненулевым. Требования расплывчатые, так что и программа будет нечёткая.

Сначала идём по второму слову и ищем его буквы в первом слове. Если буква не найдена, начисляем штраф. Штраф тем больше, чем ближе буква к началу слова. Поскольку функция обязана быть симметричной, добавляем симметричный цикл.

```
for i:=1 to Length(word2) do
  if Pos(word2[i],word1) = 0
    then result:=result + (len2-i+1);
```

```

for i:=1 to Length(word1) do
  if Pos(word1[i],word2) = 0
    then result:=result + (len1-i+1);

```

Как-то этот код уже работает. Проверьте. Расстояние между '12345' и '12340' меньше, чем расстояние между '12345' и '02345'. Это правильно. Однако, если одно слово короче другого, то расстояние между словами меньше, чем если бы слова были бы одинаковой длины, но с разными буквами на конце ('12345' и '123'). Мне кажется это несправедливым. Введём штраф за разную длину. Коэффициент 2 является плодом чистой интуиции и рациональному объяснению не поддаётся.

```

result:=result + Abs(Length(word2) - Length(word1))*2;

```

Уже лучше, но не очень. Слова '12345' и '54321' имеют между собой нулевое расстояние. Это никуда не годится. Накажем за перепутанные символы. Логика не прямолинейна, зато хоть как-то работает.

Вспомогательные процедуры:

```

// считает число вхождений символа до заданной позиции
function CountCharUntilPos( stroka : string;
                           ch      : char;
                           untilPos : integer) : integer;

```

```

  var
    k : integer;

```

```

begin
  result:=0;
  for k:=1 to Min(untilPos,Length(stroka)) do
    if stroka[k]=ch
      then Inc(result);

```

```

end;

```

```

// найти позицию вхождения символа с заданным номером вхождения
function GdeCharWithCount( stroka : string;
                           ch      : char;
                           count   : integer) : integer;

```

```

  var
    nomer : integer;
    k      : integer;

```

```

begin
  result:=0;
  nomer:=0;
  for k:=1 to Length(stroka) do begin
    if stroka[k]=ch
      then Inc(nomer);
    if nomer=count then begin

```

```
        result:=k;  
        Break;  
    end;  
end;  
end;
```

Собственно код:

```
for i:=1 to Length(word2) do begin  
    count:=CountCharUntilPos( word2, word2[i], i);  
    gde:=GdeCharWithCount( word1, word2[i], count);  
    Dec( result, Abs(gde-i)+Length(word2)-i);  
end;
```

Проверьте. Добавьте изящества и красоты. Прикрутите куда-нибудь. Продайте кому-нибудь. Получите за это деньги.

ПРИЛОЖЕНИЯ

Справочник обо всём

Приложение А.

Всё о данных

Из чего состоит программа? Ответов два. Или это программа традиционная, в стиле Турбо Паскаля/Си, что вижу, то пою. Или это программа в стиле Дельфи — реагирующая на события. Сначала кажется, что разница здесь космических масштабов. Но с возрастом и опытом работы разница сокращается.

Сначала константы.

const

```
n = 10;           // целая
fl = 3.14;        // плавающая
s = 'приплыли'; // строковая
```

В Дельфи можно и нужно объявлять типы. В других языках программирования — если можно, значит нужно. Далее пример самого простого случая объявления типа, и вариант сложнее:

type

```
Tint = integer;
```

type

```
TA = array[1..1024] of integer;
```

Слева наш только что придуманный тип, справа стандартный четырёхбайтовый целый тип. Зачем это нужно, поймёте позже, возможно. Кстати, по хорошей доброй традиции все имена типов начинаются с литеры «Т».

Простые переменные. Целые переменные объявляются так:

var

```
int           : integer;
int2,int3,int4 : integer;
```

Волшебное (зарезервированное) слово `var` используется при объявлении переменных всех типов. Объявления отдельных переменных отделяются точкой с

запятой. Целые переменные бывают разные, не только типа `integer`, но это не очень важно. Переменные других типов бывают:

С плавающей точкой — `single`. Бывает плавающие и других типов.

Булевские (логические) — `boolean`.

Строки — `string`

Символы — `char`

Указатели — `pointer`

Целые, как сказано, могут быть и других типов, в зависимости от того, есть ли знак, нет знака, сколько байтов число занимает и так далее. Для 99,9% наших задач можно считать, что нас устроят целые переменные типа `integer` (четыре байта), а все плавающие будут `single` (тоже четыре байта).

Со строками всё запутано. Пусть `n` константа, `const n = 10`. У нас есть:

`Char` один символ, один байт

`Array[1..n] of char` массив из десяти символов, десять байт

`String[n]` строка переменной длины, не более 10 символов, занимает 11 байт. `N` не может превышать 255

`String` строка, очень длинная, почти бесконечная. Технически переменная занимает четыре байта, потому что это указатель. Но чудес не бывает, строка занимает столько байт, сколько в ней символов и ещё несколько на свои тёмные цели. Лучше не задумываться.

Непростые переменные, они же агрегаты.

Массив. Массив целых чисел выглядит так:

var

`intA` : **array**[1..1024] **of** `integer`;

`intA2` : **array**[1..1024,1..16] **of** `integer`;

В квадратных скобках значения, которые может принимать индекс массива — от единицы до 1024. После **of** — тип элемента массива. Думаю, понятно, как это переделать под свои потребности. Вторым пунктом объявлен двумерный массив, то есть матрица размерностью 1024 на 16. В памяти массив (Си и Паскаль) хранится по строкам (в Фортране по столбцам). Это не имеет никакого значения, если вы обращаетесь к массиву через его индексы. Это имеет значение, если вы обращаетесь к массиву, как к области памяти (указатели, `System.Move`, файлы, потоки, поймёте потом). Элементом массива может быть другой агрегат, а массив может быть частью другого агрегата.

Главное отличие массива от других агрегатов — все элементы массива имеют один и тот же тип.

Запись (она же структура) — у неё всё наоборот. Она одна, но все поля её имеют равные типы. Например:

var

`Rec` : **record**

```

    int      : integer
    fl       : single
    s        : string;
end;

```

Здесь мы имеем структуру (запись) состоящую из трех полей — целое, плавающее и строка. Обращение к полям производится так:

```

Rec.int:=42;
Rec.fl:=3,14;
Rec.s:='птица-говорун';

```

Это присовение, чтение абсолютно аналогично. Запись может объединять не только скалярные переменные, но и другие агрегаты, например записи и массивы (но не только их), вот так:

```

var
    bigRec : record
        r      : TRec;
        a      : TIntA;
    end;

```

Что такое TRec и TIntA? Это те самые объявленные нами раньше переменные, но оформленные в качестве типов:

```

TRec = record
    int      : integer
    fl       : single
    S        : string;
end;

```

```

TintA = array[1..1024] of integer;

```

Со вторым типом аналогично. Можно ли обойтись без объявления промежуточных типов, то есть вписать объявление одной записи непосредственно в тушку другой записи? Можно, разумеется.

```

var
    bigRec : record
        r      : record
            int  : integer;
            fl   : single;
            s    : string;
        end;

```

```

    a                : array[1..1024] of integer;
end;
```

Такое объявление кажется мне страшным, корявым и уродливым.

Обращение к полям таких записей выглядит чуть сложнее, но вполне понятно:

```

bigRec.r.int:=99;
bigRec.a[77]:=777;
```

Маленькая техническая деталь. Наши объявления записей, и как переменные непосредственно, и как типы, потенциально опасны. Иными словами, в некоторых, совсем нередких, ситуациях такие объявления ведут к серьёзным ошибкам. Вместо этого, всегда, не раздумывая, пишете в заголовке записи:

```

TRec = packed record
```

Волшебное слово **packed** имеет свой аналог и в C++, но я вам его не скажу. Смысл поймёте позже, да и понимать его совершенно не обязательно, главное — всегда пишете, и у вас не будет проблем.

Есть другие типы — множества. Они вряд ли вам понадобятся. Указатели — о них поговорим отдельно. Файлы — совсем отдельная история, хотя в последнее время лично у меня возникает вопрос — можно ли считать их типом и вообще, какова их роль в разработке программы.

Для агрегатов, например массивов, тоже существует понятие константы.

```

type
  a10 = array[1..10] of integer;
const
  asq : a10 = (1,4,9,16,25,36,49,64,81,100);
var
  a                : a10;
begin
  a:=asq;
```

Обратите внимание на то, что сначала надо объявить соответствующий тип, как и в случае передачи массива в качестве параметра в процедуру. И не запугайтесь, где двоеточие, а где равенство, и где круглые скобки, а где квадратные.

Всё об организации программы на нескольких страницах

Программа состоит из последовательности операторов, из условных операторов и из циклов. Всё это помещается или в программу, или в процедуру, или в функцию, которая может выступать и в роли процедуры, как что разницы никакой нет.

Оператор — то, что отделяется от соседнего оператора точкой с запятой. Как понять, что это именно оператор? Если транслируется, значит, оператор, если не транслируется, то фигня какая-то. Бывают пустые операторы, состоящие только из точки с запятой. Немедленно об этом забудьте.

Комментарий — то, что транслятором игнорируется. Если комментируется до конца строки, то это выглядит так, заимствованно из C++:

```
a:=3; // а зачем, собственно, нам три?
```

Если комментируется часть строки спереди, или несколько строк сразу, то так:
{???} a:=4;

```
{  
    a:=5;  
    a:=6;  
}
```

Тяжёлый случай, это когда надо закоментировать что-то вместе с уже имеющимися внутри комментариями:

```
A:=12;  
(*  
A:=13; // ничего не понимаю!  
A:=14;  
*)
```

Так называемые операторные скобки. То, что внутри них, в отношении последовательности исполнения рассматривается как один оператор.

```
begin  
    //  
end;
```

Условный оператор

```
if x > 0 then begin
    //
end
else begin
    //
end;
```

Секция **else** необязательна. Операторные скобки необязательны, если у вас только один оператор внутри них.

Оператор **case** мы презираем и игнорируем, а также не говорим о том операторе, который нельзя называть.

Циклы бывают трёх видов **for**, **repeat** и **while**. На мой взгляд, разновидностей только две — правильная и неправильная. Вот это правильная:

```
for i:=1 to 10 do
    A[i]:=I;
```

Смысл понятен. Обычно (но не всегда) циклы используются совместно с массивами, или другими индексированными структурами. Переменная цикла (I) Может начинаться не от единицы, и может даже идти в обратную сторону. В отличие от других языков в Delphi нельзя задать шаг — это всегда или +1 или -1. Три примера:

```
for i:=20 to 40 do
for i:=30 downto 20 do
for i:=1 to 10 do begin
    A[i]:=0;
    B[i]:=1;
end;
```

Последний пример демонстрирует случай, когда в теле цикла выполняется более одного оператора. Ещё циклы бывают вложенные, например

```
for i:=1 to 10 do begin
    for j:=1 to 10 do begin
        r.A[I,j]:=i*10 + j;
    end;
end;
```

Индексы вложенных циклов — всегда разные переменные и (в Дельфи) всегда локальные. Напоминаю, вы и только вы отвечаете за корректность индексов массивов. Если вы обратитесь к одиннадцатому элементу массива из десяти эле-

ментов, то вы, как говорили в моей фидошной молодости — ССЗБ (посмотрите в гугле). Ещё о правильных циклах:

```
for i:=1 to 1000000 do
  if a[i] > 0 then begin
    result:=i;
    Break;
  end;
```

Этот фрагмент кода находит первый положительный элемент массива, после чего немедленно прекращает работу. За прекращение отвечает команда Break. Выбор, использовать ли её или нет, лежит исключительно на вашей совести. Есть ещё и другие команды для управления работы циклами изнутри, но ими пользоваться не надо и я вам о них не скажу.

Теперь о неправильных циклах — **while** и **repeat**. Первый цикл ещё туда-сюда, а второй ещё хуже.

```
result:=1;
while (a[result]<=0) and (result<=1000000) do begin
  result:=result + 1;
end;
```

Как нетрудно заметить, этот цикл делает ровно то же самое, что и предыдущий.

Третий вариант:

```
result:=0;
repeat
  result:=result + 1;
until (a[result] > 0) or (result>1000000);
```

Здесь, как нетрудно заметить, все условия и логические операторы вывернуты наизнанку. Кроме того, как всем известно, цикл такого вида всегда выполняется минимум один раз.

Для конкретно этой задачи первый вариант, на мой взгляд, лучший. Бывают случаи, когда лучше цикл **while**. Очень редко вперед выступает цикл **repeat**. Циклы мне эти не нравятся. Хотя, если вам по какой-то причине абсолютно необходим цикл с шагом, не равным единице, или вообще не целым, то придётся использовать один из этих двух вариантов.

Возвращаясь на шаг назад. Break прерывает выполнение цикла, не только цикла **for-end**, но и **while** и **repeat-until**. Это хорошо. Плохо то, что он прерывает выполнение только внутреннего цикла. Для обработки трёхмерного массива никакой пользы от этого оператора не было бы. Тем не менее, для невложенных циклов он очень полезен.

Всё о процедурах и функциях

Как знает каждый образованный человек, в С++ секса нет. И процедур тоже, там только функции. В Дельфи есть и то и другое, плюс к этому, любую функцию можно вызвать как процедуру. Долго думайте, как это выглядит и, главное, нужно ли это. Вообразите вызов функции синуса в качестве процедуры. Почти всё, что говорится дальше, в равной степени относится и к процедурам и к функциям, так что оговаривать каждый раз «процедура или функция» я не буду.

Теперь примеры: функция без параметров, возвращающая плавающее значение, и процедура с тремя параметрами, первый передаётся по значению, второй и третий по ссылке.

```
function Pi : single;
begin
    Result:=3.14158;
end;

procedure Sphere(    r : single);
                    var L : single;
                    var V : single);
begin
    L:=pi*2*R;
    V:=(4/3)*pi*R*R*R;
end;
```

Если параметр передаётся по значению, то он может быть переменной, константой или выражением, в нашем случае, например — X , 12.3, $(fl[i+1]+rec.s)*Sin(Pi+1)$. Если значение такого параметра меняется внутри процедуры, для внешнего мира это изменение будет потеряно. Если параметр передаётся по ссылке, то он может быть только переменной, но, в порядке компенсации, если его значение поменять, то там, снаружи, откуда эту процедуру вызывали, значение изменится тоже.

Те параметры, что с **var**, передаются по ссылке — технически, передаётся адрес переменной. Те параметры, что без **var**, передаются по значению, только в один конец, — технически эти параметры заталкиваются в стек. В Delphi есть и другие типы параметров (**const**, **out**), на мой взгляд — полностью бесполезные. Есть, напротив, очень полезные нетипированные параметры:

```
procedure DoSomething( var some);
begin
    System.Move( some, int,4);
```

Важное — массивы как таковые, без затей, передаваться в качестве параметров не могут, их сначала надо объявить типом, это врождённая особенность Паскаля.

```
procedure ProcessArray( var a : array[1..10] of single); //
НЕЛЬЗЯ!!!
```

```
type // МОЖНО!!! И нужно...
  TA10 = array[1..10] of single;
procedure ProcessArray( var a : TA10);
```

Ещё есть *иногда* полезные параметры по умолчанию. Можно объявить функцию определения площади круга вот так:

```
function CircleSquare(      R      : single;
                           ourPi : single = 3.14);
begin
  result:=ourPi * R*R;
end;
```

Вызвать эту функцию можно двумя способами — с традиционным π (параметр по умолчанию пропущен) и с альтернативным, для военного времени, π — параметр по умолчанию задан явно:

```
sq:=CircleSquare( 10);
sq:=CircleSquare( 10, 4);
```

Процедура, как и массив, может быть объявлена как тип, и после этого, как и массив, может передаваться в другую процедуру в качестве параметра. Это бывает нужно, но очень редко.

Вызов процедуры может быть рекурсивным — когда она вызывает сама себя. Бывает косвенная рекурсия — когда процедура ААА вызывает процедуру ВВВ, а та, в свою очередь, вызывает процедуру ААА. Рекурсия очень редко, но тоже бывает нужна.

Напоминание. Извините, если вам это и так понятно. Если вы передаёте в процедуру в качестве параметра экземпляр класса, и хотите в процедуре изменить значения каких-то свойств класса (или полей, если в вашем языке нет свойств), совершенно *не обязательно* передавать этот параметр как **var**. Можно сформулировать и более общее утверждение — если передаёте в качестве параметра указатель, и хотите изменить состояние области памяти, на которое он указывает, **var** не надо. Если хотите изменить значение самого указателя — то надо, хотя, на мой взгляд, менять в процедуре значение параметра-указателя — странная, неправильная идея.

И ещё, тема, которой раньше придавалось огромное значение, а сейчас как-то всё это стало не очень важным — *область видимости*. Зачем было столько шума из ничего — непонятно. Всё очень просто. Процедуры могут быть объявлены в модуле вот так, на одном уровне:

```
procedure AAA;
begin
end;
procedure BBB;
begin
end;
procedure CCC;
begin
end;
```

А могут вот так, это называется вложенные процедуры:

```
procedure BBB;
  var
    x : single;
  {.....}
  procedure AAA;
    var
      x : single;
  begin
    x:=1.23;
  end;
  {.....}
  procedure CCC;
  begin
    x:=4.56;
  end;
  {.....}
  begin
    x:=7.89;
  end;
```

Что такое область видимости? Элементарно, Ватсон — кстати, а вы знаете, что Конан-Дойль этого не писал?

Когда мы выполняем присваивание в процедуре ВВВ, оно влияет на переменную, объявленную в процедуре ССС. На переменные в процедурах ААА и ССС оно никаким образом повлиять не может. Присваивание в процедуре ААА влияет только на переменную X, в этой же процедуре объявленную, но не затрагивает одноимённую переменную из процедуры ВВВ. Это потому, что область видимости переменной ААА.X *перекрывает* область видимости переменной ВВВ.X. А вот с процедурой ССС всё по другому. Присваивание в ней немедленно меняет

значение переменной ВВВ.Х, по той простой причине, что в самой процедуре ССС переменной с именем Х нет.

Терминология. Переменные ВВВ.Х и ААА.Х являются *локальными* для одноимённых процедур. Переменная ВВВ.Х является *глобальной* для процедуры ССС.

Связанный с этой темой технический нюанс — переменные цикла обязаны быть локальными переменными, то есть их обязательно надо объявить именно в той процедуре, где они используются.

Я говорил уже об операторе Break. Он производит немедленный выход из исполняемого цикла. Для процедур есть похожий оператор Exit, он заставляет выйти из исполняемой процедуры. Точно так же, как Break выходит только из самого внутреннего цикла в тот, что снаружи его, оператор Exit выводит из внутренней процедуры в процедуру, её вмещающую.

Теперь о перезагрузке интерфейсов процедур. Когда-то давно я программировал для Delphi, которое называлось именно так, без номера версии, потому что было первым. Там не было, по паскалевской традиции, функции минимума, и я написал свою, для плавающих чисел. Потом я написал свою, но для целых чисел. Интерфейсы у них были примерно вот такие:

```
function MinFloat(      fl1,fl2 : single) : single;
function MinInt(       int1,int2 : integer) : integer;
```

Хорошо ли это? Нет, ничего хорошего здесь нет, просто потому, что это очень некрасиво. Теперь в Delphi появилась казённая процедура для минимума, но проблему можно решить и без неё, вот так:

```
function Min(          fl1,fl2 : single)      : single;   overload;
function Min(          int1,int2 : integer)    : integer;  overload;
```

То же самое, вид сбоку:

```
function NOD(          x1,x2      : integer) : integer;  overload;
function NOD(          x1,x2,x3   : integer) : integer;  overload;
```

Это, как легко догадаться, отыскание Наибольшего Общего Делителя для двух и трёх чисел соответственно. Обратите внимание, что в первом примере отличаются типы параметров, а во втором примере количество параметров. Это не случайно, хоть что-то, но должно отличаться, иначе как транслятор различит процедуры? Обдумайте.

Всё продвинутое программирование на нескольких страницах

Что такое файлы? Вам это было понятно сразу? Я рад. Для меня было трудно понять принципиальное отличие файла от простой переменной, записи или массива. Да, файл — тоже переменная, но, в то же время, он существует реально. Настолько реально, насколько что-то вообще может реально существовать в пределах системного блока.

С файлами можно работать традиционным способом — как с файлами, извините за тавтологию. Имеется в виду работать с файловыми переменными средствами языка. Можно работать как с файлами средствами Windows API, если очень хочется. Когда я был молодой, я целый месяц плотно изучал работу с файлами на языке ассемблера машины IBM/360. С тех пор ничего не изменилось, они (Microsoft) всё тупо украли оттуда для своего API. Иногда, очень редко, это бывает просто необходимо, но прибегать к этому средству следует только тогда, когда это действительно необходимо. Можно общаться с файлами через потоки, через файловый поток, например. Мне больше нравится работа с файлами через, как это называется по-русски, поток в памяти (TMemoryStream в терминологии Delphi). Далее я кратко изложу два способа работы — способ самый древний, исходный, традиционный и работу через поток в памяти.

Традиционный подход разделяется на два варианта, очень напоминающие варианты работы с указателями. Есть типированные файлы и нетипированные, типированные мы игнорируем. Синтаксис далее приводится для Delphi. Если вам *зачем-то* понадобится Турбо Паскаль, в небольших отличиях разберётесь сами. В примере сначала записываем в файл две целых переменных, затем читаем, но сначала вторую переменную, а только потом первую — вот такие мы затейники.

```

var
    F          : file;
    Int1,int2  : integer;
begin
    AssignFile( f, 'путь:\имя.расширение' );
    ReWrite( f, 1 );
    BlockWrite( int1, 4 );
    BlockWrite( int2, 4 );
    CloseFile( f );
.....
begin
    AssignFile( f, 'путь:\имя.расширение' );
    ReSet( f, 1 );
    Seek( f, 4 );
    BlockRead( int2, 4 );

```

```
Seek( f, 0);
BlockRead( int1, 4);
CloseFile(f);
```

Всё интуитивно понятно. Seek перемещает нас по файлу, задавая смещение от его начала. Важная деталь — из середины файла можно не только читать, но и в неё, в середину, писать. Важно помнить, что новые данные при этом не вставляются, раздвигая старые, а переписывают их. Если в файле было 0123456789 и мы со смещения 4 пишем три байта ABC, то в итоге получим 0123ABC789.

На всякий случай, вдруг пригодится, напомним, чем отличается работа с файлами в Турбо Паскале — вместо AssignFile пишем просто Assign, а вместо CloseFile — Close. Подумайте о причинах. Ещё бывают текстовые файлы, объявляемые с типом TextFile (в Паскале, ясное дело, просто Text), но я настоятельно рекомендую пользоваться вместо них чем-нибудь вроде TStringList.

Работа с файлами через поток.

//Как страшно жить!

Только что спросил молодого коллегу, есть ли у них в C++ возможность работы с файловыми потоками, или с потоками в памяти, но не через Windows API, а с помощью уже предоставленных языком классов. Коллега впал в анабиоз, а потом признался, что ему всё равно, потому что он на это положил и перешёл на C# (в котором, как всем известно, всё хорошее полностью списано с Delphi, а плохое — своё).

//ещё не конец, всё-таки...

Далее то же самое, но в переложении для потока:

```
var
    MS                : TMemoryStream;
    Int1,int2         : integer;
```

```
begin
    MS:=TMemoryStream.Create;
    MS.Write(int1,4);
    MS.Write(int2,4);
    MS.SaveToFile( 'd:\aaa.bbb' );
    MS.Free;
```

```
.....
begin
    MS:=TMemoryStream.Create;
    MS.LoadFromFile( 'd:\aaa.bbb' );
    MS.Seek( 0, 4);
    MS.Write(int2,4);
    MS.Seek( 0,0)
    MS.Write(int1,4);
    MS.Free;
```

Единственное, что необходимо объяснить, это первый параметр у метода `Seek`. По правильному, мы должны были бы записать вызов этого метода так:

```
MS.Seek( soFromBeginning, 4);
```

Отсюда, для розуміючих англійську мову, становится совершенно ясным, что первый параметр указывает, от какого места считывается смещение во втором параметре. В нашем случае от начала данных, так всегда и поступайте, это единственный разумный способ.

Именно в этом разделе нашлось время и место для упоминания процедурного типа. Пример очень простой, и потому слегка вымученный и искусственный. Есть процедура, на вход этой процедуры подаётся функция, переводящая одну строку в другую. Например, переводящую число в цифровом виде (123) в сумму прописью (сто двадцать три). Процедура выводит число в цифровом виде и соответствующими словами. Функция выполняет эту операцию для разных языков. Разумеется, выполняет плохо и условно, только для чисел до ста, но это ведь пример.

Вот это сама основная функция, со всем своим модулем, всё очень просто:

```
unit Num2Word; // 04.04.2015
               // 04.04.2015
{-----}
interface
{-----}
  uses
    SysUtils;
  type
    TConvFunc = function( num : integer) : string;

function Num2WordOut ( num : integer;
                      cFunc : TConvFunc) : string;
{-----}
implementation
{-----}
function Num2WordOut ( num : integer;
                      cFunc : TConvFunc) : string;

  var
    word : string;

begin
  word:='';

  if @cFunc <> nil
    then word:=cFunc(num);
```

```

    result:=IntToStr(num) + ' ( ' + word + ' )';
end;
{-----}
end.

```

Сами функции для конкретных языков не то, чтобы сложнее, но объёмнее. Главное, понять, что функции эти в общем случае не поставляются в продажном комплекте, а разрабатываются самостоятельно покупателем. Далее два экземпляра, для русского языка и для английского. Они очень похожи и немедленно возникает желание их немного обобщить и унифицировать. Это неправильно, нам просто повезло, что в русском и английском числительные до ста формируются очень похоже. В других языках всё будет по другому.

Это необходимые константы:

```

var
    wR1          : array[0..9] of string =
('','один','два','три','четыре','пять','шесть','семь',
'восемь','девять');
    wR11         : array[1..9] of string =
('одиннадцать','двенадцать','тринадцать','четырнадцать','пятнадцат
ь','шестнадцать','семнадцать','восемнадцать','девятнадцать');
    wR10         : array[0..9] of string =
('','десять','двадцать','тридцать','сорок','пятьдесят','шестьде
сят','семьдесят','восемьдесят','девяносто');

    wE1          : array[0..9] of string =
('','one','two','three','four','five','six','seven','eight','ni
ne');
    wE11         : array[1..9] of string =
('eleven','twelve','thirteen','fourteen','fifteen',
'sixteen','seventeen','eighteen','nineteen');
    wE10         : array[0..9] of string='',=( 'ten',
'twenty','thirty','forty','fifty','sixty','seventy','eighty','nine
ty');

```

А это сам программный код, незаатейливый, но эффективный:

```

function Num2Russian(    num : integer) : string;
var
    n1,n10       : integer;
begin
    if num < 100 then begin
        n1:=num mod 10;
        n10:=num div 10;
        result:=wR10[n10] + ' ' + wR1[n1];
        if (num >= 11) and (num <= 19)

```

```
        then result:=wR11[n1];
    end
    else result:='no possible';
end;
{-----}
function Num2English(    num : integer) : string;
    var
        n1,n10            : integer;
begin
    if num < 100 then begin
        n1:=num mod 10;
        n10:=num div 10;
        result:=wE10[n10] + ' ' + wE1[n1];
        if (num >= 11) and (num <= 19)
            then result:=wE11[n1];
        end
    else result:='no possible';
end;
```

Как всё это использовать, попытайтесь догадаться сами.

Всё о списках. Или почти всё...

Список — это классика программирования, это то, чем программист начинает свою программистскую жизнь, в процессе программистского обучения, и чем кончает эту жизнь, наверное, я не пробовал. Что такое список? Чем больше думаю, тем больше возникает вопросов. Нужны ли списки вообще? Возможно, что и нет. Уже нет. Чем список отличается от массива? Тем, что список имеет переменную в процессе работы с ним длину, а массив нет, его длина не меняется. Но ведь это не так, есть динамические массивы, да и разница, даже без учёта динамических массивов незначительна. Очень просто обеспечить как бы переменную длину с помощью обычного, традиционного массива. Разумеется, списки экономят память — память выделяется только для тех элементов, которые заполнены, в отличие от массива. Но динамические массивы делают то же, да и для обычных массивов, если количество элементов ограничивается первыми тысячами, разница неощутима.

Так для чего же нужны списки? Не знаю ответа, они просто есть, и всё. А раз они есть, значить надо уметь ими пользоваться.

Просто список — TList, предназначен для хранения указателей. Создать, добавить три элемента, взять один, уничтожить по прочтении. Далее псевдокод:

```

var
    L                : TList;
    p                : pointer;
begin
    L:=TList.Create;

    p:=?;
    L.Add(p);
    p:=??;
    L.Add(p);
    p:=???;
    L.Add(p);

    p:=L[1];

    L.Free;

```

Перед присваиванием указатель откуда-то получает какое-то значение. При чтении индексы начинаются с нуля. Недостаток, вытекающий из особенностей — нет возможности сохранения в файл или поток и, соответственно, возможности восстановления. Это нетрудно восполнить, написав свой порождённый класс. Напоминаю, что любой экземпляр класса — тоже указатель. Можно

без труда хранить в списке и другие типы, имеющие размер в четыре байта — `integer` или `single`.

```
L.Add( Pointer(int));  
int:=Integer(L[0]);
```

Список для хранения строк, `TStringList`, незаметно включает в себя предущий `TList` (свойство `Objects`). Кроме того, он умеет сохраняться и восстанавливаться. Делает это он, к сожалению, не очень хорошо. Если сохранить два списка в один поток, а потом попробовать восстановить, то оба сохранённых списка окажутся в первом восстановленном. Это никуда не годится. Напишите свой, улучшенный, класс, порождённый от `TStringList`.

Подумайте о написании класса для хранения данных произвольной, но одинаковой длины, так, чтобы список сам выделял бы и освобождал для них память. То же самое для разной длины каждого элемента списка. Создайте список для дерева, в смысле теории графов, а не ботаники.

Всё Объектно ориентированное программирование

Гоша, он же Жора... © из кино

С терминологией путаница. У любой переменной есть тип, например `integer`, а есть и сама переменная, например `int`, вот так объявленная — `int : integer`. Вроде бы просто и понятно. С классами примерно так же, но то, что справа от двоеточия в зависимости от языка и его версии может называться класс или объект. То, что слева, соответственно будет или тоже объект или экземпляр класса. Такой разницей в терминах немного путает, но догадаться по контексту всегда можно.

Вообще-то, я всё об этом уже сказал в своей книге с незатейливым названием «Объектно Ориентированное Программирование». Сказал то, что знал, конечно, а я не утверждаю, что знаю по этой теме всё, хотя знаю очень много, а понимаю ещё больше.

Далее ООП означает Объектно Ориентированное Программирование. Три составляющих ООП — инкапсуляция, наследование, полиморфизм. ООП излагается в первую очередь применительно к Delphi, хотя отличия от других языков непринципиальны. Речь здесь идет о традиционных языках программирования — Delphi, C++, C#. Есть ещё и чисто объектные языки.

Инкапсуляция — если мы даём команду нарисовать квадрат, нас не интересует, каким именно способом он будет нарисован, мы должны только задать размер его стороны. И, что важнее, если мы даём команду нарисовать квадрату диагонали, мы уже не должны задавать сторону квадрата, достаточно указать, о каком именно квадрате идёт речь. Объект может существовать во многих экземплярах, почему-то это не сформулировано как догмат, но считается само собой разумеющимся, что, на мой взгляд, неправильно.

Что такое инкапсуляция, простыми словами? Если у класса есть свойство радиус и есть свойство площадь, то при изменении свойства радиус свойство площадь соответственно меняется. А по какой формуле меняется, никто не знает. Это и есть инкапсуляция.

Для общего понимания важно различать три уровня отношения программиста к классу, это отражено и в синтаксисе языка. Программист может написать класс с нуля и являться его полным хозяином. Программист может написать порождённый класс от написанного ранее, здесь свобода его действий ограничена программистом, создателем класса. И, наконец, программист может просто использовать созданный кем-то другим класс.

Класс имеет методы, поля и свойства. Класс объявляется в обычном модуле. Интерфейс класса объявляется в секции интерфейса (**interface**), реализация в

секции реализации (**implementation**). Очень простой модуль с очень простым классом, совершенно бесполезным, но разнообразным :

```

unit SimpleClass;

interface
  type
    TSimpleClass = class
      private
        fB          : single;

        procedure SetB(      value : single);

      protected
        procedure CD; virtual;

      public
        property b : single read fB write SetB;

        constructor Create;
        destructor Destroy; override;
        procedure C;

      private
        a          : integer;
        p          : pointer;
    end;
{-----}
implementation
{-----}
constructor TSimpleClass.Create;
begin
  a:=0;
  fB:=0;
  GetMem( p, 1024);
end;
{-----}
destructor TSimpleClass.Destroy;
begin
  FreeMem( p, 1024);
end;
{-----}
procedure TSimpleClass.C;
begin
  CD;
  //

```

```

end;
{-----}
procedure TSimpleClass.CD;
begin
    //
end;
{-----}
procedure TSimpleClass.SetB(    value : single);
begin
    if value >= 0 then fB:=value;
end;
{-----}
end.

```

Public, protected, private — уровни доступа и видимости. **Public** — то, что видит пользователь класса, **protected** — то, что видит тот, кто пишет порождённый класс, **private** — то, что видит только разработчик исходного класса. В C++ есть понятие дружественного класса — когда два класса имеют доступ к внутренним (**private**) полям друг друга. В Дельфи это тоже есть, но реализовано через совершенно неправильное место. Классы в Delphi являются дружественными, если они объявлены и реализованы в одном программном модуле (**unit**).

fB, a, p — поля. В Delphi считается абсолютно недопустимым объявлять поля в секции **public**. Технически препятствий этому нет, но идеологически это — ужас, ужас, ужас! Пользователь класса может пользоваться только свойствами, это принципиально.

B — свойство типа **single**. Слова **read** и **write** указывают, как берётся и записывается значение свойства. В нашем случае значение берётся из переменной fB. По традиции, имя связанной со свойством переменной совпадает с именем свойства с добавленной впереди буквой f. Запись значения в свойство производится через процедуру с традиционным именем SetB. В нашем конкретном случае процедура проверяет будущее значение свойства и не допускает отрицательных значений. В случае, если ситуация сложнее и извлечение значения свойства не так просто, используется функция, имеющая, опять же по традиции, имя GetB.

У каждого класса есть конструктор и деструктор. Даже если вы их не объявите, они всё равно есть. Все классы автоматически являются порождёнными от класса TObject, и наследуют от него всё, что могут, о наследовании чуть позже.

Конструктор создаёт экземпляр класса. Загадочно? Забудьте! Главное, что надо помнить:

```

SC.Create; // Неправильно!!!
SC:=TSimpleClass.Create; // Правильно

```

Транслятор пропустит и то и другое. Почему — не знаю.

Странно и смешно, наш устроен мозг,

*В нём четырнадцать извилин
 В девятнадцать лет.
 Две из них про секс, восемь про любовь,
 В двух хранится код подбезда,
 Две — зачем, не знаю.*
 © Ободзинский и немного ОСП

И у нас та же фигня. Почему транслятор это пропускает? И почему программа потом вылетает? И почему она вылетает не всегда? Да какая разница! Главное, всегда делайте правильно, а неправильно не делайте. Отвлекаясь от формальной стороны, смысл конструктора (и деструктора) очень прост. Конструктор конструирует, создаёт что-то нужное для работы класса. Деструктор всё портит, ломает и выбрасывает.

Что это значит? Сначала рекурсивно. В конструкторе вызываются конструкторы других классов, которые используются в нашем классе. В деструкторе вызываются, соответственно, их деструкторы. Это совершенно правильно, но малоинформативно, вопрос всё равно остаётся — а зачем конструктор? Хороший ответ содержится в примере выше — в конструкторе выделяется память, в деструкторе освобождается.

Теперь наследование. Пример очень простой и такой же бессодержательный. Интерфейс:

```
TSimpleClassAdd = class(TSimpleClass)
protected
  procedure CD; override;

public
  constructor Create;
  procedure C;
end;
```

Реализация:

```
constructor TSimpleClassAdd.Create;
begin
  inherited;
  ///
end;
{-----}
procedure TSimpleClassAdd.C;
begin
  inherited;
  ///
end;
```

```
procedure TSimpleClassAdd.CD;  
begin  
    inherited;  
    ///  
end;
```

Главное, что здесь заслуживает понимания — в том месте, где написано **inherited**, вызывается одноимённый метод из вышестоящего класса.

А полиморфизм мы здесь рассматривать не будем. Об этом я уже написал книгу — «Объектно-ориентированное программирование. Хорошая книга для хороших людей».

Простенький модуль для работы со строками. Простой, но очень полезный

Какая функция что именно делает — разберитесь сами.

```

unit UsStrUtils; // 08.10.2014
                // 15.11.2014

{-----}
interface
{-----}
// сколько слов в строке
function WordCount( S : string) : integer;
// выделить слово по номеру
function GetWord( S : string;
                  nomer : integer) : string;
// удалить слово по номеру
function DeleteWord( S : string;
                     nomer : integer) : string;
// заменить символ на другой
function ReplaceChar( S : string;
                      whatC : char;
                      forC : char) : string;
// заменить символ на строку
function ReplaceWi( S : string;
                    whatC : char;
                    forS : string) : string;
// заменить последовательность повторяющихся пробелов на один
function CompressSpaces( S : string) : string;
// строка состоит из одних только пробелов
function Empty( S : string) : boolean;
// удалить начальные и конечные пробелы
// и заменить буковки на маленькие
// кажется пустяк, а на самом деле - абсолютно незаменимая вещь
function Flatten( stroka : string) : string;
// если строка вроде бы в кодировке WIN-1251 ==> true
// если Dos ==> false
function WINstr( wStr : string) : boolean;
// из какого языка символ
function IsEngChar( ch : char) : boolean;
{-----}
implementation
uses
    SysUtils;
{-----}

```

```

function TrimSpaces(      S : string) : string;
var
    first,last           : integer;
begin
    first:=1;
    while (first < Length(S)) and (S[first] = ' ') do
        Inc(first);
    Delete( S, 1, first-1);

    last:=Length(S);
    while (last > 1) and (S[last] = ' ') do
        Dec(last);
    Delete( S, last+1, 999);

    TrimSpaces:=S;
end;
{-----}
function WordCount(      S : string) : integer;
var
    num                  : integer;
    inWord               : boolean;
    i                    : integer;
begin
    S:=TrimSpaces(S);
    num:=0; inWord:=false;
    for i:=1 to Length(S) do begin
        if S[i] = ' ' then inWord:=false
        else begin
            if not inWord
                then Inc(num);
            inWord:=true;
        end;
    end;
    result:=num;
end;
{-----}
function GetWord(        S      : string;
                        nomer : integer) : string;
var
    num                  : integer;
    inWord               : boolean;
    found                : boolean;
    theWord              : string;
    i                    : integer;
begin
    S:=TrimSpaces(S);
    num:=0; inWord:=false; found:=false; theWord:='';
    for i:=1 to Length(S) do begin

```

```

    if S[i] = ' ' then begin
        inWord:=false;
        if found
            then Break;
    end
    else begin
        if not inWord
            then Inc(num);
        inWord:=true;
        if num = nomer
            then found:=true;
        if found
            then theWord:=theWord + S[i];
    end;
end;
result:=theWord;
end;
}-----}
function GetWordEx(      S      : string;
                       nomer : integer;
                       var wPos : integer;
                       var wLen : integer) : string;
var
    num           : integer;
    inWord        : boolean;
    found         : boolean;
    theWord       : string;
    i             : integer;
begin
    S:=TrimSpaces(S);
    num:=0; inWord:=false; found:=false; theWord:='';
    wPos:=0; wLen:=0;
    for i:=1 to Length(S) do begin
        if S[i] = ' ' then begin
            inWord:=false;
            if found
                then Break;
        end
        else begin
            if not inWord then begin
                Inc(num);
                if num = nomer
                    then wPos:=i;
            end;
            inWord:=true;
            if num = nomer
                then found:=true;
            if found then begin

```

```

        theWord:=theWord + S[i];
        Inc(wLen);
    end;
end;
end;

```

```

    result:=theWord;

```

```

end;

```

```

{-----}

```

```

function ReplaceChar(      S      : string;
                        whatC : char;
                        forC  : char) : string;

```

```

    var

```

```

        i                : integer;

```

```

begin

```

```

    for i:=1 to Length(S) do

```

```

        if S[i] = whatC

```

```

            then S[i]:=forC;

```

```

    result:=S;

```

```

end;

```

```

{-----}

```

```

function ReplaceWi(      S      : string;
                        whatC : char;
                        forS  : string) : string;

```

```

    var

```

```

        i                : integer;

```

```

begin

```

```

    for i:=1 to Length(S) do

```

```

        if S[i] = whatC then begin

```

```

            Delete(S,i,1);

```

```

            Insert(forS,S,i);

```

```

        end;

```

```

    result:=S;

```

```

end;

```

```

{-----}

```

```

function CompressSpaces( S : string) : string;

```

```

    var

```

```

        newS                : string;

```

```

        lastCh              : char;

```

```

        i                   : integer;

```

```

begin

```

```

    newS:=''; lastCh:=#0;

```

```

    for i:=1 to Length(S) do begin

```

```

        if (S[i] <> ' ') or (lastCh <> ' ') then begin

```

```

            newS:=newS + S[i];

```

```

            lastCh:=S[i];

```

```

        end;

```

```

    end;

```

```

    result:=newS;
end;
{-----}
function Empty(      S : string) : boolean;
var
    i                : integer;
begin
    result:=true;
    for i:=1 to Length(S) do
        if S[i] <> ' ' then begin
            result:=false;
            Break;
        end;
    end;
end;
{-----}
function DeleteWord(      S      : string;
                        nomer : integer) : string;
var
    num                : integer;
    inWord              : boolean;
    found               : boolean;
    theWord             : string;
    theEnd              : integer;
    i                   : integer;
begin
    S:=TrimSpaces(S);
    num:=0; inWord:=false; found:=false; theWord:='';
    theEnd:=Length(S);
    for i:=1 to Length(S) do begin
        if S[i] = ' ' then begin
            inWord:=false;
            if found then begin
                theEnd:=i-1;
                Break;
            end;
        end
        else begin
            if not inWord
                then Inc(num);
            inWord:=true;
            if num = nomer
                then found:=true;
            if found
                then theWord:=theWord + S[i];
        end;
    end;
end;
end;

```

```
Delete( S, 1, theEnd);
```

```
result:=S;
```

```
end;
```

```
{-----}
```

```
function Flatten( stroka : string) : string;
```

```
begin
```

```
result:=Trim( AnsiLowerCase(stroka));
```

```
end;
```

```
{-----}
```

```
function WINStr( wStr : string) : boolean;
```

```
var
```

```
nANSI, nOEM : integer;
```

```
ordStr : integer;
```

```
i : integer;
```

```
begin
```

```
result:=true;
```

```
nANSI:=0;
```

```
nOEM:=0;
```

```
for i:=1 to Length(wStr) do begin
```

```
ordSTR:=Ord(wStr[i]);
```

```
if ((ordStr >= 128) and (ordStr <=175)) or
```

```
((ordStr >= 224) and (ordStr <=241))
```

```
then Inc(nOEM);
```

```
if ((ordStr >= 192) and (ordStr <= 255)) {or
```

```
((ordStr >= 142) and (ordStr <= 255))}
```

```
then Inc(nANSI);
```

```
end;
```

```
if nOEM > nANSI
```

```
then result:=false;
```

```
end;
```

```
{-----}
```

```
function IsEngChar( ch : char) : boolean;
```

```
begin
```

```
result:=((ch>='A') and (ch<='Z')) or ((ch>='a') and
```

```
(ch<='z'))
```

```
end;
```

```
{-----}
```

```
end.
```

Задания для самостоятельного развития

Сначала обязательная программа. По одному заданию на каждую главу.

1. Найдите в интернетах картину «Масленица», автор то ли Куклин, то ли Окуклин. Если долго смотреть на картину, можно заболеть той же болезнью, что и автор. Угадайте, какой именно болезнью. Об этом вас обязательно спросят на собеседовании. Говорят, правильному психиатру для правильного ответа достаточно надеть белый халат.
2. Нарисуйте график чего-то действительно нетрадиционного. Необязательно ведь на вход поступают именно числа, в смысле просто действительные числа. А если мы сейчас живём в учебнике функций комплексного переменного? *Необходимо* ли для этого 3D отображение? И *достаточно* ли для этого 3D отображения? А если и аргумент и значение являются множествами?
3. Напишите объект — генератор знаков числа π неограниченной точности. Найдите индекс, с которого начинается номер вашего телефона. Если не нашли, придумайте правдоподобное объяснение, почему не нашли. Этот навык пригодится вам в общении с руководством.
4. А если нужно хранить данные по наполным весам индивидуально для каждой квартиры? Для каждого дома? Для района? Для всего города? На каком этапе сломается текущая версия вашей программы? На каком этапе вы задумаетесь?
5. Осмотрите по часовой стрелке ваших коллег, сотрудников и подчинённых. Вычислите, кто из них больше всех будет сопротивляться внедрению VSS. Внедрите. Убедитесь в своей правоте. Запишите фамилии в блокнотик.
6. На всяком ли языке программирования можно написать динамическую библиотеку? Найдите язык, на котором нельзя. Докажите. А теперь напишите на нём библиотеку и докажите, что были неправы.
7. Что-то я задумался. Опыты по измерению скорости сортировки я проводил над массивом из ста тысяч элементов, случайным образом заполненным. И это правильно. А что, если взять массив из ста тысяч, а лучше из ста миллионов элементов, но с самого начала правильно упорядоченных? Какая сортировка будет эффективнее? Из теории мы знаем, что если элементы *почти* упорядочены, то сортировка Шелла выбывает из соревнования. Проверьте.

8. Прочтите внимательно:

М.Р.Шура-Бура, В.С.Штаркман «Вычислительная машина М-20. Инструкция по математической эксплуатации», М, 1962

Запрограммируйте крестики-нолики. На худой конец — обход шахматным конем всех клеток шахматной доски без повторов. Это возможно. Вы справитесь.

9. Очень простое задание. Найдите книгу

А.Е.Ферсман «Занимательная геохимия», Издательство Академии наук СССР, Москва, 1959

Это то издание, которое я с полпинка нашёл, как сейчас принято говорить, в электрическом Интернете. Когда я был маленький я читал издание примерно сорок девятого года. Нет, я не настолько старый. Вот такие книги были тогда в библиотеках. Впрочем, год выпуска абсолютно неважен, потому что академик Ферсман умер в сорок пятом году, оставив книгу в набросках и черновиках. Дописали книгу неизвестные писатели. В смысле, неизвестно, были ли они писателями.

Как результат — книга ни разу не занимательная, с Перельманом («Занимательная физика», «Занимательная геометрия и далее по списку») не сравнить. Не сравнить даже с книгой того же Ферсмана «Занимательная минералогия».

А зачем здесь эта книга? Сама книга для наших целей не очень интересна, зато интересно приложение к ней под названием «Геохимик в поле». Прочитайте, там немного страниц. Подумайте.

А теперь о главном. То есть, о математике. В Университете, где я учился, есть человек, доказавший, что нечётных совершенных чисел не бывает. Я об этом узнал из городской версии Комсомольской Правды. Тянет ли это открытие на Нобелевку, или на что там Перельман плюнул и растёр? Тянет ли это на докторскую? Почему этого человека не любят в Университете?

10. Объедините всё вместе, все три задачи — бредогенератор, поиск скрытого смысла и выбор ближайшего слова. Вы вводите инструкции для бредогенератора с орфографическими ошибками, по принципу — как слышится, так и пишется.

Отвлечитесь, изучите технологию ввода в компьютер японских иероглифов. Узнайте, какие преимущества японский ввод в компьютер даёт малограмотным японцам и гайдзинам (уточните значение этого слова и является ли оно в Японии матерным). Выясните, почему у китайцев с этим проблемы, завидуют ли они японцам, и как это может влиять на межнациональные отношения.

Скорректируйте текст с помощью технологии поиска ближайшего слова. Выбирать слово лучше из словаря языка с кириллицей, но не очень русского. Попробуйте болгарский или монгольский. Найдите в этом тайный смысл.

Обязательная программа закончена, теперь произвольная.

1. Запрограммируйте этот ряд, по возможности, не ругаясь матом. Говорят, он очень быстро сходится. Это вас утешит.

Канадские математики Джонатан и Питер Борвейны в 1987 году нашли удивительный ряд:

$$\frac{1}{\pi} = 12 \sum_{n=0}^{\infty} \left\{ \frac{(-1)^n (6n)! [212\,175\,710\,912 \sqrt{61} + 1\,657\,145\,277\,365 + (n!)^3 (3n)! [5\,280 (236\,674 + 30303 \sqrt{61})]^{3n + \frac{3}{2}} + n(13\,773\,980\,892\,672 \sqrt{61}) + 107\,578\,229\,802\,750]}{(n!)^3 (3n)! [5\,280 (236\,674 + 30303 \sqrt{61})]^{3n + \frac{3}{2}}} \right\}$$

где $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$; $0! = 1$

2. Нарисуйте в графике Turbo Pascal версии 5.5 ваше видение отказа Перельмана от кучи денег. Деньги покрасить зелёным цветом. Перельмана изобразить в кипе, с пейсами, с оседельцем и в кокошнике одновременно. Для перехода на следующий уровень программистского мастерства обязательно дорисовать balalaika, домашний medved и samovar с vodka. Для высшего уровня — водка должна кипеть.

Полезная ссылка http://www.luth.org/web_extras/Flynn_Balalaika-GAL.pdf

3. Запрограммируйте, в конце концов, функцию для вычисления функции Дирихле. Без матов, но рекурсивно.
4. Путём статистического анализа докажите, что все сочинения Шолохова написал Солженицин, а все манускрипты Солженицина — Шолохов.

Путём Гугля выясните, кто все эти люди, упомянутые в предыдущем пункте — Шолохов, Манускрипт и Солженицин.

5. Напишите эмулятор M-20 на IBM PC или под Windows, как вам больше нравится формулировать. Легко? А теперь наоборот! А вы что хотели?

Комлев Николай Юрьевич

Полезное программирование

Ответственный за выпуск:

В. Митин

Верстка и обложка:

СОЛОН-Пресс

ООО «СОЛОН-Пресс»

123001, г. Москва, а/я 82

Телефоны/495) 617-39-64, (495) 617-39-65

E-mail: avtor@solon-press.ru,

www.solon-press.ru

По вопросам приобретения обращаться:

ООО «ПЛАНЕТА АЛЬЯНС»

Тел.: (499) 782-38-89, www.aliants-kniga.ru

По вопросам подписки на журнал «Ремонт & Сервис» обращаться:

ООО «Ремонт и Сервис 21»

тел.: (495) 617-39-64, www.remserv.ru

ООО «СОЛОН-Пресс»

115142, г. Москва, Кавказский бульвар, д. 50

Формат 70x100/16. Объем 16 п. л. Тираж 200 экз.