

*Серия «Программирование»*

**Рик Гаско**

***Простой учебник  
программирования***

**Москва  
СОЛОН-Пресс  
2018**

УДК 681.3  
ББК 32.973-18  
К 63

*Под редакцией Н. Комлева*

**Рик Гаско**

**Простой учебник программирования.** — М.: СОЛОН-Пресс, 2018. — 320 с.: ил. (Серия «Программирование»)

ISBN 978-5-91359-281-1

Книга написана необычным для многих — живым, простым и емким языком. Автор не любит длинных описаний программ, поэтому прерывается на пояснения, что делает книгу удобной для понимания и легкой в усвоении материала.

Чтение учебника не утомляет, а наоборот, — захватывает. Это лучший учебник программирования, по крайней мере, из всех доступных на русском языке. Проработав книгу от начала и до конца, читатель получит ясное понимание — что это такое, программирование. Выбранный в дальнейшем конкретный язык программирования неважен, важны принципы.

В первую очередь это книга для тех, кто, являясь профессионалом в своей области, хочет овладеть программированием или, по крайней мере, научиться разговаривать с программистами на равных.

От самых начал до понятий достаточно глубоких. Прочтите и узнайте!

**Rick Gassko. Straight Programming.** — Moscow: SOLON-Press, 2018. — 320 p. (*The "Programming" series*)

По вопросам приобретения обращаться:

**ООО «СОЛОН-Пресс»**

Тел: (495) 617-39-64, (495) 617-39-65

**E-mail: [kniga@solon-press.ru](mailto:kniga@solon-press.ru), [www.solon-press.ru](http://www.solon-press.ru)**

ISBN 978-5-91359-281-1

© «СОЛОН-Пресс», 2018

© Рик Гаско, 2018

© Rick Gassko, 2018

*Посвящается  
тем  
кто прочитал  
мою предыдущую книгу  
и кто теперь на самом деле хочет  
научиться писать настоящие программы*

## Содержание

Всяческие вступления и предисловия .....	8
Вступление №1 .....	8
Вступление №2 – от старой книги с моей давно изменившейся точки зрения.....	11
Для кого эта книга .....	11
Почему я решил эту книгу написать. И почему именно я.....	11
Почему буковки разные .....	12
Что я ожидаю, что читатель уже знает .....	12
Ещё раз - почему Паскаль?.....	14
А почему, собственно, <i>именно</i> Turbo Pascal .....	16
Что бы ещё почитать .....	17
Disclaimer .....	19
Том первый, Война и немцы .....	20
Глава 1 Просто программа.....	20
Самая простая программа, которая ничего не делает .....	20
Очень простая программа, которая делает хоть что-то .....	24
Улучшаем программу. Много новых слов .....	26
Весело, в цветочек.....	31
И кое-что ещё.....	32
Глава 2 Переменные.....	33
Что такое и зачем.....	33
Ввод и вывод.....	37
Дроби.....	38
Глава 3 Условные операторы .....	43
Что такое и зачем.....	43
Усложняем .....	44
Окончательно усложняем .....	46
Небольшая программка и кое-что ещё .....	47
Глава 4, очень простая Немного графики .....	50
Начальные заклинания.....	50
Точки, линии и окружности .....	51
Прямоугольнички и кружочки .....	53
Красивые буковки .....	55
Что там ещё осталось? .....	56
Полезная вещь – метод опорной точки .....	57
Глава 5, сложная Циклы и массивы.....	59
Просто массив.....	59

Просто цикл .....	61
Просто циклы и графика .....	67
Ещё одна несложная программа .....	70
А теперь всё вместе .....	72
Опыты .....	75
Ещё опыты .....	77
Самый главный опыт .....	78
Как не делать ничего .....	81
Что-нибудь полезное .....	84
Глава 6 Строки .....	91
Просто строка .....	91
Просто строка и её процедуры .....	93
Строка и цикл .....	95
Ой, кто пришёл! .....	98
Считаем, наконец, слова .....	100
Глава 7, продолжение пятой Ещё циклы и массивы .....	104
Массивы двумерные и далее .....	104
Вложенные циклы .....	105
Пример посложнее .....	108
О самом важном. Всё сразу и побольше .....	111
Другие циклы .....	113
Глава 8 Процедуры и функции .....	120
Процедура без параметров .....	120
То же и с параметрами .....	121
А какие бывают параметры? .....	124
О грустном .....	127
Скучная, но необходимая теория .....	129
А теперь функция .....	132
А теперь тараканчик .....	135
А этот раздел просто больше некуда было вставить .....	137
Всем стоять и не разбегаться! .....	145
Применим к тараканчику .....	146
Глава 9 Совсем настоящая программа .....	148
Про что программа? .....	148
Отладка. Давно пора .....	149
Ещё одна очень важная вещь. Модули .....	153
С чего начать? .....	157
Поле .....	161
Крестик и нолик .....	162

Курсор и чтобы бегал.....	163
Делаем ход .....	166
А не выиграл ли кто?.....	167
Вражеский интеллект.....	169
Имеем в результате.....	176
Глава 10 Файлы.....	177
Коротенько. Почему это очень важно .....	177
Найти и снова найти.....	177
Файлы текстовые и никому не нужные.....	179
Бинарные .....	183
Бинарные файлы. Задача.....	186
К чему-нибудь прикрутим .....	187
Глава 11 Всякие глупости, она же Глава очень длинная .....	190
Записи. И как мы только без них обходились! .....	190
Указатели .....	192
Round, Ord, Chr и другие пустячки.....	196
Есть такая штука – множество .....	200
Совсем глупость – про музыку.....	201
Оно надо? Рекурсия.....	209
Меряем время.....	213
Страшная сила .....	219
Никаких новых слов.....	222
Том второй, пять старушек – рупь.....	225
Глава 2-1 Ещё раз: простая программа и переменные .....	225
Повторение пройденного .....	225
Разбор полётов.....	230
Глава 2-2 Вспомнить всё или Не очень сложная программа – Ханойские Башни.....	232
О чём речь? .....	232
Всем всё понятно, программируем.....	233
Решительно кончаем программу.....	242
Глава 2-3 Всё таки кое что новое .....	247
Как устроена большая программа.....	247
Очень простая и маленькая большая программа.....	247
Глава 2-4 По ту сторону - опустимся чуть ниже .....	261
Вступление.....	261
Начало. Просмотр картинок .....	261
Вариант 1.....	263
Вариант 2.....	264

Вариант 3 .....	267
Глава 2-5 Указатели. Зачем они действительно нужны .....	276
А чем списки лучше массивов? А чем хуже? .....	276
Всё то же самое, но медленно и по шагам. Шаг первый .....	277
Всё то же самое, но медленно и по шагам. Шаг второй .....	282
Усложняем. Шаг третий .....	289
Дополнение Всякие важные вещи .....	296
Как установить Турбо Паскаль .....	296
Как настроить Турбо Паскаль, чтобы было приятно и удобно .....	299
И ещё кое-что .....	301
Имейте свой стиль .....	301
Все полезные клавиши на одной странице .....	307
Все типы данных на одной странице (ну, на двух...)	
Даже те, которые от вас скрывали .....	308
Чем заняться на досуге .....	310
Модуль для работы с клавиатурой .....	311
Модуль для работы с нотами .....	313
Полный и аккуратный текст программы про Ханойские Башни .....	316

## Всяческие вступления и предисловия

### Вступление №1

Почему это не просто Вступление, а Вступление под номером один? Сейчас поймёте. Это потому, что я очень честный человек. И я очень честно признаюсь, что 50% этой книги – совершенно свежий и новый текст. А другие 50%, как легко догадаться, не совсем новый и не совсем свежий.

Что я могу сказать в своё оправдание и убедить вас купить и прочитать эту книгу – главное, конечно, купить? После покупки читать совершенно не обязательно.

Первое – это хорошая книга. Я плохих не пишу. Эта книга *действительно* простой учебник программирования, и даже альтернативно одарённый может по ней научиться программировать – если захочет.

Второе – откуда в этой книге взялся не очень новый текст? И откуда, кстати, взялся новый? Несколько лет назад я написал книгу по программированию на языке Паскаль в среде программирования Turbo Pascal. Книга была благосклонно принята Главным Издателем™, который предварительно отправил её – книгу – на экспертизу к, извините за тавтологию, к экспертам. Через два месяца я объяснил Главному Издателю™, что главный эксперт здесь – это я. Главный Издатель™ согласился, но за две недели до выпуска объявил, что Turbo Pascal нынче не в тренде, а в тренде нынче, наоборот, Pascal ABC, потому что его заставляют учить в школах. Действительно заставляют, я проверял.

Я стремительно переписал книгу под Pascal ABC, который, напоминая, преподают в школах. Книга даже в таком покоцанном виде имела три переиздания, что как бы намекает на её определённый, не низкий, уровень.

Третье – одна моя давняя знакомая-подруга имеет то ли дочку, то ли внучку, которая, чуть что сделали не по ней объявляет злобным голосом – *Сделайте, как было!* Так вот, мне написал письмо Главный Издатель, в котором попросил вежливыми русскими буквами – *Сделайте, как было!*

Как несколько раньше говорил товарищ Сталин – *Гитлеры приходят и уходят, а немецкий народ остаётся*. Pascal ABC приходит и уходит, а простое программирование остаётся. И это правильно, товарищи.

Четвёртое – я достал исходный текст книги, сдул с него пыль, но посчитал совершенно бесчестным издавать её в таком виде. Ведь кто-то уже заплатил свои деньги за этот текст!

Пятое – я прошёлся по книге, по каждому абзацу и, уверяю вас, в книге не случилось ни одного абзаца мною неизменённого. Что важнее, сразу после первой книги, которая, если вы помните, называлась *Самоучитель игры на Паскале*, я начал писать новую, под названием *Школа игры на Паскале*. Книга, по объективным обстоятельствам, так и осталась наполовину недописанной – или наполовину написанной – как кому менталитет подсказывает. Не пропадать же добру.

А теперь – о главном! Я, во-первых, восстановил всё о Турбо Паскале, во-вторых, переписал весь текст, в-третьих, добавил сотню страниц из *Школы*.

*И кое что ещё,  
И кое-что другое,  
О чём не говорят,  
О чём не учат в школе* © Старая смешная песня

Далее, я учёл все замечания благодарных и восторженных читателей, а замечания завистливых и злобных, наоборот, проигнорировал. Ещё я стандартизировал текст под мои последующие, теперь уже вышедшие, книги – моноширинный шрифт для текстов программ в интегрированных средах программирования используется не случайно. Ещё я везде заменил *Вы* с большой буквы на *вы* с маленькой. Мне кажется это более правильным, орфографически допустимым и не обидным.

И вы даже не представляете, сколько в книге обязательно остаётся опечаток – сколько бы раз и сколько бы людей её не вычитывали.

*Решено было не допустить ни одной ошибки. Держали двадцать корректур. И все равно на титульном листе было напечатано: "Британская энциклопедия" © Ильф и Петров*

А теперь – о самом главном! Эта книга – не учебник какого-то конкретного, пусть и самого лучшего, языка программирования! Это и есть тот самый что ни на есть самый простой учебник программирования. Когда Великий Дейкстра написал свою гениальную книгу *Дисциплина программирования*, он не использовал ни один существующий язык, он тут же, на месте, изобрёл свой собственный!

И все были счастливы. Так считайте, что и Турбо Паскаль, мною использованный, так же изобретён мною специально для этой книги, хотя это, увы, не совсем так. Даже, точнее, совсем не так. Язык – это условность, фикция, для обучения концепциям программирования он абсолютно не важен, кроме, конечно, случаев использования в программировании совершенно ортогональных, маргинальных и перпендикулярных языков. Впрочем, не пугайся, мой маленький дружок. И на эту тему, об этих языках я собираюсь написать книгу. А Добрый Издатель™ даже пообещал её опубликовать. Потом.

## **Вступление №2 – от старой книги с моей давно изменившейся точки зрения**

### **Для кого эта книга**

Для тех, кто хочет научиться программировать. И кто при этом не умеет программировать вообще. Возможно, другая книга научила бы вас программировать на Delphi/Pascal/Python быстрее. Но если вы хотите научиться программировать, неважно на каком языке, то эта книга – то, что вам нужно. Принципы программирования остаются неизменными, а им я и стараюсь научить. Ну а уж если вы хотите научиться программировать с нуля, и именно на Паскале, то мы с вами встретили друг друга... Это именно учебник программирования, неважно на чём, хоть на кухонном комбайне, в конце концов.

### **Почему я решил эту книгу написать. И почему именно я**

Так получилось, что шесть лет я учил программировать. Сначала решил для интереса попробовать один год, но получилось шесть.

Есть очень много преподавателей программирования со значительно большим педагогическим стажем.

Но до того как начать учить, я много лет сам программировал. И пока учил, тоже программировал. Сейчас учить закончил, но всё равно программирую.

// Занудства ради

Преподавание – один сплошной восторг, если это развлечение после работы. Первые два-три года. Потом надоедает и утомляет. Потом или бросают, или занимаются этим профессионально. Я бросил.

// конец Занудства ради

А в придачу руководил, и руковожу группой коллег-программистов, а это занятие ещё более увлекательное. Недаром переводная с американского языка книга о ремесле руководства программистами называется *Как пасти котов*.

Так получается, что опытные педагоги, обучающие программированию, обычно программисты-теоретики и ничего существенного запрограммировать им в жизни не довелось. Опять-таки, занудства ради, среди моих компаньонов по программированию был целый доктор

физико-математических наук. Впрочем, вряд ли и он что-то запрограммировал, что можно было бы продать. Несмотря на национальность.

А опытные программисты-разработчики обучают редко - не хочется ерундой заниматься да и вообще...

В общем, куда ни кинь – учить некому.

И тут вхожу я весь в белом...

### Почему буковки разные

Названия клавиш будут выделены вот так – F2,Enter. Если нажаты сразу две клавиши, это будет выглядеть вот так – Ctrl/F9. Пункты меню Турбо Паскаля заключаются в угловые скобки – <Options\Save>.

А тексты программ будут выглядеть вот так:

```
program kuku;  
begin  
    {Вот тут программа}  
end.
```

Полужирным шрифтом будут выделены так называемые зарезервированные слова, редактор Турбо Паскаля их тоже выделит. Курсивом выделены комментарии – в редакторе Паскаля они будут выглядеть бледненько.

Зарезервированные слова в программах я пишу маленькими буквами, в тексте могу и большими – для наглядности. То же самое с однобуквенными именами – I, K, X и тому подобное. В программах они будут маленькими, а в тексте – большими – чтобы не потерялись.

### Что я ожидаю, что читатель уже знает

Можно конечно начинать программировать с полного нуля, не зная абсолютно ничего ни о чём. Такие подопытные попадались и, в общем, ничего. В смысле научить всё равно можно.

*Если зайца долго бить,*

*Можно выучить курить!* © Народный стих

Но некий минимум знаний всё же очень ускоряет процесс.

Естественно, я понимаю, что общее развитие у будущих программистов отсутствует начисто. Книжек они не читают, Бабеля от Бебеля не отличают, пишут по-русски фантастически безграмотно. Это нормально. Татьяна Ларина, как известно *изъяснялася с трудом на языке своём родном*, Бабеля с Бебелем я и сам не читал, а орфографию спеллчекер проверит.

За время с написания этой книги, один из работавших у меня молодых программистов стал чемпионом России по программированию. Это я к тому, что писать по-русски без ошибок он вообще не мог, и вам не надо.

*На заборе в честь Гоголя написано слово Вий, но с двумя ошибками* © А как бы вы, как программист, обработали в своей программе эту ситуацию?

Однако от программиста всё же хотелось бы:

Знание арифметики и умение считать в уме, хотя бы на уровне целых чисел – как это ни покажется странным, несмотря на то, что компьютер считает быстрее человека, от программиста арифметика очень даже требуется. Дробные числа можно считать на бумажке. Для профессионального программиста знание высшей математики крайне желательно, и чем с большим количеством разделов этой самой математики он хотя бы поверхностно знаком, тем лучше. Знающий математику программист крайне полезен в хозяйстве, его все любят и повышают ему зарплату – шутка – пояснение для людей без чувства юмора. Утешение – программист, не знающий математики, совершенно не понимает, чего он лишён и с довольным лицом сидит в своём загончике, блаженно похрюкивая и пуская пузыри.

Из геометрии – ну хотя бы понимание, что такое система координат. Сокровенное знание, чем круг отличается от окружности, также весьма полезно. Самое важное – несложная формула расчета расстояния между двумя точками на плоскости. Она обязательно пригодится, рано или поздно, скорее даже рано.

Вот она, формула:  $d_{ab} = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$ .

Для гениев обязательно – умение расширить эту формулу на n-мерное пространство. Утешение для незнающих – если не будете заниматься графикой – ну её нафиг, эту геометрию.

Ещё из математики – системы счисления и, как ни страшно это звучит, немного математической логики. Хотя, куда-то меня не туда несёт – многие не знают – и ничего. Забудьте.

Английский язык на уровне знания сотни-другой слов – begin, end, repeat, until, red, green, blue. Очень способствует. Для профессионального программиста знание технического английского абсолютно обязательно. Профессиональная пригодность незнающего английский язык программиста автоматически делится на два. Зарплата тоже. Утешение – умение требуется только одно – перевести текст; говорить и понимать на слух ни к чему. Сам-то я говорить могу, а на слух не понимаю. Ну так мне в школе на уроках пения и петь запрещали.

*Чукча не читатель, чукча писатель* © Чукча

Из чисто компьютерных материй – что есть такая сущность – каталог (директория (папка)), что есть такая единица измерения – байт, что есть такая штука – файл, что бывают они текстовые и бинарные и что лежат они в каталогах. Суперзнание – что была такая штука DOS, и что были такие программы, которые под ним работали. Если про байты и файлы не понимаете – ну не знаю, чем и утешить. Если вы думаете, что в наше продвинутое время это совсем ни к чему, то очень даже ошибаетесь.

### **Ещё раз - почему Паскаль?**

Давным-давно, когда я был маленьким программистом, а языков программирования уже было много, в каждой книжке по программированию повторялось заклинание, что нет хороших и плохих языков программирования, нет, и не может быть самого лучшего языка, нет, и не может быть языка на все случаи жизни. Прошло много времени, с предрассудками давно покончено, век наивности завершился, зелёная трава кончилась. Над миром распростёр сумрачные крылья лучший язык в мире на все случаи – C++. Точнее, мне так показалось, потому что так

оно и было в моей окрестности. Если взглянуть чуть шире, то не всё так однообразно. Есть и малосимпатичная мне Java, есть и более симпатичный мне C# – напоминаю, произносится Си Шарп – вот эта вот решётка – # – является музыкальным знаком *диез*.

Upd. Уже устарело. C++ заворачивается в простыню и ползёт в направлении кладбища.

Так зачем же я предлагаю не совсем модный Паскаль (мне больше нравится писать название по-русски)? Для начала, рекомендация меркантильная - на нашей части суши Турбо Паскаль (в реинкарнации Object Pascal/Borland Delphi) всё ещё очень популярен, наверное, популярнее, чем во всех других частях суши, вместе взятых. В последние годы идёт процесс вытеснения Паскаля и Delphi в пользу сами знаете чего. Если ваша тётя живёт на Брайтон-Бич, то C++ конечно полезнее. А если ваш дядя проживает в деревне Красный Слон, учите Паскаль. И C#, само собой.

*Не выпендривайся, Петрик. Слушай свою любимую песенку про комбайн*  
©Анекдот.

Так что если вы, дорогой читатель, не собираетесь отправляться программировать в более другие места прямо завтра, Паскаль вам очень даже пригодится в плане зарабатывания денег. Далее, бесконечное число раз повторено, что Паскаль - идеальный язык для обучения программированию. Как ни странно, так оно и есть - в конце концов, он для того и создан. Ещё дальше, с желанием оттоптать по телу безобидных сиплюсплюсников - C++ язык очень сильный, позволяющий программисту буквально всё. Паскаль не позволяет. Шаг влево, шаг вправо - ошибка компиляции. Си программиста не ограничивает ни в чём, Паскаль берёт за хобот и тащит унылой, но безопасной дорогой.

Сильного программиста Паскаль местами сковывает, хотя для сильного программиста как всегда - если нельзя, но очень хочется, то есть в запасе хитрый фокус. Но в реальной жизни сильных программистов мало. Больше средних. Ещё больше слабых.

И вот для них и существует Паскаль. Он, худо-бедно, ГАРАНТИРУЕТ получение приемлемого результата. А когда программирование

используется не как средство самоудовлетворения, а как возможность для получения разноцветных бумажек, именуемых также деньгами, наличие результата становится главным. Пробегаящим мимо сишникам просьба не мнить себя могучими кодерами и не плевать в колодец.

Ну и главное - Паскаль лично мне очень нравится.

### **А почему, собственно, именно Turbo Pascal**

Потому что, как минимум, учиться программировать надо именно на Паскале, в этом я вас, кажется, убедил. А теперь вполне резонный вопрос - зачем, собственно, начинать с Turbo Паскаля (я опять по-русски) если есть Delphi. Надо признать, что программировать на Turbo Паскале, кроме как в процессе обучения, почти наверняка не придётся - если только не достанется сопровождать какую-нибудь древнейшую программу. Turbo Паскаля в природе уже давно нет - есть только Delphi. Печально, но такова суровая реальность.

Так почему Паскаль? А вот почему. Логика Delphi не совсем проста и привычна, даже для опытного программиста, переходящего на него с Паскаля или любого традиционного языка. Традиционный язык – тот, который, что вижу - то пою. Программа медленно и торжественно разворачивается сверху вниз. Как сильно в Delphi будет вам не хватать маленькой радости – той, что у программы есть начало и есть конец. Это как воздух – пока есть, его не замечаешь. Вот так и с Turbo Паскалем, извините за пафос.

У дельфовой программы начала нет, и конца нет. Вся она состоит из обрывков-методов, реагирующих на различные события в жизни исполняемой программы. Потом, немного поработав, программист поймёт, что это логика правильная и естественная, но сначала выглядит это как-то странно и ужасно. Кроме того, Delphi оказывает полумедвежью услугу разработчику – отчасти само генерирует текст. Это плюс – при том маленьком условии, что вы этот сгенерированный текст полностью понимаете и можете в любой момент переписать его сами.

Ну и конечно, любимый вопрос дельфиста, за который над ним глумятся иноязычные коллеги программисты – а где взять вот такой компонент? Синдром Гарри Поттера - махнуть волшебной палочкой, компонентом в

смысле – и счастье. Быстро и дешево. Поневоле оценишь справедливость армейского юмора – мне не надо, чтоб быстро, мне надо, чтоб ты задолбался! В нашем случае – научился!

Так что начинать программировать сразу на Delphi (или аналогичном средстве разработки) - это беспощадная травма для профессионального будущего начинающего программиста.

- *Всё, всё... Всё. Теперь так и останется...*

- *Что останется?*

- *Что, что? Косоглазие!!!* © к/ф Любовь и голуби

Величайший программист всех времён и народов Дейкстра вообще писал программы – в своих книгах, по крайней мере – на лично им выдуманном исключительно для этих целей языке программирования. Чем Паскаль хуже? Паскаль лучше! Он, по крайней мере, существует, как объективная реальность.

### Что бы ещё почитать

Само собой, лучше всего читать и перечитывать только мои книги – изданные, переизданные, неизданные и недоизданные. Список изданных и переизданных будет, надеюсь, на задней обложке. Впрочем, в позапрошлый раз там напечатали портрет какого-то совершенно левого мужика. Объясняю для девушек – я какой угодно, но не лысый!

Учебник обычно сопровождается задачиком, или идёт с ним вместе под одним переплетом. На задачник меня не хватило, так что могу только порекомендовать какой-либо из уже существующих. Лучшее, что мне встречалось – *Н.Культин "Turbo Pascal в задачах и примерах"*. Несколько задач из него мы разберем. У этого автора есть ещё пачка задачиков для почти всех языков на свете – а кстати, здесь могла быть и ваша реклама!

Ещё пара книжек, которые читать или уже поздно или ещё рано. Уже поздно - *Б.Керниган, Ф.Плуджер "Элементы стиля программирования"*, "Радио и связь", 1984. Поздно потому, что языки программирования, в этой книге используемые, сильно повымерли. С фортраном, впрочем, встреча очень даже возможна, а вот PL/L однозначно того... Тем не менее,

это лучшая книга об этом. О чём? – о том, что в заголовке – о стиле программирования. Читать от начала до конца, потом снова – от начала до конца и снова...

Ещё рано - Лу Гринзоу “Философия программирования Windows 95/NT”, Санкт-Петербург, “Символ”, 1997 Lou Grinzo Zen of Windows 95 Programming. Рано только в том смысле, что мы тут программируем исключительно под DOS, а там, как из заглавия понятно, несколько о другом.

И ещё книги:

*Э.Дейкстра “Дисциплина программирования”*  
*У.Дал, Э.Дейкстра, К.Хоор “Структурное программирование”.*

Попробуйте почитать, даже если в них будет ничего не понятно. Как-то где-то прочитал, что какой-то профессор говорил своим студентам – Читайте, понимание придёт потом. Где-то так оно и есть.

И ещё старая книжка попопсовей:  
*Дж. Хьюз, Дж. Мичтом “Структурный подход к программированию”.*

Половина из того, что там написано, уже неправда, но всё равно прочитайте. Вторая-то половина остается в силе, а из той половины, что скончалась, узнаете, какими представлениями жил программистский мир тридцать лет назад. Способствует развитию здорового скептицизма, – какой же ерундой вам пудрят мозги сейчас.

Как вы, конечно, заметили, большинство рекомендуемых книг изданы очень-очень давно, при коммунистах. Дело в том, что Советская Власть была заинтересована в умных людях и, применительно к программированию, выпускала литературу, ориентированную на думающих разработчиков. Сейчас нужны тупые кодеры, для их массового изготовления книжки и печатают. Попадаются, конечно, полужемчужные зерна, но всё же редко. Когда подрастёте и станете маленьким начальником, прочитайте Брукс “*Мифический человеко-месяц*”. Это наше всё.

Похоже, что книги эти в последние годы не переиздавались, кроме разве что *Дисциплины программирования*. Впрочем, в Интернете все найдутся.

### **Disclaimer**

Это модное американское слово, на русский его можно перевести только целым предложением, примерно так – *Фирма веников не вяжет, или, по другому – Никто ни за что не отвечает*.

*Электрон, как и атом, неисчерпаем* © В.И.Ленин

Опечатки в этой книге тоже неисчерпаемы. Не обижайтесь и не огорчайтесь.

# Том первый, Война и немцы

## Глава 1 Просто программа

### Самая простая программа, которая ничего не делает

Пишем в редакторе, или по-другому в IDE – интегрированной среде программирования:

```
program kuku;  
begin  
end.
```

Что интересно, некоторые слова после написания сами окрашиваются в другой цвет, в зависимости от настроек редактора – скорее всего белый. В связи с трудностью отображения на белой бумаге шрифта белого цвета, у нас эти слова будут полужирными.

Это так называемые зарезервированные слова. Зарезервированность их в том, что эти слова нельзя использовать в качестве идентификаторов. Идентификатор – то же самое, что имя, но звучит гораздо лучше. Имена бывают у переменных, процедур, функций, констант, у самой программы, в конце концов.

Вместо `kuku` можно написать любое слово, главное английскими буквами. `Kuku` – это имя нашей программы.

Действительность немного сложнее – это слово может состоять из английских букв, цифр и знака подчёркивания, но начинаться должно с буквы или подчёркивания - с цифры начинаться оно не имеет права. Большие буквы, или маленькие – это никакой роли не играет – а в C++ играет! Точно так же, все остальные маленькие буквы по желанию могут быть заменены на большие, от этого ничего для компилятора не изменится.

Всё сказанное имеет очень большое значение - эти правила распространяются на все идентификаторы языка Pascal. С этого и начинается программирование. Идентификаторами являются все имена в программе. В данном случае имя только одно: `kuku` – имя программы.

Остальное должно быть в точности как написано. Ну, почти – там где стоит один пробел, можно поставить сколько угодно, это общее правило для всех языков. Но лучше не увлекаться. Но там, где стоит хотя бы один пробел – хотя бы один пробел и должен быть.

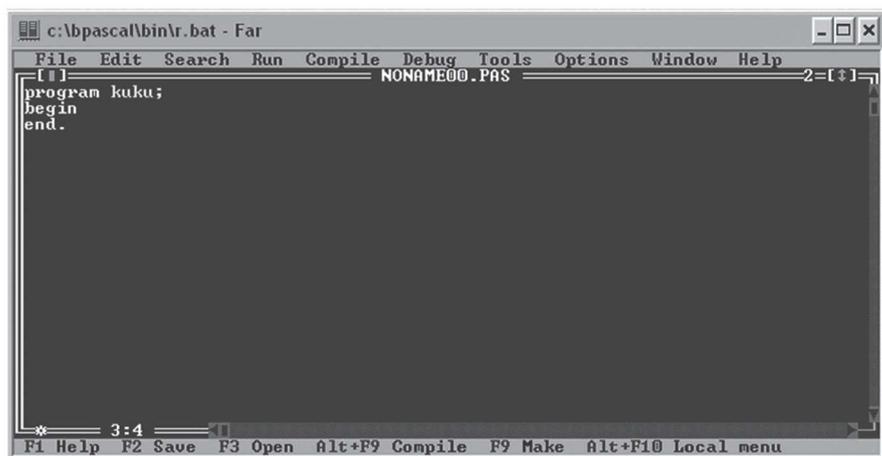
Сокровенный смысл написанной выше программы.

То, что находится между строчками **program** <её имя>; и **begin** является частью не выполняющейся, а декларативной. Говоря по другому, эта часть ничего не делает, а объясняет, как понимать написанное далее (понимать не программисту - компьютеру). А вот то, что написано между **begin** и **end** как раз что-то делает. Даже если какая-то строка не делает как будто ничего, на самом деле это не так - она указывает как, в каком порядке, при каких условиях должны выполняться остальные строки.

Да, впервые упомянуто новое слово - оператор. Что это такое?

Обычно это одно слово в языке Паскаль, которое что-то делает. Но есть, например, оператор цикла, который состоит из многих слов, и оператор присваивания, который вообще не слово. Как ни печально, про оператор все всё понимают, но объяснить не могут.

Экран сейчас должен выглядеть вот так:

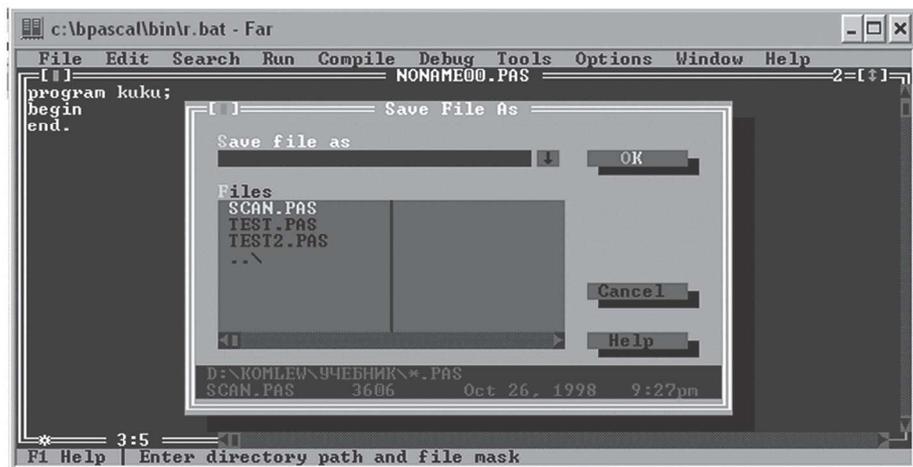


```
c:\bpascal\bin\r.bat - Far
File Edit Search Run Compile Debug Tools Options Window Help
NONAME00.PAS
program kuku;
begin
end.
```

F1 Help F2 Save F3 Open Alt+F9 Compile F9 Make Alt+F10 Local menu

Теперь нажимаем клавишу F2. Да, конечно, можно и мышью. Но ещё раз, Паскаль создавался в древние безмышьные времена и без мыши, как ни странно, действительно будет проще, хотя можно и мышью. Привыкайте.

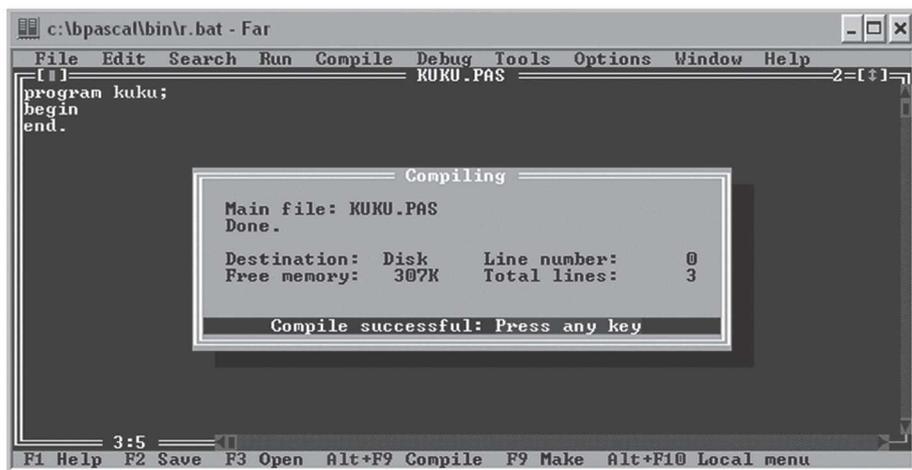
Зачем F2? Сохранить текст нашей программы  
Появится вот такое окошко:



Остаётся указать имя файла, в котором будет сохранена наша программа, и нажать клавишу **Enter**.

В каталоге, откуда мы запускали Турбо Паскаль, появился файл `kuku.pas`. Для упрощения жизни настоятельно рекомендую давать файлу то же имя, что и программе, то есть тот идентификатор, что стоит после слова **program**. И не забывать, что по техническим причинам имя должно быть не длиннее восьми символов, не содержать пробелов и так далее.

Теперь нажать клавишу **F9**. Это трансляция нашей программы – перевод её из исходного, написанного нами, текста в исполняемый машинный код. И, что на данном этапе даже важнее, проверка исходного текста на правильность и отсутствие в нём ошибок. Экран должен теперь выглядеть вот так:



Нам сообщают, что *пока* всё идёт хорошо. Теперь нажимаем - как нас собственно и просят - любую клавишу. Итак, в программе ошибок нет – да и откуда им взяться – программа-то маленькая. А теперь программу запускаем!!!

Жмём **Ctrl/F9**. Происходит быстрое мелькание экрана – и всё. Что случилось? Да ничего, собственно. Программа наша маленькая, ничего не делает. Соответственно при запуске она ничего и не сделала.

Сейчас в каталоге, где находится исходный текст нашей программы, появился файл `kuku.exe`. Точнее, появился чуть раньше, после нажатия **F9**. Его - `kuku.exe` - можно запустить на исполнение обычным способом, через проводник или FAR или Total Commander. Результат его исполнения, будет в точности тот же, что и после запуска через нажатие **Ctrl/F9**.

А теперь мы завершили работу, и вышли из Турбо Паскаля. Выйти, кстати, можно или через меню, или нажав **Alt/X**. При выходе нас, естественно, спросят, не забыли ли мы сохранить наш текст – если мы забыли. Обратите внимание на звёздочку слева внизу над **F1**. Если она есть, значит, сохраниться мы забыли.

А теперь мы запустили Турбо Паскаль снова. Если всё настроено правильно, у нас автоматически загрузится та самая программа, с которой

мы работали, и на том самом месте, где мы с ней работу прекратили. А если нет? Или если нами написано программ уже много и нам надо загрузить совсем другую? Очень просто. Нажимаем F3 и выбираем то, что нам надо.

Каждая новая открытая программа занимает в окне редактирования чуть меньшее пространство – чтобы были видны заголовки всех открытых программ. Если окошко стало уж очень маленьким, нажмите клавишу F5 – текущая программа займёт всё доступное место. Ранее открытые программы никуда не делись – они где-то там, только сзади. Вывести их на передний план можно клавишей F6 – по мере нажатия она переберёт поочередно все загруженные программы.

### Очень простая программа, которая делает хоть что-то

Если мы хотим, чтобы программа что-то делала, это что-то должно быть выражено в тексте, находящемся в исполняемой части программы – между **begin** и **end**. Например:

```
program kuku;  
begin  
    write('Au!');  
end.
```

Вместо Au!, естественно, можно написать всё, что угодно, но пока по-английски. И ещё одно ограничение – текст не должен содержать кавычек.

Если хочется писать по-русски? Придётся постараться. Сначала надо раздобыть русификатор для DOS. Рекомендую keygus Дмитрия Гуртыка. Как найти? Наберите в Яндексе keygus и выбирайте подходящий источник. Возможно, в скачанном архиве будет несколько файлов, нам понадобится только один – keygus.com. Скопируем его в c:\bpascal\bin\. Можно в другое место, только не забудьте куда. Как написано в дополнении, хотя Вы, возможно, обошлись без него, мы предполагаем, что Турбо Паскаль установлен в каталог c:\bpascal.

Теперь нам надо, чтобы перед запуском Турбо Паскаля (то есть turbo.exe) запускался keygus.com. Будем писать командный файл, известный также

как пакетный. Заходим в notepad.exe, или что-нибудь аналогично текстовое и набираем:

```
c:\bpascal\bin\keyrus.com  
c:\bpascal\bin\turbo.exe
```

Сохраняем то, что набрали под именем g.bat. Можно не g, но расширение .bat обязательно. Теперь там, где у нас вызывался turbo.exe, меняем его вызов на вызов g.bat. Пробуем. Всё должно заработать.

Внешне всё выглядит так же. Нажимаем правый Shift и пишем – теперь пишется по-русски. Снова правый Shift – пишется по-английски. При нажатом Shift'е в полноэкранном режиме вокруг экрана появляется тонкая синяя рамочка. Обратите внимание – переключение на русский язык с помощью Shift'a необходимо только для набора текста по-русски, отображаться по-русски текст будет и без этого, достаточно загруженного русификатора.

Кстати, вы освоили простейший *скриптовый* язык. Потому что язык командных файлов DOS именно им и является.

Если очень хочется кавычек, то всё, как в анекдоте, очень просто - *дайте две!* Вместо одной кавычки поставьте две. Не двойную кавычку вместо одиночной, а именно две. Видно в тексте программы будет две, но восприняты компилятором они будут не как две, а как одна. При подсчете символов в строке - об этом дальше - обе они будут восприняты как один символ. При выводе такой строки на экран видно будет тоже только одну кавычку.

Далее снова то же самое:

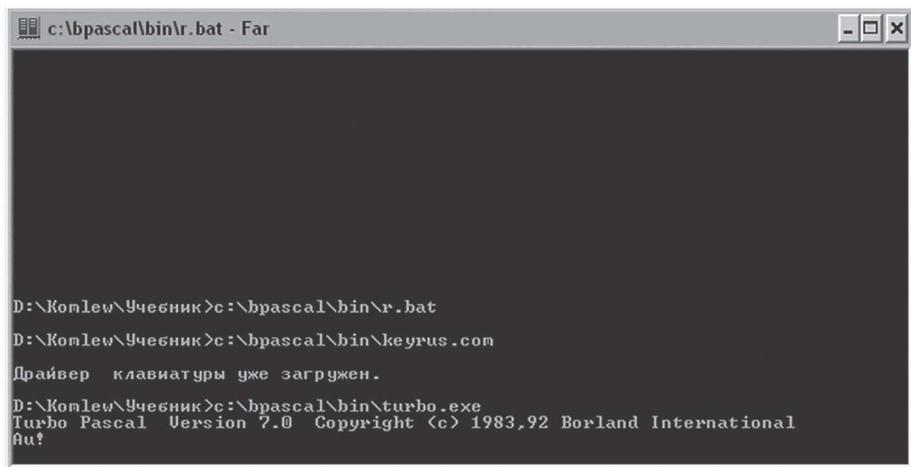
F2 – сохранить.

F9 – скомпилировать.

Ctrl/F9 – выполнить.

Опять на экране что-то промелькнуло и исчезло. Обидно, да?

Учим новое слово, в смысле сочетание клавиш – Alt/F5. Нажимаем и видим приблизительно вот такую картинку:



```
c:\bpascal\bin\r.bat - Far

D:\Konlev\Учебник>c:\bpascal\bin\r.bat
D:\Konlev\Учебник>c:\bpascal\bin\keyrus.com
Драйвер клавиатуры уже загружен.
D:\Konlev\Учебник>c:\bpascal\bin\turbo.exe
Turbo Pascal Version 7.0 Copyright (c) 1983,92 Borland International
Au?
```

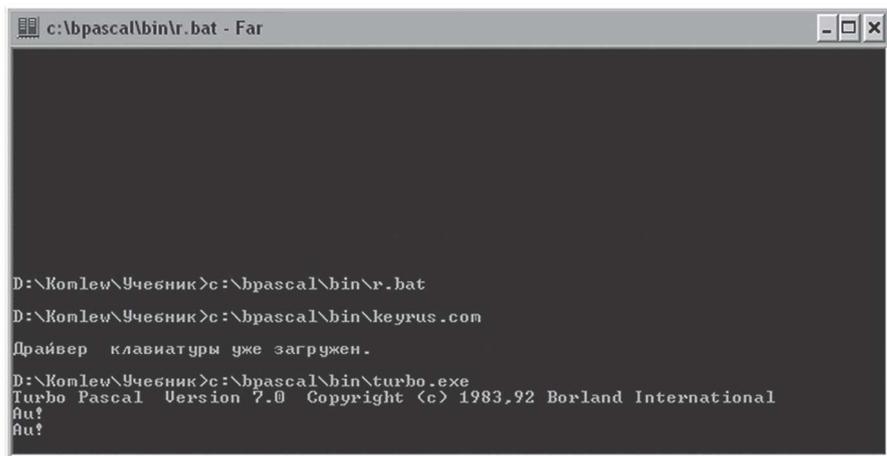
Много всего непонятного, и среди этого непонятного наше Au! Нажимаем любую клавишу и возвращаемся в привычный (уже) редактор Турбо Паскаля. Вывод из произошедшего – оператор write выводит текст на экран. Текст в кавычках, кавычки в скобках. Кавычки и скобки всегда ходят парами – если есть открывающая кавычка или скобка, где-то неподалёку должна быть и закрывающая.

### Улучшаем программу. Много новых слов

Лично мне нажимать Alt/F5 не нравится. Хотелось бы автоматизировать процесс. Добавим в программу ещё одну строчку:

```
program kuku;
begin
    write('Au!');
    readln;
end.
```

Опять те же движения- F2, F9, Ctrl/F9. И сразу, безо всяких Alt/F5 видим знакомую уже картинку. Или вот такую:



```
c:\bpascal\bin\r.bat - Far

D:\Komlew\Учебник>c:\bpascal\bin\r.bat
D:\Komlew\Учебник>c:\bpascal\bin\keyrus.com
Драйвер клавиатуры уже загружен.
D:\Komlew\Учебник>c:\bpascal\bin\turbo.exe
Turbo Pascal Version 7.0 Copyright (c) 1983,92 Borland International
Au!
Au!
```

Теперь чтобы вернуться в редактор Турбо Паскаля надо нажать клавишу **Enter**. Вывод – оператор `readln` занимается тем, что ничего не делает, ожидая нажатия клавиши **Enter** (внимание – не любой клавиши, а именно клавиши **Enter**). И ещё вывод – каждый оператор заканчивается точкой с запятой.

Ведутся глубокие философские дискуссии, ставится ли точка с запятой после оператора, или она является его неотъемлемой частью. Вообще-то, что совой об пень, что пнём об сову.

Однако что-то ещё не так. На экране кроме нашего `Au!` Куча левого мусора. Мало того, `Au!` от предыдущего запуска тоже висит на экране. А если запустим программу на выполнение ещё раз-другой, `Au!` размножатся в соответствующем количестве. Так что предлагаю при запуске программы очищать экран от всего ранее на него выведенного. Правда здесь одним словом-оператором не обойтись, новых слов придётся добавить побольше.

Программа приобретает такой вид:

```
program kuku;
uses
  Crt;
begin
  ClrScr;
  write('Au!');
```

```
    readln;  
end.
```

Запускаем программу на исполнение – и имеем чистенькую картинку с одной только нашей надписью, как и хотелось.



Волшебное слово `ClrScr` очищает экран, а чтобы оно заработало, надо написать ещё два волшебных слова – `uses Crt`. Если написать `ClrScr`, но забыть про `uses Crt`, то получим вот такое несимпатичное сообщение об ошибке:

Error 3: Unknown identifier

Означает оно, что Паскаль не знает и даже не догадывается, кто такой `ClrScr` и где его искать – а искать надо в модуле `Crt`. `ClrScr` очень похож на оператор, но это не оператор – это вызов внешней подпрограммы или процедуры - об этом позже – кому как больше нравится. Впрочем, для всех практических целей `ClrScr` можно считать оператором

Для расширения кругозора. Невразумительное слово `ClrScr` – сокращение от `Clear Screen` - очистить экран. Внешние процедуры содержатся в модулях, об этом позже. Имя модуля – `Crt`. Переводится ЭЛТ – электронно-лучевая трубка. В этом модуле содержатся подпрограммы для работы с экраном в текстовом режиме. Используемые модули

перечисляются через запятую после слова **uses**. Любознательные это слово переведут сами.

А теперь очень важное! Если кому-то это покажется очевидным, заранее извиняюсь – как показывает опыт, очень даже многим это далеко не очевидно.

Если пишем:

```
ClrScr;  
write('Au!');
```

То имеем чистый экран и надпись на нём. Что и требовалось.

А если пишем вот так:

```
write('Au!');  
ClrScr;
```

Чистый экран имеем, но, собственно, и всё.

Операторы выполняются по порядку. Сверху вниз. В порядке написания. Сначала первый, потом второй. Как ещё объяснить?

Ещё раз приношу извинения, если кому-то кажется это тривиальным. До многих почему-то не доходит, или доходит, но очень медленно. А теперь вместо

```
write('Au!');
```

напишем

```
write('Au!');  
write('Au!');
```

После запуска программы получим на экране:

```
Au!Au!
```

А теперь слегка изменим программу

```
writeln('Au!');  
write('Au!');
```

Картинка после запуска изменится на более приятную –

```
Au!  
Au!
```

После недолгого размышления приходим к выводу - отличие оператора `writeln` от оператора `write` заключается в том, что `writeln` переходит на новую строку после вывода текста. А если написать вот так:

```
writeln('Au!');  
writeln;  
write('Au!');
```

то между нашими `Au!` появится пустая строка. Запомним на всякий случай – оператор `writeln` без параметров (параметрами здесь называется то, что в скобках) обеспечивает переход на новую строку.

А теперь ещё и ещё раз – каждый оператор заканчивается точкой с запятой. Или – операторы между собой разделяются точкой с запятой. Это уж как вам больше нравится. Естественно, начинающие программисты регулярно эти точки с запятой пропускают. Программисты продвинутые тоже, хотя и не так регулярно. Компилятор это замечает и выдает тревожное сообщение “Error 85: “;”expected”. Сообщение в целом понятное, но вот курсор при этом указывает не на то место, где пропущена точка с запятой, а на начало следующего оператора. То есть, если у нас вот такой текст

```
writeln('a');  
writeln('b')  
writeln('c');
```

то курсор будет мигать в начале третьей строки:

```
writeln('a');  
writeln('b')  
_writeln('c');
```

Мигающий курсор на бумаге не отображается. Будьте внимательны.

## Весело, в цветочек

А теперь несколько дополнительных возможностей и несколько новых слов. Такая программа:

```
TextColor(Green);  
Writeln('Au!');
```

Получили текст зелёного цвета. Пустяк, а приятно. Обратите внимание, что Green пишется без кавычек. Можно задать и другие цвета. Вот полный список, или почти полный.

Black	чёрный
Blue	синий
Green	зелёный
Cyan	фигня какая-то синеватая
Red	красный
Magenta	опять фигня, но красноватая
Brown	коричневый
LightGray	светло-серый
DarkGray	темно-серый
LightBlue	светло-синий
LightGreen	светло-зелёный
LightCyan	нет слов
LightRed	розовый
LightMagenta	смотри выше
Yellow	жёлтый
White	белый

Запомните на всякий случай. Позже, в процессе освоения графики, научимся красить и в более экзотические цвета.

А если сделать вот так:

```
TextColor(Green+Blink);  
Writeln('Au!');
```

то оно будет ещё и моргать. Вот так тоже неплохо:

```
GoToXY(10,10);  
Writeln('Au!');
```

Надпись переехала в другое место экрана. Первая десятка – номер строки, на которую мы отправились. Всего строк 24. Вторая десятка – номер столбца, или, говоря по-другому, номер символа в строке. Всего их 80. Поэкспериментируйте. Строки нумеруются сверху вниз. Это, конечно, естественно – пока мы не доберемся до графики. Там нумерующиеся сверху вниз строки пикселей вызывают некоторое неудобство у обладающих начальными математическими познаниями и отличающих ось абсцисс от оси координат – извините за умные слова.

### **И кое-что ещё**

Текст, заключённый в фигурные скобки, - это комментарии. Компилятор этот текст игнорирует, как будто его нет совсем. То есть пишется он не для компьютера, а для человека – в первую очередь для вас, когда завтра вы забудете, что насочиняли вчера.

```
{ Переходим в центр экрана}  
GoToXY(40,12);  
Writeln('Au!');
```

## Глава 2 Переменные

### Что такое и зачем

А теперь о главном. Три вещи, по нарастающей сложности, которые надо понять зарождающемуся программисту - переменные, массивы и указатели. Всё остальное - сущая ерунда. Сейчас будут переменные. Пишем вот такую программу:

```
program kuku;  
  uses  
    Crt;  
  var  
    x           : integer;  
begin  
  ClrScr;  
end.
```

Что мы сделали? Объявили переменную. Имя у неё X. Тип у неё целый, то есть переменная эта может содержать только целые значения. Непонятно? Мы ещё на этом задержимся и задумаемся. А теперь делаем вот так:

```
x:=5;  
writeln(x);
```

Транслируем, запускаем, смотрим. На экране получилось вот что –

5

А теперь медленно повторяю : мы  
- объявили переменную целого типа  
- присвоили ей значение  
- вывели значение переменной.

А что такое переменная, собственно? Вопрос из категории – а что такое электричество?

Переменная – ключевое понятие программирования. Без неё не обходится ни один язык. Ну, почти ни один. (Здесь, и далее, а также и прежде под языками я понимаю только и исключительно языки программирования). Переменная имеет две составляющих – имя и значение. Как известно, на

заборе что-то там написано, а лежат там дрова. Так вот, то, что написано – это имя переменной, а дрова – это значение.

В большинстве языков переменная имеет ещё и третью составляющую – тип. В нашем случае тип переменной – целое число. То есть поместить в неё можно только целое число, дробное – никак, не говоря уже о всякой экзотике. Но к экзотике ещё вернёмся, попозже.

Для любознательных - тип переменной `integer` позволяет хранить только значения от +32768 до -32767. Причина этого - размер два байта. Если хочется простора, используйте вместо `integer` тип `longint`. Он занимает четыре байта. Посчитайте на досуге, какое максимальное значение в него поместится. Это сложно, разумеется.

`x : integer;` - это объявление переменной. Слева от двоеточия имя переменной. Часто говорят об идентификаторе переменной. Это то же самое, что имя. Имя может быть написано большими буквами или маленькими – Паскалю всё равно. Два подряд идущих неразъёмных символа “:=” называются оператором присваивания. Слева от него имя переменной, справа – значение, которой ей присваивается.

А теперь сделаем с переменными что-нибудь полезное. Для начала объявим три переменные:

```
var
  x, y, z          : integer;
```

Двум из них присвоим значения:

```
x:=5;
y:=2;
```

А почему только двум? Потому что значение третьей переменной мы считаем. На самом деле не мы, конечно, а компьютер.

```
x:=5;
y:=2;
z:=x+y;
writeln(z);
```

Получили ожидаемый результат – на экране красуется семёрка. Теперь быстро осваиваем вычитание и умножение.

```
z:=x-y;
writeln(z);
```

```
z:=x*y;
writeln(z);
```

Соответственно, в результате имеем два и десять. Осталось четвертое действие – деление. С ним сложнее и намного. Почему? Потому что числа у нас целые, а результат деления целым быть не обязан – от деления пятерки на двойку получим два с половиной. Что делать? Вспомнить чудное детство в первом классе – деление нацело с остатком. Делим пять на два – получаем два и один в остатке. Попрактикуйтесь в этом деле – как ни странно, некоторые напрочь это забыли. Так что вместо одной операции получаем две – деление нацело и получение остатка.

Сначала деление нацело.

```
x:=5;
y:=2;
z:=x div y;
writeln(z);
```

В результате имеем два. А теперь получение остатка

```
x:=5;
y:=2;
z:=x mod y;
writeln(z);
```

В ответе получаем единицу. Задача сложнее – разобрать двузначное число на составляющие его цифры.

```
var
  x, c1, c2      : integer;
begin
  x:=85;
  c1:=x div 10;
  c2:=x mod 10;
  writeln(c1);
  writeln(c2);
end.
```

Маленькое новшество – имена переменных у нас уже не из одного символа, а из двух, причём второй символ – цифра. Главное, чтобы имя начиналось с буквы. И само собой, чтобы имена не повторялись. Обратите внимание на особый смысл выражения  $x \bmod 2$ . Если оно равно нулю, то число  $x$  чётное, если единице – то нечётное. Это пригодится далее.

Переменные можно было бы объявить и каждую в отдельной строке:

```
var
  x           : integer;
  c1          : integer;
  x2          : integer;
```

Но мы написали так, как короче. Аналогично можно сэкономить и на операторах `writeln`, объединив два в один. Пишем :

```
writeln(c1,c2);
```

Запускаем программу на выполнение и получаем не совсем то, что бы хотелось.

85

Циферки слиплись. С одной стороны, всё правильно – вот восьмерка, вот пятерка – но хотелось бы как-то видеть, где кончается одно число, и где начинается другое. Напишем вот так:

```
writeln(c1:3,c2:3);
```

Гораздо лучше. Тройка означает, что мы требуем выделения для вывода числа трех символов. Число у нас всего из одной цифры, так что впереди перед ней появились два пробела.

А почему мы должны, глядя на результат работы программы, догадываться, где у какой переменной значение? Делаем так:

```
writeln('c1=', c1:3, 'c2=', c2:3);
```

То, что в кавычках, будет выведено *как есть*, это мы уже проходили. Долго думаем о тонкой разнице между `c1` в кавычках и `c1` без кавычек. В кавычках это просто бессмысленный набор символов. Компьютер

выведет его так, как он написан. А без кавычек это – имя переменной, и выведено будет её значение. Если это сразу кажется понятным и очевидным – поздравляю...

Вы, конечно, уже заметили, что вывод всё равно не совсем такой, как хотелось бы – опять чего-то слиплось. Но теперь уж постарайтесь справиться сами.

## Ввод и вывод

Напишем программу возведения числа в квадрат.

```
program Square;
  uses
    Crt;
  var
    x, x2      : integer;
begin
  x:=5;
  x2:=x*x;
  writeln('x2=',x2);
  readln;
end.
```

Всё хорошо, всё работает – пятью пять двадцать пять. А если надо шестью шесть тридцать шесть? Менять текст программы, снова компилировать - кажется как-то не очень правильным. Хотелось бы, чтобы программа при запуске спросила, а какое, собственно, число нам надо возвести в квадрат.

Легко. Вместо

```
x:=5;
```

пишем

```
readln(x).
```

Теперь при запуске программы видим чёрный экран. Программа упорно ждёт, когда мы наберем на клавиатуре число, хотя бы ту же пятерку, и нажмём Enter. Тоже как-то не очень удобно. Мы-то, конечно, знаем, что надо делать, а если вдруг кто-то непонятливый к компьютеру подойдёт? Улучшаем программу. Пишем:

```
write('x = ');  
readln(x).
```

Обратите внимание, что написано не `writeln`, а именно `write`. Исключительно из эстетических соображений. Иначе мы бы перешли на новую строку и уже там набирали вводимое число. Число ввелось бы, конечно, но выглядит это как-то не очень аккуратно. Ещё обратите внимание на пробел после равенства. Исключительно из тех же эстетических соображений. В результате наконец-то получили первую хоть сколько-то полезную программу.

Делаем вывод - порядочная программа должна:

- ввести данные - предварительно предложив их, данные, ввести;
- что-то с ними сделать;
- вывести результат.

А теперь немного о грустном. Мы пока что работаем только с целыми числами. Если попытаться ввести вместо пяти пять с половиной, результат будет катастрофическим - попробуйте. Если в процессе ввода промахнуться мимо нужной клавиши и набрать вместо цифры букву – та же самая катастрофа. Что делать? С непопаданием по клавишам – пока ничего. Внимательнее смотрите на клавиатуру. Разберёмся с этой проблемой попозже.

## Дроби

Так что насчёт пяти с половиной? Это не просто, а очень просто. Меняем всего одну строку. Вместо

```
var  
  x      : integer;
```

пишем

```
var  
  x      : single;
```

и наступает полное дробное счастье – ну почти. Почему счастье не совсем полное?

Single - дробное число, занимающее четыре байта. Дробные числа обычно называют плавающими или числами с плавающей точкой. Когда-то, давным-давно, были ещё и числа с фиксированной точкой, например, три знака до точки, два после и никак иначе. Числа с плавающей точкой отличались от чисел с фиксированной точкой тем, что помнили определенное количество знаков (single - шесть, семь, как повезёт) - а точка могла стоять где угодно. То есть, плавающее число помнит шесть знаков - 123456, но это может быть 123.456 или 0.0123456 или 123456000. А седьмой знак тут уже никак поместится не может.

Если хочется большей точности, single можно заменить на double (восемь байтов). Как очевидно из английского языка, число значащих цифр удвоится. В новых/старых учебниках часто встречается также single - шестибайтовое плавающее. Это атавизм - забудьте. Обратите внимание, что у целых переменных при изменении количества байт на число меняется максимальная величина, которое это число может содержать, а у дробных – ещё и точность представления числа.

Во-первых, дробные числа вводятся (и выводятся) с точкой в качестве разделителя. То есть, пресловутые пять с половиной будут выглядеть так: 5.5, а не 5,5. Потом, в реальной жизни реального программиста, всё будет гораздо сложнее, но пока что именно так.

В качестве разделителя целой и дробной части в Турбо Паскале используется всегда точка, независимо от того, какой разделитель установлен в Windows.

Во-вторых, оператор `writeln(x)`, оставленный без присмотра, выведет на экран что-то страшное и ужасное. На самом деле, никакое оно не страшное и, даже, где-то полезное, но разбираться с ним (пока, по крайней мере) не будем. Будем улучшать. Напишем

```
writeln(x:8:2);
```

Очень напоминает вывод целых чисел – там тоже первая и единственная цифра задавала общее количество выводимых цифр. Если реально цифр было меньше, спереди выводимое число дополнялось пробелами. Но сейчас для нас интереснее вторая цифра. Двойка указывает, сколько знаков после десятичной точки должно быть выведено. И опять очень

сложный момент. Восьмерка здесь – сколько знаков всего выводится – до точки плюс после точки плюс один знак на саму точку.

А теперь у нас появилась ещё одна арифметическая операция – обычное деление, не нацело. Обозначается оно так:

```
x:=y/z;
```

сейчас будет ещё одна программа, с небольшим отличием. В ней используется константа, то есть число, значение которого мы заранее знаем и, в процессе выполнения программы, измениться это значение не может. Чем она - константа - собственно, и отличается от переменной. А чем она отличается от просто числа? – тем, что имеет имя.

Если мне не изменяет память, константы делятся на константы именованные и неименованные. То есть, термины могут быть другими, но смысл остаётся. Сейчас мы встретимся с неименованной константой, а до именованных доберёмся в процессе освоения массивов.

```
program Circle;
  uses
    Crt;
  var
    R,S      : single;
begin
  writeln( 'Radius = ');
  readln(R);
  S:=3.14*R*R;
  writeln('S=',S:8:2);
  readln;
end.
```

Здесь 3.14 – число  $\pi$  – константа. На всякий случай, если кто-то не понял, мы только что вычислили площадь круга.

Идею поняли. А теперь – быстро пишем программу, которая считает, что мы заработаем, если положить в банк X рублей под Y процентов на один год. Для тех, кто совсем никак, дальше искомая программа. Но всё-таки, постарайтесь сначала написать сами.

```
program Money;
  uses
    Crt;
```

```

var
    Rub, rate, HowMany : single;
begin
    writeln( 'Rubles = ');
    readln(Rub);
    writeln( 'Rate = ');
    readln(rate);

    HowMany:=Rub + (Rub*rate)/100;
    writeln('Result=',HowMany:8:2);
    readln;
end.

```

Ещё немного математики. В школе кроме четырех действий что-то говорилось и про пятое – логарифмирование. Вспоминается также и тригонометрия с синусами, косинусами и прочими тригонометрическими функциями. В Паскале это всё есть и выглядит очень похоже на обычные математические формулы:

```

y:=Sin(x);
y:=Cos(x);
y:=Ln(x);

```

И называется это так же, как и в математике – функции. В Паскале есть много уже готовых функций, а попозже будем осваивать и самостоятельное их изготовление.

И ещё о важном. Например, у нас есть объявление переменных

```

var
    x      : single;
    n      : integer;

```

В начале нашей программы следуют вот такие операторы присваивания:

```

x:=3.5;
n:=4;

```

До сих пор всё хорошо и понятно.

Теперь, если мы напишем

```

x:=n;

```

то по-прежнему всё будет хорошо. Но если мы напишем

```
n := x;
```

то мы даже не дойдем до этапа выполнения программы. В процессе трансляции нам сообщат, что мы глубоко не правы:

### Error 26: Type mismatch

Это надо постараться или запомнить, или как-то понять. Чтобы запомнить – учим мантру: дробному целое присвоить можно, целому дробное присвоить нельзя. Постараться понять – на уровне для блондинок: у дробного есть ящички под целые и дробные части, а у целого – только один ящичек – для части целой. Поэтому, если мы пишем целое в дробное, то оно из одного ящичка в два влезет, а если, наоборот, из двух ящичков в один – ну никак.

А вообще всё даже проще – в Паскале в принципе нельзя присваивать друг другу переменные разных типов. Таков Великий Замысел Отцов-Основателей. Единственная данная ими поблажка – дробному числу можно присвоить целое. Вот и всё.

## Глава 3 Условные операторы

### Что такое и зачем

Программа выполняется сверху вниз, я уже говорил. А если надо не совсем сверху вниз, а зигзагом? Для этого у нас есть условные операторы. Условный оператор – это очень просто. Покажем на примере. Программа определяет, положительное ли у нас число, или отрицательное, и выдаёт соответствующее сообщение.

```
program Plus;
  uses
    Crt;
  var
    num      : integer;
begin
  writeln( 'num = ');
  readln(num);
  if num > 0 then writeln('positive');
  if num < 0 then writeln('negative');
  readln;
end.
```

На интуитивном уровне вроде понятно. Теперь подробнее – из чего состоит условный оператор.

Слово **IF** – в начале. Дальше условие. Условие у нас пока что самое простое – два числовых выражения, разделенные знаком сравнения. Знаки бывают вот такие:

- > < - это понятно, больше и меньше
- >= <= - больше или равно, меньше или равно
- = - проверка на равенство
- <> - а вот таким хитрым образом обозначается неравенство.

Дальше слово **THEN**. А после него оператор. И оператор этот будет выполнен только тогда, когда выполняется условие, заданное нами (условие – это то, что между **IF** и **THEN**). Разумеется, оператор не должен обязательно быть оператором вывода, а может быть любым – ну почти.

А теперь вспоминаем арифметический оператор **mod** и пишем программу проверки числа на чётность. Кто быстро справился, вспоминает ещё и **div**

и пишет программу проверки номера трамвайного шестизначного билета на счастливость – в смысле, чтобы сумма первых трёх цифр равнялась сумме трёх последних.

### Усложняем

Два *а если?*. А если у нас не одно условие, а несколько – например, не просто больше нуля, а от нуля до ста? А если у нас должен выполняться не один оператор, а несколько – например, сообщение о положительности числа покрасить в красный цвет, а об отрицательности – в синий?

Сначала ответ на второй вопрос. Вместо

```
if num>0 then writeln('positive');
```

пишем

```
if num>0 then begin
  TextColor(Red);
  writeln('positive');
end;
```

Появилась новая конструкция языка, которая в дальнейшем будет употребляться очень-очень часто. Там где можно употребить один оператор, всегда – почти - можно употребить несколько, поставив перед первым **begin**, а после последнего оператора **end**!. Перечитал и испугался – точка здесь исключительно в предложении, а не в синтаксисе Паскаля!

По научному это называется операторные скобки. Операторы, заключенные внутри пары **begin-end** могут быть любыми, в том числе условными операторами, содержащими **begin-end**, внутри которых...

И ещё. До сих пор у нас был только один **end** в конце программы, после которого стояла точка. После нашего нового **end**'а ставится точка с запятой. Это как раз типично, а **end** с точкой – аномалия. Разумеется, наш исходный условный оператор

```
if num>0 then writeln('positive');
```

можно записать и так

```
if num>0 then begin
  writeln('positive');
end;
```

Кому что больше нравится.

А теперь что делать, если нам недостаточно простой проверки на положительность, а хочется чего-нибудь эдакого – например положительное, но не больше ста.

Важно! – если у нас условий несколько - два, например - каждое из них должно быть взято в скобки! В особенности это важно потому, что при отсутствии скобок сообщение компилятора об ошибке является неадекватным и вводящим в заблуждение, понять по нему, в чём дело трудно. Разумеется, у компилятора на это есть весьма уважительные причины, но я их не знаю.

Итак, имеем два условия:

```
{num>0} {num<=100}.
```

Осталось объединить два простых условия в одно сложное:

```
{num>0} and {num<=100}
```

**and** обозначает, что мы требуем одновременного выполнения обоих условий. В конечном итоге получаем:

```
if {num>0} and {num<=100} then begin
  writeln('Ok');
end;
```

А какие ещё случаи бывают?

**or** - выполняется или первое условие, или второе, или оба сразу. Заметьте, это несколько отличается от бытового подхода – когда или-или, или одно или другое, но оба сразу – ни-ни. У нас можно и оба.

**not** - отрицание. Эквивалентно замене условия на противоположное. То есть **not**( $x > 100$ ) то же самое, что ( $x \leq 100$ ). Сложные условия также можно объединять между собой, получая условия, всё сложнее и сложнее.

Например (случай средней тяжести) - проверка, является ли год високосным:

```
{year mod 4 = 0} and {(year mod 100 <> 0) or {(year div 100) mod 4 = 0}}.
```

Если условие ещё сложнее, что-то в программе спроектировано и запрограммировано не так.

### Окончательно усложняем

А может, и упрощаем. Пришёл к нам заказчик. И заказал программу проверки делимости числа на семь. И сочинили мы вот такой шедевр:

```
program Divis;
  uses
    Crt;
  var
    num      : integer;
begin
  writeln( 'num = ');
  readln(num);
  if (num mod 7) = 0 then writeln('divisible');
  if (num mod 7) <> 0 then writeln('not divisible');
  readln;
end.
```

Всё хорошо, всё работает и даже, что удивительно, работает как будто правильно. Но мрачный образ заказчика введён не случайно. На следующий день после завершения проекта он придёт и объявит, что он теперь хочет проверять делимость не на семь, а на восемь. Заказчик всегда прав – деньги у него.

А дальше как всегда. Меняем семерку на восьмерку, в смысле в первой строке меняем, во второй забываем. Потом три дня ищем, где ошибка. Это нормально, это жизнь. Общее правило - если в программе одно число встречается хотя бы дважды - да хотя бы и однажды - оно в обязательном порядке меняется на именованную константу. В обязательном, в том смысле, что не выполняющие это программисты подлежат немедленному отстрелу. Исключением является начало цикла for i:=1, но о циклах мы ещё ничего не знаем. И вообще. числа ноль и единица (0 и 1) допускаются без угрозы расстрела, хотя есть у меня некоторые сомнения...

Увлекательную тему устранения неполноценных программистов я постараюсь полнее раскрыть позже, а пока вернёмся к нашей программе.

Использование констант программу значительно улучшит, но в нашем случае надо начать с другого. Обратите внимание, что условия в наших двух условных операторах таковы, что выполняться будет всегда только один оператор – или число делится на семь или нет. Для таких или-или случаев предусмотрена особая – полная - форма условного оператора. Сначала теория. Выглядит это так:

**IF** условие **THEN** оператор **ELSE** оператор;

И два наших условных оператора превращаются в один:

```
if (num mod 7) = 0
  then writeln('divisible')
  else writeln('not divisible');
```

Если условие выполняется – работает оператор после **then**, не выполняется - оператор после **else**. Важно! После первого оператора (который после) **then** точка с запятой не ставится! Вам и не хочется? Странно, а всем хочется...

Ну и самый общий случай – когда в зависимости от условия выполняется не один оператор, а несколько. Вот так:

```
if (num mod 7) = 0 then begin
  TextColor(Green);
  writeln('divisible');
end
else begin
  TextColor(Red);
  writeln('not divisible');
end;
```

Поразмышляйте над расстановкой точек с запятой.

### Небольшая программка и кое-что ещё

Как-то и не в тему в этой главе, но оно и в любой главе будет не в тему. Так что расскажем здесь. Встречайте – случайные числа. А зачем, собственно?

Напишем программу проверки устного счёта – спрашиваем, сколько будет два плюс три, если клиент отвечает пять, одобряем, если не пять – порицаем. Программа, без конца спрашивающая про два плюс три, производит несколько утомительное впечатление – вот тут и пригодятся случайные числа.

Сначала пишем монотонно-однообразный вариант, потом улучшаем.

```
program Test;
  uses
    Crt;
  var
    a,b,sum          : integer;
begin
  ClrScr;
  a:=2;
  b:=3;
  write('How many is `a,+` ,b,` `');
  readln(sum);
  if sum=a+b
    then writeln('Ok')
    else writeln('Very very bad');

  readln;
end.
```

Вместо

```
a:=2;
b:=3;
```

пишем

```
a:=Random(100);
b:=Random(100);
```

Random(N) – функция, возвращающая случайное целое число в диапазоне от 0 до N-1, в нашем случае от 0 до 99.

Обратите внимание – не до 100, а до 99. Для нашей программы это совершенно безразлично, но часто становится принципиально важным. Позже к этому ещё вернёмся.

И ещё – в начало программы, где-нибудь в районе ClrScr надо добавить волшебное слово Randomize; - без него Random будет возвращать одни нули. Randomize надо вызвать ровно один раз до самого первого

использования Random. В конечном итоге программа приобретает такой вид:

```
program Test;
uses
  Crt;
var
  a,b,sum          : integer;
begin
  ClrScr;
  Randomize;
  a:=Random(100);
  b:=Random(100);
  write('How many is ',a,'+',b,' ');
  readln(sum);
  if sum=a+b
    then writeln('Ok')
    else writeln('Very very bad');

  readln;
end.
```

Каждый раз при запуске программы на экране будет появляться новый пример, но только один. К программе этой мы позже ещё вернёмся, с целью её дальнейшего улучшения и развития.

На чём попрактиковаться? Маленькая, но вредная задача. Решите квадратное уравнение. На вход программы – три коэффициента А,В,С. На выходе – корни уравнения. Или приговор об их отсутствии. Формулу найдите сами. Или вспомните.

Справились? Теперь прогоните программу на следующих тестах. Далее коэффициенты А,В,С и результат, который вы должны получить.

2	5	2	$X_1 = -2$	$X_2 = -0.5$
2	4	2	$X_{1,2} = -1$	
2	1	2	корней нет	
0	3	6	$X_{1,2} = -2$	
0	0	3	корней нет	
0	0	0	тождество	

## Глава 4, очень простая Немного графики

### Начальные заклинания

Глава действительно очень простая – будем рисовать картинки. Но сначала придётся выучить несколько заклинаний, понимать их необязательно - это, конечно, непедагогично, я понимаю.

Чуть-чуть теории – монитор отображает данные или в текстовом режиме, или в графическом. Текстовый режим на самом деле, конечно, тоже в глубине души графический, но для нас это совершенно неважно.

Давным-давно назад мы написали программу, которая ничего не делала. Не делала она ничего в текстовом режиме. А теперь напишем программу, ничего не делающую в графическом режиме.

```
program Gr;
  uses
    Graph;
  var
    driver, mode : integer;
begin
  driver:=VGA;
  mode:=VGAHi;
  InitGraph( driver, mode, 'c:\bpascal\bgi');

  { а вот тут что-нибудь потом нарисуем }

  readln;
  CloseGraph;
end.
```

Особое внимание вот этой строке:

```
InitGraph( driver, mode, 'c:\bpascal\bgi');
```

А точнее, тому, что в кавычках - 'c:\bpascal\bgi'.

Строка эта может быть другой в зависимости от места, куда вы установили Турбо Паскаль. Здесь должен быть прописан полный путь до подкаталога bgi, находящегося в каталоге, в который вы установили Турбо Паскаль – вы же помните, куда его установили? Вы вообще его установили? В подкаталоге bgi должен находиться файл egavga.bgi – его

наша программа будет искать при запуске и, если не найдёт, выдаст соответствующее страшное сообщение.

Есть вариант – берём файл `egavga.bgi` и кидаем его в тот каталог, где размещается наша программа – её исходные тексты и, главное, исполняемый файл. А соответствующую строку программы меняем вот так:

```
InitGraph( driver, mode, '' );
```

Если Вы захотите передать кому-то программу, работающую в графическом режиме, то вместе с исполняемым файлом надо в тот же каталог поместить и файл `egavga.bgi`. Вызов `InitGraph` в этом случае должен быть точно таким, как написано выше – с пустой строкой.

Вообще-то при большом желании можно не таскать с программой файл `egavga.bgi`, а вкрутить его внутрь исполняемого файла. Но о таких подвигах в какой-нибудь другой книге.

`InitGraph` переводит весь вывод из текстового режима в графический режим. После этого можно рисовать. `CloseGraph` возвращает нас в текстовый режим. Всё с трудом нарисованное при этом пропадает. `readln` выполняет привычную функцию – ждёт нажатия `Enter`, чтобы мы могли взглядеться на нашу картинку.

Слова *весь вывод* значат именно то, что они значат и имеют слегка неприятный побочный эффект – текстовые операторы вывода `write` и `writeln` работать перестают.

### Точки, линии и окружности

Как и положено в геометрии, начнём с точек. Занятие малопродуктивное с прикладной точки зрения и малополезное с точки зрения педагогической. Перевозу – в жизни рисовать точки вряд ли придётся, а рисуются они совершенно непохожим на всё остальное способом, так что ничему полезному не научимся. Но раз надо, значит надо.

Сначала о главном. Размер экрана у нас – 640 по горизонтали, 480 по вертикали. Точки нумеруются от 0 до 639 и от 0 до 479 соответственно. По горизонтали нумерация слева направо, по вертикали сверху вниз.

Одна экранная точка, зажжённая или нет, часто называется пиксел - или пиксель.

Рисуем зелёную точку приблизительно посередине экрана:

```
PutPixel(320,240, Green);
```

Первое число – координата по X - абсцисса, второе – координата по Y - ордината. Дальше цвет. С цветами всё так же, как и в текстовом режиме. Вот, собственно, и всё о точках.

Переходим к линиям. Если точка на плоскости задаётся двумя числами, то линия, как известно, четырьмя.

```
Line(100,100, 200,200);
```

Получили линию, идущую направо вниз под сорок пять градусов. Первые два числа – координаты начала линии, вторые два – координаты конца линии. Если написать

`Line(200,200, 100,100)` – получим абсолютно тот же результат - какая разница, с какого конца линию начинать рисовать.

А теперь квадрат, с левым верхним углом в точке (100,100) и стороной 100. Сначала верхняя сторона – горизонтальная. Раз линия горизонтальная, то координата Y неизменна, меняется только X.

```
Line(100,100, 200,100);
```

Теперь правая – вертикальная. Наоборот – X неизменна, Y меняется. А начинается она там, где кончается верхняя сторона – в точке (200,100).

```
Line(200,100, 200,200);
```

Продолжаем и в итоге имеем искомый квадрат:

```
Line(100,100, 200,100);  
Line(200,100, 200,200);  
Line(200,200, 100,200);  
Line(100,200, 100,100);
```

Квадрат у нас белый. Хотелось бы чего-нибудь поживее. Вспоминаем текстовый режим и команду `TextColor`. Всё аналогично, только команда называется `SetColor`. Цвета называются точно так же. И точно так же, заданный цвет действует только на то, что рисуется после задания цвета, а не до.

Зелёный квадрат:

```
SetColor(Green);  
Line(100,100, 200,100);  
Line(200,100, 200,200);  
Line(200,200, 100,200);  
Line(100,200, 100,100);
```

Задание – самостоятельно нарисовать красный треугольник.

Теперь обещанная в заголовке окружность - вспоминаем, чем окружность отличается от круга. Рисуем окружность в центре экрана радиусом 100.

```
Circle(320,240, 100);
```

Первые два числа – координаты центра, третье число – радиус.

А теперь голубенькую:

```
SetColor(Blue);  
Circle(320,240, 100);
```

Всё очень просто.

## Прямоугольнички и кружочки

Помните, с чего мы начали при выводе текста? Сначала что-то вывели, намусорили, потом решили почистить экран. Сейчас то же самое – только попутно освоим кое-что полезное.

Пишем:

```
Bar(200,200, 400,300);
```

Получили заполненный белым цветом прямоугольник размером 200 по горизонтали и 100 по вертикали с верхним левым углом в точке (200,200). Первая пара чисел – координаты левого верхнего угла, это всем понятно. Не всем понятно, что вторая пара чисел это не размеры сторон нашего прямоугольника, а тоже координаты, но только правого нижнего угла. Теперь будем красить. К сожалению, `SetColor(Green)` нам теперь не поможет. Чтобы закрасить прямоугольник, придётся написать оператор посложнее:

```
SetFillStyle(SolidFill, Green);
```

Второй параметр задаёт цвет, а первый? Первый - это стиль заполнения – в крапинку, в горошек, в данном случае – без затей, сплошным зелёным цветом. А чтобы почистить весь экран до исходного черного цвета, пишем:

```
SetFillStyle( SolidFill, Black);  
Bar( 0,0, 639,479);
```

Можно очистить экран и более правильным способом, но так нагляднее и понятнее. Мне так кажется. `SolidFill`, как и следует ожидать, константа. Её значение – 1. Некоторые ленивые товарищи так и пишут `SolidFill(1, Black)`. Есть ещё и суперленивые, которым и `Black` тяжело написать, но таких мы безусловно осуждаем.

Обещанных в заголовке кружочков не будет. В этом месте Турбо Паскаль несколько ассиметричен, и, вместо кружочков, предлагает эллипсы. Если очень хочется именно кружочков, вспоминаем – круг, это такой эллипс, у которого обе оси равны. Рисуем:

`FillEllipse(320, 240, 200,100)` – большой такой эллипс, вытянутый по горизонтали.

`FillEllipse(320,240, 100,100)` – частный случай эллипса – круг.

И то и другое, разумеется, белое. Покраска круга немного отличается от покраски прямоугольника. `SetColor` покрасит контур круга (окружность), а за внутренности отвечает `SetFillStyle`. Попрактикуйтесь.

В Delphi в таких случаях используются метафоры пера и кисти. Перо отвечает за проведение линий, а кисть закрашивает сразу поверхности.

## Красивые букочки

Чтобы не мучаться, технологию вывода текста – в графическом режиме, не в текстовом - излагаю сразу.

```
SetColor(Green)
SetTextStyle( 0, HorizDir, 1);
OutTextXY( 100,100, 'Lala');
```

Первая строка очень даже знакомая – именно таким способом задаётся цвет текста. Вторая сложнее. Первый параметр – номер шрифта - от нуля до пяти. На самом деле, есть, конечно, уже заранее определённые константы, и выбранный шрифт можно задать словами. Например:

```
DefaultFont    = 0;
TriplexFont    = 1;
SmallFont      = 2;
SansSerifFont = 3;
GothicFont     = 4;
```

Второй параметр объясняет, что сегодня мы захотели вывести текст слева направо, а не сверху вниз. Если всё же хочется сверху вниз, то второй параметр будет VertDir.

Третий параметр – размер шрифта. Диапазон изменения для каждого шрифта свой. Эффект влияния для каждого шрифта – тоже свой, то есть первый шрифт второго размера очень просто может оказаться больше второго шрифта пятого размера.

Последняя строка нашей программы – собственно вывод текста. Третий параметр последней строки понятен сразу – это сам текст, а вот первые два требуют умственного напряжения – это левый верхний угол прямоугольника, в который вписан выводимый текст. Или, говоря проще, левый верхний угол нашего текста. Текст пока только по-английски.

Конечно, если очень хочется, то можно и по-русски. Но процедура немного сложнее, нежели для вывода по-русски текста в текстовом

режиме. Точнее, если мы согласны на шрифт номер ноль(DefaultFont), то ничего дополнительно делать не надо. Достаточно, чтобы был загружен keugus.com. Далее всё как обычно, просто в OutTextXY пишем по-русски. В чем засада? – буковки очень страшные получаются, особенно если размер шрифта задать побольше.

Для того чтобы было красиво, надо раздобыть файлы с русскими шрифтами. Раздобыть их можно, понятное дело, в Интернете. Наберите в Яндексе или в Гугле что-то вроде *Turbo Pascal шрифты русские* и тщательно изучите результаты. По нахождению искомых файлов, всё что требуется – просто заменить имеющиеся шрифты на шрифты найденные. Найденных может оказаться больше, чем исходных. Это нормально и даже хорошо.

Вы уже знаете, что если хочется отдать кому-то на чужой компьютер свою программу для исполнения, то в случае программы, работающей в текстовом режиме, достаточно передать исполняемый файл. Если программа работает в графическом режиме, надо в комплект добавить файл egavga.bgi. Это необходимо, но не всегда достаточно. Если программа ещё и выводит текст в графическом режиме, надо добавить файлы шрифтов. Шрифт DefaultFont файла не требует, а для каждого из остальных шрифтов требуется соответствующий ему файл со шрифтом. Можно не думать головой, а просто скопировать все файлы с расширением .chg из каталога c:\bpascal\bgi\.

Теперь о русском языке. Если используется опять-таки шрифт DefaultFont, то предварительно должен быть загружен русификатор. Для остальных шрифтов русификатор не нужен, но, как это не банально, шрифты должны быть русскими.

Ну и опять-таки, если ну очень хочется, то файлы шрифтов, как и файл egavga.bgi можно вкрутить внутрь исполняемого файла.

### Что там ещё осталось?

Ещё кое-что полезное вдогонку ранее освоенному. Линию уже выводили. И цвет её задавали. А теперь толщина линии и стиль – красивое слово.

```
SetLineStyle( SolidLn, 0, NormWidth);
```

Первый параметр – стиль линии – сплошная, пунктирная... Вместо SolidLn(сплошная) можно написать 0. Возможный диапазон – от нуля до трёх - точнее до четырёх, но четвёрка нам не нужна. Второй параметр сейчас и в ближайшем будущем строго равен нулю. Третий параметр – толщина линии. Или NormWidth – тонкая, или ThickWidth – толстая. ограничение – стиль линии учитывается, только если линия тонкая. Если линия толстая – то без баловства, только сплошная. Это не я придумал, это такое ограничение.

И ещё о цвете. Зелёный прямоугольник – освоили. Зелёный эллипс – освоили. А зелёный треугольник? С треугольником сложнее. Зато, освоив треугольник, освоим и всё остальное, сколь угодно многоугольное и без углов вообще. Знакомьтесь – FloodFill. Возвращаемся к когда-то нарисованному нами треугольнику - вы ведь его нарисовали, правда?:

```
SetColor(Green);  
Line(100,100, 300,100);  
Line(300,100, 100,200);  
Line(100,200, 100,100);
```

А теперь:

```
SetFillStyle(SolidFill, Green);  
FloodFill( 105,105, Green);
```

FloodFill закрашивает (заливает) всё, начиная с заданной точки (первые два параметра – (105,105)) пока не встретит границу заданного цвета (третий параметр - Green). Каким цветом красить, задаёт SetFillStyle. У нас цвет границы и цвет заливки совпадают – всё зелёненькое – но это необязательно. А что будет, если промахнёмся и не попадём в треугольник? А он, FloodFill, всё равно будет красить, пока не встретит зелёную границу, только не внутри границы, а снаружи. Непонятно? Попробуйте.

### Полезная вещь – метод опорной точки

Вернёмся к нашему старому квадрату:

```
Line(100,100, 200,100);  
Line(200,100, 200,200);  
Line(200,200, 100,200);  
Line(100,200, 100,100);
```

Это квадрат с левым верхним углом в точке (100,100). А теперь пришёл злой заказчик и передумал – левый верхний угол теперь в точке (120,130). Трудолобиво переписываем:

```
Line(120,130, 220,130);  
Line(220,130, 220,230);  
Line(220,230, 120,230);  
Line(120,230, 120,130);
```

Понравилось? Дурацкий вопрос. Пересчитали без ошибок? Ну, повезло, значит... А по умному нельзя? На случай, если заказчик опять придёт. Можно. Объявляем две переменные, которые будут олицетворять левый верхний угол:

```
var  
    x0,y0          : integer;
```

Дальше так:

```
x0:=100;  
y0:=100;  
Line(x0,y0, x0+100,y0);  
Line(x0+100,y0, x0+100,y0+100);  
Line(x0+100,y0+100, x0,y0+100);  
Line(x0,y0+100, x0,y0);
```

Теперь при возникновении злобного заказчика достаточно поменять две первые строки на следующий текст:

```
x0:=120;  
y0:=130;
```

и получаем желаемое заказчиком. Всем медитировать три минуты! Если кого-то посетила мысль, не завести ли в программе ещё одну переменную, например А, и не присвоить ли ей значение стороны квадрата, например 100, поздравляю. Мысль очень правильная. Развивать здесь пока не будем, и оставим в качестве самостоятельного упражнения. После освоения понятия процедуры методика эта станет ещё привлекательнее и заблещет новыми красками.

А называется это счастье *Метод опорной точки*.

## Глава 5, сложная Циклы и массивы

### Просто массив

Три вещи, которые надо постичь программисту – переменные, массивы и указатели. То есть, конечно, надо постичь ещё очень много других понятий, но эти, выделяясь, лежат на одной линии. Что сложного в переменной? Что у переменной есть имя и есть значение. С массивами и указателями та же проблема, только на другом уровне. Сейчас займёмся массивами, без этого никак. Без указателей, по правде говоря, тоже никак, но многие программисты как-то обходятся.

Сначала массив объявляем:

```
var
  a      : array[1..5] of integer;
```

Объявлен одномерный массив из пяти элементов целого типа. Элементы массива имеют индексы от одного до пяти. Кажется, я уже говорил, что присутствие в программе числа, отличного от нуля или единицы, является вредительством, а допустивший это программист подвергается экзекуции. Остаюсь при этом же мнении, но с пятеркой разберёмся попозже, а пока пусть будет.

А теперь что-нибудь с массивом сделаем. Для наглядности приведём всю программу полностью.

```
program arr;
  uses
    Crt;
  var
    a      : array[1..5] of integer;
begin
  ClrScr;
  a[1]:=2;
  a[2]:=1;
  a[4]:=a[1]+a[2];
  writeln('a[1]=',a[1]);
  writeln('a[2]=',a[2]);
  writeln('a[4]=',a[4]);
  readln;
end.
```

В нашем массиве под одним именем А скрываются пять целых переменных – А первое, А второе и так далее. Какая от этого польза – будет понятно чуть позже, а пока разберёмся в том, что мы написали. Вначале первому элементу массива присваивается значение два. Номер элемента массива называется индексом массива. Индекс массива указывается в квадратных скобках после имени массива. Имя массива в большинстве случаев сопровождается индексом. В остальном оператор присваивания выглядит совершенно как обычно.

Далее второму элементу массива точно так же присваивается значение один. В чем здесь трудность – у некоторых изучающих программирование? В том, что у массива в целом есть имя, а у элемента массива есть индекс и значение. Пустяк, а некоторым даётся трудно.

Четвёртому элементу массива присваиваем сумму первого и второго – только чтобы убедиться, что к элементам массива можно обращаться и в правой части оператора присваивания. Дальше мы выводим на экран значения первого, второго и четвертого элементов массива – тех, которым что-то присвоили. Что будет, если вывести значение третьего элемента? То же самое, что будет, если попытаться вывести значение переменной, которой ничего не присвоили – скорее всего, ноль, или, возможно, какой-то мусор.

И ещё. Пусть у нас N объявлено как целая переменная. Тогда можно написать вот так:

```
n:=3;  
a[n]:=10;
```

Как ни странно, это очень, очень важно. Почему возможность использования переменной в качестве индекса массива так важна, станет ясно в дальнейшем. А какая вообще польза от массивов? Да пожалуй, что и никакой, если использовать их без циклов.

Но всё же – массивы совершенно необходимы для хранения и обработки большого количества однотипных данных. Важны оба условия: большое количество и однотипность. Массив из двух элементов вызывает сильное подозрение, что что-то не так с головой программиста. Однотипность означает, что если в первом элементе массива хранится температура, то в следующих девяноста девяти элементах будет она же. Если в первом

элементе температура, а во втором давление, то программисту надо вежливо выкрутить руки и нежно спрашивать, слышал ли он о существовании записей. О записях позже.

## Просто цикл

С бесполезными массивами без циклов мы уже ознакомились, а теперь – циклы без массивов, предмет более осмысленный. Вот такая программа:

```
program cycle;
  uses
    Crt;
  var
    i      : integer;
begin
  ClrScr;
  for i:=1 to 5 do
    writeln('Au!!!');
  readln;
end.
```

Запускаем программу на выполнение и видим

```
Au!
Au!
Au!
Au!
Au!
```

А теперь разбираемся, что же мы напрограммировали и что получили в результате. Цифра пять в программе и пять выведенных на экран строчек наводят на мысль, что между ними есть какая-то связь. Действительно, если мы заменим 5 на 10, то выведенных на экран строк тоже станет десять. А теперь назовём всё своими именами. *i* – переменная цикла. Она должна быть объявлена как целая.

**for i:=1 to 5 do** – оператор цикла. То, что следует за словом **do**, обычно называют телом цикла. Тело цикла выполняется по одному разу для каждого значения переменной цикла от первого (1) до последнего (5) – то есть, в нашем случае, пять раз. После каждого выполнения значение переменной цикла возрастает на единицу. Одно выполнение цикла

называется *итерация*. Про переменную цикла говорят, что она *управляет* циклом.

Сначала не совсем понятно, зачем вообще нужна переменная цикла и начальное и конечное значения – если мы в нашей программе напишем вот так - `for i:=11 to 15 do` – то получим абсолютно тот же результат, цикл выполнится те же пять раз. Причина в том, что в нашей программе переменная цикла в теле цикла не используется. Это редкий случай, обычно она там присутствует.

По причинам исторического характера, переменным цикла принято давать такие имена: I, J, K, L, M, N. Это если имя короткое, в особенности из одной буквы. Также вполне допустимы имена длинные и относительно осмысленные – NumberOfLine, к примеру. Теперь напишем такой цикл:

```
for i:=1 to 5 do  
  writeln(i);
```

На экране после выполнения программы получим:

```
1  
2  
3  
4  
5
```

А если мы снова поменяем начальное и конечное значение цикла:

```
for i:=11 to 15 do  
  writeln(i);
```

то результат будет соответственно

```
11  
12  
13  
14  
15.
```

В целом понятно, не правда ли? И на всякий случай - цикл может выполняться ровно один раз. Вот так:

```
for i:=11 to 11 do
  writeln(i);
```

А может вообще ни разу. Вот так:

```
for i:=15 to 11 do
  writeln(i);
```

Ещё немного теории и немного практики. Тело цикла у нас состоит ровно из одного оператора. А если нужно больше, например, покрасить текст перед выводом в зелёный цвет? Всё то же самое, что и для условного оператора:

```
for i:=1 to 5 do begin
  TextColor(Green);
  writeln('Au!!!');
end;
```

Разумеется, точно так же между **begin** и **end** может находиться не только один оператор.

При каждом выполнении цикла значение переменной цикла возрастает и ровно на единицу. Так и только так. Это не потому, что изобретатели Турбо Паскаля по-другому не смогли, это из-за того, что они по-другому не захотели. По своей вредности они считали, что для целей образования переменная цикла должна только возрасть и только на единицу. А циклы, в которых это не так – неправильные циклы. Всё же, одно послабление они сделали. Если очень надо, чтобы переменная цикла убывала на единицу, то можно, но надо специально попросить. Цикл в этом случае пишется слегка по-другому: **for i:=5 downto 1 do**.

Теперь обещанная практика. Задачи, которые будут преследовать вас всю Вашу программистскую жизнь.

Сосчитать сумму целых чисел от 1 до 100. Я понимаю, что это арифметическая прогрессия. Я понимаю, что формула есть. Мы будем считать без формулы, тупо и в лоб. Это жизнь. Потом благодарить будете.

Что нужно, чтобы сосчитать сумму? Для начала нужна переменная, в которой эта сумма будет лежать. А поскольку считаем мы сумму именно целых чисел, то и переменная для суммы будет целой.

Если в корзину ничего ещё не положили, что в ней лежит? Правильно, ничего. Следовательно, нашей переменной надо присвоить нулевое значение. А откуда возьмутся целые числа от 1 до 100? Из переменной цикла, естественно, которая и будет меняться от 1 до 100. Итого имеем

```
sum:=0;
for i:=1 to 100 do
  sum:=sum + i;
writeln(' sum= ', sum);
```

А теперь чуть сложнее – сумма только чётных чисел в этом же диапазоне. Раз нам нужны только чётные числа – перед суммированием надо проверить, что число чётное. А раз нужно проверить – значит, будет присутствовать условный оператор. А условие на чётность, насколько помним, реализуется с помощью оператора **mod**. В результате:

```
sum:=0;
for i:=1 to 100 do begin
  if (i mod 2) = 0
    then sum:=sum + i;
end;
writeln(' sum= ', sum);
```

Пару **begin-end** в нашем конкретном случае можно не писать, но с ней как-то нагляднее.

Теперь вместо суммы – произведение. Надо заметить, что сумму приходится считать всегда, а произведение редко. Переменная для произведения по-прежнему целая, а вот с начальным значением по-другому. Немного поразмыслив, приходим к выводу, что ноль не подойдёт, а подойдет как раз единица. Считаем  $N!$ . Если кто помнит, так обозначается факториал от числа  $N$ , то есть произведение всех чисел от единицы до  $N$  включительно.

```
N:=5;
fact:=1;
for i:=1 to N do
  fact:=fact * i;
writeln('fact = ', fact);
```

А теперь – сумма дробных чисел. Посчитать сумму первых десяти элементов ряда  $1/N$ . Кто про ряды ещё - или уже - ничего не знает – пропускайте, когда понадобится, разберётесь сами. Всё почти то же самое, только сумму объявляем как single.

```
sum:=0;
for i:=1 to 10 do
  sum:=sum +1/ i;
writeln(' sum= ', sum:8:3);
```

Обратите внимание, что при выводе дробного числа мы указали, с какой разрядностью его выводить. Восемь – это число знаков всего, а не до запятой! Так, напоминаю на всякий случай... И, совсем на всякий случай и для расширения кругозора, то что мы запрограммировали называется *гармонический ряд*.

А теперь – очень сложная задача. Про деньги. Вначале имеем деньги в некотором количестве. Количество по запросу должен ввести пользователь программы. Деньги хранятся в банке некоторое количество лет, под некоторое количество ежегодных процентов. В порядке вредности – проценты сложные. Сложные проценты – это когда у вас в начале 100 рублей под 5 %. Через год будет 105 рублей – 5 рублей набежавшие проценты. Через год будет не  $105+5=110$  рублей, а  $105+105*0.05=110$  рублей 25 копеек. В порядке гуманизма – количество лет строго целое. И годы, и проценты, естественно, тоже вводятся пользователем. Вопрос очевиден – а сколько будет денежек в конце концов? Ваша задача – программно реализовать эту увлекательную задачу.

Ещё раз с другими константами - для альтернативно одарённых. Если в банк положили 200 руб. под 10 процентов, то через год будет 200 рублей плюс проценты, а проценты это  $200*(10/100)=20$ . Итого в кармане  $200 + 20 = 220$ . А через два года? Ответ  $200 + 20*2$  – неверный. Проценты у нас сложные, то есть во втором году начисляются не с исходной суммы 200 руб., а с того, что образовалось в конце первого года, то есть с 220 рублей. Результат будет  $200 + 200*(10/100) + (200 + 200*(10/100))*(10/100) = 200 + 20 + 22 = 242$  рубля. Вот такой ужасный ужас. А теперь – программируем!

Начинать надо всегда с простого и очевидного. Трудное на потом. К тому же всегда есть шанс, что пока делаешь, всё отменят и до трудного просто не дойдёт. Простое и очевидное у нас – что есть начальная сумма, количество лет и годовой процент. Значит надо объявить три переменные – одна целая - годы, две дробных – и предложить пользователю ввести их. Ещё объявить дробную переменную для результата, а после всех расчётов вывести её значение. Ну и очевидно, что в программе будет цикл – значит можно, не раздумывая, объявить переменную цикла. Так что, практически не думая, получаем такую полупрограмму:

```
program percent;
uses
  Crt;
var
  sum      : single; { начальная сумма}
  proc     : single;
  years    : integer;
  sumRez   : single; { конечная сумма}
  i        : integer;
begin
  write('sum = ');
  readln(sum);
  write('proc = ');
  readln(proc);
  write('years = ');
  readln(years);

  { а здесь будет всё самое главное - потом }

  writeln('money = ', sumRez:8:2);
  readln;
end.
```

И ещё о простом – понятно, что цикл у нас будет от единицы до количества лет, а перед циклом переменной sumRez надо что-то присвоить.

А теперь о главном. Чем проинициализировать переменную sumRez – нулём или начальной суммой? Если начальной суммой, то на каждом шаге мы будем прибавлять к ней образовавшиеся за год проценты. Если нулём, то, как-то даже и непонятно, что мы потом будем суммировать. Предлагаю инициализировать начальным вкладом. Прибавлять будем проценты за год, рассчитанные не от исходной суммы, а от той суммы, что у нас уже накопилась. Чтобы не удерживать в уме слишком длинных формул, делим математику пополам – сначала посчитаем проценты за

год, а потом прибавим их к итоговой сумме. А раз так, надо объявить для процентов за год специальную переменную - дробную. Назовем её prYear. Теперь вперёд:

```
sumRez:=sum;
for i:=1 to years do begin
  prYear:=sumRez*(proc/100);
  sumRez:=sumRez + prYear;
end;
```

Подставляем этот фрагмент на место комментария в выше приведённую программу – ну и собственно всё. Вот что получилось:

```
uses
  Crt;
var
  sum      : single; { начальная сумма}
  proc     : single;
  years    : integer;
  sumRez   : single; { конечная сумма}
  prYear   : single;
  i        : integer;
begin
  ClrScr;

  write('sum = ');
  readln(sum);
  write('proc = ');
  readln(proc);
  write('years = ');
  readln(years);

  sumRez:=sum;
  for i:=1 to years do begin
    prYear:=sumRez*(proc/100);
    sumRez:=sumRez + prYear;
  end;

  writeln(' money = ', sumRez:8:2);
  readln;
end.
```

Протестируйте.

## Просто циклы и графика

Совсем недавно мы написали такую программку:

```
for i:=1 to 5 do begin
  TextColor(Green);
```

```
writeln('Au!!!');  
end;
```

В результате получили, напоминая, пять одинаковых зеленых строчек на экране. Возникает естественная мысль изготовить графический аналог:

```
for i:=1 to 5 do begin  
  SetColor(Green);  
  Line(100,200, 300,200);  
end;
```

Но вместо пяти зеленых линий получили только одну. Почему, собственно? В чем разница между двумя текстами? Оператор `writeln` не только выводит текст на экран, он ещё и переходит на новую строку, так что следующий оператор `writeln` будет выводить свой текст уже там. У нас нет возможности перейти на другую строку в графическом режиме, просто потому, что строки вообще нет, но мы можем – и должны – явным образом задать, где начинается и где кончается наша линия. Конкретизируем, чего мы собственно хотим? – Нарисовать пять горизонтальных линий на расстоянии десять экранных точек – пикселей – друг от друга. Для горизонтальной линии у начальной и конечной точки координата  $Y$  одна и та же, а координата  $X$  разная. В процессе изготовления горизонтальных линий, координаты  $X$  будут оставаться неизменными, а координаты  $Y$  принимать другое значение, но одно и то же и у начальной и конечной точки. То есть, координаты  $Y$  должны последовательно принимать значения 200, 210, 220, 230, 240.

Чтобы координата  $Y$  менялась, она должна как-то зависеть от переменной цикла. Но нам нужен шаг десять, а переменная цикла всегда меняется только на единицу – значит надо переменную цикла умножить на десять! Получаем вот такое:  $200 + i*10$ . Быстренько перебираем в уме, какие значения принимает выражение, когда  $i$  меняется от единицы до пяти: 210, 220, 230, 240, 250. Стабильно получаем на десять больше. Значит, надо так же стабильно от этого десятка избавляться. Очевидный примитивный вариант – просто написать  $200 + i*10 - 10$ . Работает, но выглядит как-то неэстетично. Вспоминаем алгебру, выносим десять за скобки, в итоге имеем:

```
for i:=1 to 5 do begin  
  SetColor(Green);  
  Line(100,200+10*(i-1), 300,200+10*(i-1));  
end;
```

Работает. Выражение  $10*(i-1)$  примите к сведению и обдумайте. SetColor тоже неплохо вынести за скобки, за операторные скобки:

```
SetColor(Green);
for i:=1 to 5 do begin
    Line(100,200+10*(i-1), 300,200+10*(i-1));
end;
```

Только не подумайте, что это для ускорения программы. Исключительно из принципа и эстетических соображений – зачем повторять пять раз то, что достаточно сделать один? Вертикальные линии нарисуйте сами и получите симпатичную решёточку.

А теперь кружочки! То есть, конечно, окружности. Пойдём тем же путем. Рисовали пять параллельных линий – нарисуем десять окружностей. Чуть не сказал – параллельных окружностей. Хотя, в каком-то, вышем, смысле, да, параллельных. Десять окружностей с центром в одной точке и с радиусом, возрастающим на те же десять точек. Такие окружности называются ещё концентрическими.

Окружности у нас рисуются процедурой Circle, у которой три параметра. Два первых – центр окружности, третий – радиус. Центр у всех окружностей общий, так что два первых параметра не меняются, а третий возрастает, почти как при рисовании линий.

```
SetColor(Green);
for i:=1 to 10 do begin
    Circle(200,200, 100+10*(i-1));
end;
```

Готово. Пользуюсь возможностью ещё раз напомнить, что единожды заданный оператором (формально процедурой) SetColor цвет, действует до того, как будет явно изменён, тем же оператором SetColor. Звучит тривиально, на самом деле многих удивляет.

Вам зелёный цвет нравится? Мне нравится, и чтобы поярче. И ещё красный нравится и синий, как положено правильному программисту. Правильные программисты любят чистые цвета. Сейчас во всё это сразу и покрасим наши окружности. Красная, синяя, зелёная, и снова – красная, синяя, зелёная. Формализуем задачу – первая красная, вторая зеленая,

третья синяя, четвертая красная, пятая зелёная... Формализуем дальше - если остаток от деления номера окружности на три равен единице, то красная, если двум, то зелёная. Вспомнив про существование оператора **mod**, как будто специально для этого предназначенного, получаем:

```
for i:=1 to 10 do begin
  if (i mod 3) = 1 then SetColor(Red);
  if (i mod 3) = 2 then SetColor(Green);
  if (i mod 3) = 0 then SetColor(Blue);
  Circle(200,200, 100+10*(i-1));
end;
```

И ещё обратите внимание - привет первоклассникам! – остаток от деления на три, трём равняться не может, иначе число бы делилось на три без остатка. Так что в третьем условном операторе закономерно стоит ноль.

А теперь самостоятельно рисуем десять разноцветных концентрических кругов, это которые с помощью **SetFillStyle** и **FillEllipse**. Там, правда, есть один нюанс, но вы с ним, конечно, без проблем справитесь.

### Ещё одна несложная программа

Вспомним уже написанную программу по проверке способностей к сложению в уме (*Задачник Культина №88*). Не пожалеем места и напомним, что мы там напрограммировали.

```
program Test;
uses
  Crt;
var
  a,b,sum          : integer;
begin
  ClrScr;
  Randomize;
  a:=Random(100);
  b:=Random(100);
  write('How many is ',a,'+',b,' ');
  readln(sum);
  if sum=a+b
  then writeln('Ok')
  else writeln('Very very bad');

  readln;
end.
```

Программа, конечно хорошая, но можно и лучше. Задать не один пример для решения, а десять и, в зависимости от количества правильных ответов, осуществить раздачу слонов в виде оценок. Всё очень просто. Если нужно десять повторений – значит, нужен цикл от одного до десяти вокруг задания вопроса, получения ответа и проверки его правильности. Если нужно считать количество правильных ответов – значит нужно:

- А. объявить переменную для количества - целую
- Б. обнулить её перед циклом
- В. при каждом правильном ответе увеличивать значение переменной на единицу.

А после цикла, в зависимости от значения переменной, охарактеризовать испытуемого. Вроде бы всё. Получаем:

```
program Test;
uses
  Crt;
const
  N = 10;
var
  a,b,sum      : integer;
  okCount      : integer;
  i            : integer;
begin
  ClrScr;
  Randomize;
  okCount:=0;
  for i:=1 to N do begin
    a:=Random(100);
    b:=Random(100);
    write('How many is ',a,'+',b,' ');
    readln(sum);
    if sum=a+b then begin
      writeln('Ok');
      okCount:=okCount + 1;
    end
    else writeln('Very very bad');
  end;

  if (okCount>=0) and (okCount<=5) then writeln('Bad');
  if (okCount>=6) and (okCount<=8) then
writeln('Good');
  if (okCount>=9) and (okCount<=10)
    then writeln('Excellent');
  readln;
end.
```

Усовершенствуйте программу. По мелочи – пусть сообщение об оценке выдается разным цветом в зависимости от оценки.

И более существенное пожелание. Пусть N будет не константой, а переменной. То есть в начале выдётся запрос, сколько тестов хочет пройти испытуемый, а оценки расставляются так: 50% правильных ответов и ниже – плохо, от 50% до 80% - хорошо, выше – отлично.

### А теперь всё вместе

Скрещиваем массивы и циклы. Идея напрашивается сама собой – в массиве много-много элементов, цикл выполняется много-много раз. У массива есть индекс элемента, у цикла есть переменная цикла. Поскольку и то и другое – целое, ничто не мешает использовать переменную цикла в качестве индекса массива. Чем сейчас и займемся. Введём в цикле значения массива, и, в цикле же, выведем их. Вот так:

```
program arr_2;
uses
  Crt;
var
  a      : array[1..5] of integer;
  i      : integer;
begin
  ClrScr;
  { тут вводим}
  writeln('Input');
  writeln;
  for i:=1 to 5 do begin
    write('a[',I,'] = ');
    readln(a[i]);
  end;
  writeln;

  { тут делаем что-нибудь полезное}

  { тут выводим}
  writeln('Output');
  writeln;
  for i:=1 to 5 do
    writeln(a[i]);
  readln;
end.
```

После запуска получим вот такую картинку:



```
c:\bpascal\bin\r.bat - Far
Input
a[1] = 11
a[2] = 22
a[3] = 33
a[4] = 44
a[5] = 55
Output
11
22
33
44
55
```

Мешанина кавычек и запятых в строке `write('a[',I,'] = ');` призвана обеспечить более приятное и удобное приглашение для ввода элементов массива. Если что-то непонятно, придётся заучить наизусть. Все наши ближайшие программы подразумеваются имеющими в точности такой же объявленный массив и в точности такой же ввод и вывод.

Возвращаемся к проблеме трижды встречающейся в тексте программы пятерки. Это, безусловно, нехорошо, по причинам объясненным ранее. Будем проблему решать, раз и навсегда. Поговорим о константах.

Что такое константа? Почти то же самое, что и переменная, только в отличие от переменной получает своё значение раз и навсегда. Ну и по мелочи:

- переменная объявляется в секции **var**, константа в секции **const**;
- после имени переменной следует двоеточие, после имени константы знак равенства;
- после разделителя у переменной тип, у константы значение;
- в одной строке можно объявить много переменных, а константы объявляются строго по одной.

Вот так выглядит объявление необходимой нам константы:

```
const
  N = 5;
```

Секция констант по традиции предшествует секции переменных. Во-первых, по традиции и для единообразия. Во-вторых, как в нашем случае, иначе нельзя по причине использования констант в объявлении переменных. У нас константа N служит верхней границей в объявлении массива. Теперь каждый идентификатор N в нашей программе будет эквивалентен числу пять, и, если размер массива изменится, достаточно будет внести исправление только в одно место нашей программы. А программа наша тем временем приобрела такой вид:

```
program arr_2;
uses
  Crt;
const
  N = 5;
var
  a      : array[1..N] of integer;
  i      : integer;
begin
  ClrScr;
  { тут вВОДИМ}
  writeln('Input');
  writeln;
  for i:=1 to N do begin
    write('a[',I,'] = ');
    readln(a[i]);
  end;
  writeln;

  { тут делаем что-нибудь полезное}

  { тут ВЫВОДИМ}
  writeln('Output');
  writeln;
  for i:=1 to N do
    writeln(a[i]);
  readln;
end.
```

А можно вообще без констант? Можно. Почти. Вообще константы можно было бы заменить переменными, присвоить им один раз значение, а с программиста взять честное слово, что он не будет эти значения менять. К сожалению, на честное слово программиста надежда – я проверял – небольшая, лучше ограничить программиста компилятором. В Delphi появились константы, значение которых можно менять. Лично мне кажется, что разработчики языка в данном случае были неправы.

А почему нельзя совсем без констант? А потому что в Паскале начальный и конечный диапазоны в объявлении массива не могут быть переменными. И ещё на что нужно обратить внимание – одно имя может быть дано в программе только один раз. Имя можно дать переменной. Имя можно дать константе. Можно никому не давать. Но нельзя одним и тем же именем назвать и константу и переменную.

## Опыты

Стандартные задачи при работе с массивами. Встречаться они будут не часто, а очень часто. Ничего изобретать не надо, думать тоже ничего не надо – программист должен это писать, не думая абсолютно и на полном автопилоте.

У нас всё тот же массив от одного до N. Все появившееся неупомянутые ранее переменные – целые.

1. Заполнить массив одним и тем же значением, чаще всего нулём:

```
for i:=1 to N do  
  a[i]:=0;
```

2. Заполнить массив случайными значениями в диапазоне от 0 до 99:

```
for i:=1 to N do  
  a[i]:=Random(100);
```

3. Поместить в каждый элемент массива его индекс:

```
for i:=1 to N do  
  a[i]:=i;
```

4. Увеличить каждый элемент массива на единицу:

```
for i:=1 to N do  
  a[i]:=a[i] + 1;
```

5. Определить сумму элементов массива:

```
sum:=0;  
for i:=1 to N do  
  sum:=sum + a[i];
```

6. Определить среднее элементов массива (результат - дробный)

```
sum:=0;  
for i:=1 to N do  
  sum:=sum + a[i];  
result:=sum/N;
```

7. Найти минимальный элемент массива и его индекс:

```
min:=a[1];  
index:=1;  
for i:=2 to N do begin  
  if a[i] < min then begin  
    min:=a[i];
```

```

        index:=I;
    end;
end;

```

8. Определить количество отрицательных и неотрицательных элементов массива:

```

numNeg:=0;
numNonNeg:=0;
for i:= 1 to N do begin
    if a[i] < 0
    then numNeg:=numNeg + 1
    else numNonNeg:=numNonNeg + 1;
end;

```

9. Вставить нулевой элемент в пятую позицию - верим, что количество элементов массива не меньше пяти. При этом все элементы, начиная с бывшего пятого должны сдвинуться на одну позицию вправо:

```

for i:=N downto 6 do
    a[i]:=a[i-1];
a[5]:=0;

```

10. Сдвинуть все элементы, начиная с четвертого налево на три. Хвост заполнить нулями.

```

for i:=1 to N-3 do
    a[i]:=a[i+3];
for i:=N-2 to N do
    a[i]:=0;

```

Вариант альтернативный, но допустимый:

```

for i:=1 to N do
    if i <= N-3
    then a[i]:=a[i+3]
    else a[i]:=0;

```

11. Определить индекс первого чётного элемента массива:

```

indexEven:=0;
for i:=1 to N do begin
    if ((a[i] mod 2) = 0) and (indexEven=0)
    then indexEven:=i;
end;

```

Вторая проверка добавлена, чтобы вернуть номер только и только первого чётного элемента. Если этой проверки не будет, всё может случиться совсем по-другому. В случае нечётности всех элементов массива мы должны вернуть в качестве ответа ноль. Это общепринятая практика – если ответа нет, возвращать некоторое невозможное значение. Если ноль является допустимым ответом, обычно возвращают минус единицу. А если и единица оказывается допустимым ответом? Нет в мире идеала, нет...

Скобки вокруг `a[i] mod 2` поставлены исключительно для наглядности и выразительности – по синтаксису они не нужны.

По хорошему, вторая проверка вообще не нужна – вам поможет волшебное слово `Break` – и на гигантском массиве программа заработает чуть-чуть быстрее (тот, кого это волнует – не очень хороший программист). Поскольку оно – `Break` - противоречит принципам структурного программирования, о слове `Break`, возможно, вспомним потом. О структурном программировании, тоже, возможно, потом – или в совсем другой книге. Впрочем, все приведённые программы от заветов структурного программирования пока не отступили не на шаг.

### Ещё опыты

Всё то, что мы делали в предыдущем разделе, мы делали с одним массивом. Несколько типовых задач, в которых участвуют два массива. Первый массив зовут `A`, второй массив зовут `B`. Оба массива содержат одинаковое число элементов – `N`.

1. Проще не бывает. Присвоить все элементы одного массива другому.

Вопрос на самом деле не совсем простой. Потому что можно и так: `b:=a;` Но только при условии, что массивы объявлены так `a,b : array[1..N] of integer`. А если вот так `a : array[1..N] of integer; b : array[1..N] of integer;` то уже ничего не выйдет, хотя объявления абсолютно идентичны. Дальше нам надо плавно перейти к обсуждению определённых пользователем типов, но это позже. Вот этот способ работает всегда:

```
for i:=1 to N do
  b[i]:=a[i];
```

2. Посчитать поэлементно сумму массивов и отправить в первый массив:

```
for i:=1 to N do
  a[i]:=a[i] + b[i];
```

3. Положительные элементы из первого массива отправить во второй, но чтобы во втором они шли без пропусков в начале массива. То есть, было в первом массиве `[1,2,3,-5,1,-7,2]`. Во втором должно образоваться `[1,2,3,1,2]`.

```

ind:=0;
for i:=1 to N do begin
  if a[i]>0 then begin
    ind:=ind + 1;
    b[ind]:=a[i];
  end;
end;
end;

```

4. В первом массиве для нас представляют интерес только первые N1 элементов, во втором массиве, соответственно, N2. Объединить N1 и N2 первых элементов их двух массивов, записав их в первый массив. Верим и надеемся, что  $N1+N2 \leq N$ .

```

for i:=1 to N2 do
  a[N1+i]:=b[i];
N1:=N1 + N2;

```

5. Переписать элементы первого массива во второй, но в обратном порядке

```

for i:=1 to N do
  b[N-i+1]:=a[i]

```

А теперь перепишите элементы массива в обратном порядке, не используя второго массива. То есть, внутри него самого.

Как поменять два элемента местами? Обычно делают так (wrk – целая переменная, исключительно для вспомогательных целей):

```

wrk:=a[i];
a[i]:=a[i+1]
a[i+1]:=wrk;

```

Подумайте, как сделать это, не используя вспомогательной переменной wrk, и безо всяких хакерских трюков, само собой. Это несложно, для целых чисел точно несложно.

### Самый главный опыт

Из всех упражнений, которые приходится проделывать над массивами, наиважнейшим для нас является, бесспорно, сортировка. Что делает сортировка? Сортировка, ну, извините, в общем, сортирует...

На входе:

[10, 22, 11, 50, 30]

На выходе

[10, 11, 22, 30, 50]

Методов сортировки огромное количество. Главный критерий оценки этих методов – быстродействие. Математическую сторону, как и везде в этой книге, мы беспощадно упрощаем, уплощаем и опошляем. Одни методы сортируют быстрее, другие ещё быстрее, а совсем другие – совсем наоборот...

Большинство программистов используют самый худший из худших методов – пузырьковую сортировку. И я тоже. Такова традиция. Программист, который не может её запрограммировать за пять минут – неполноценен во всех отношениях, даже если освоил все остальные методы. Мы в неполноценные не хотим, поэтому сейчас будем программировать пузырьковую сортировку. Почему она пузырьковая? Запрограммируйте, включите воображение и поймёте.

В чём идея? Сравниваем первый и второй элементы. Если первый меньше второго (или равен второму), то всё хорошо и ничего делать не надо. Если первый больше второго, то это плохо, поэтому меняем первый элемент со вторым местами. Потом сравниваем второй элемент с третьим, реагируем аналогично. И так доходим до сравнения предпоследнего элемента с последним.

А затем начинаем всё сначала. Если за время всего прохода по массиву мы никого не поменяли местами, то всё хорошо, все элементы на своих местах. и можно заканчивать. Получившие математическое образование могут потребовать от меня строгого доказательства и получить от меня в ответ предложение доказать эту пустяковину самому.

Что понятно сразу? Внешний, главный цикл прямо-таки напрашивается быть запрограммированным как **repeat-until** – повторять до тех пор, пока не станет всё хорошо. Внутренний цикл обычный, и повторяться он будет очевидно  $N-1$  раз.  $N$  – как всегда, количество элементов в массиве.

Объявим специальную переменную по имени Ok. Как только она – да, то мы – всё. Получаем такой эскиз программы:

```

const
  N = 5;
var
  a          : array[1..N] of integer;
  ok         : integer;
  i          : integer;
.....
repeat
  for i:=1 to N-1 do begin
    if a[i] > a[i+1] then begin
      {поменять местами}
    end;
  end;
until ok=1;

```

Мы просто записали на Паскале то, что раньше сказали словами. Что осталось неясным? Судьба переменной Ok. Договоримся, что если Ok=1, то всё хорошо, то есть о`кей, а если Ok=0, то, наоборот, всё плохо, не о`кей, в смысле. Мы уже решили, что основной цикл выполняется до тех пор, пока не Ok – и записали это. Очевидно, перед началом сортировки у нас не всё Ok – а иначе, зачем сортировать? Значит, перед циклом переменная устанавливается в ноль. Опять-таки, если пришлось поменять элементы местами, значит у нас опять всё плохо – и опять переменной Ok присваиваем ноль. Но когда-то же должна она стать единицей, цикл ведь должен закончиться? Стандартная технология для таких случаев – перед циклом предполагаем, что всё хорошо. Если внутри цикла хоть раз что-то оказалось не так – переменная получает значение ноль.

И, наконец, наша программа приобретает законченный вид:

```

const
  N = 5;
var
  a          : array[1..N] of integer;
  ok         : integer;
  wrk       : integer;
  i          : integer;
.....
ok:=0;
repeat
  ok:=1;
  for i:=1 to N-1 do begin
    if a[i] > a[i+1] then begin

```

```

        ok:=0;
        {поменять местами}
        wrk:=a[i];
        a[i]:=a[i+1];
        a[i+1]:=wrk;
    end;
end;
until ok=1;

```

Прикрутите в начало заполнение массива случайными числами, вывод до сортировки и вывод после неё и любуйтесь результатом.

### Как не делать ничего

У нас уже была программа, которая не делает ничего и строка, которая не содержит ни одного символа. Двинемся дальше по этому заманчивому пути – теперь у нас оператор, который не делает ничего. Как он называется? Пустой оператор. Как он выглядит? Да никак. Точнее есть две школы. Согласно первой, пустой оператор состоит из точки с запятой. Согласно второй школе, пустой оператор фигуры не имеет и места не занимает.

Иллюстрирую на примере. Это без пустого оператора:  
`writeln('qwerty');`

А это с пустым оператором:  
`writeln('qwerty');`

Вот то, что находится между двумя точками с запятой и есть пустой оператор. Что значит, там ничего нет? Ну да, ничего нет, так он ничего и не делает. Вообще-то, Паскаль несколько чувствительно относится к лишним точкам с запятой. Точнее, не то, чтобы чувствительно, а в строгом соответствии с синтаксисом языка. Точка с запятой между `begin` и `end`. Это, как и обещано, пустой оператор. А вот лишняя точка с запятой в декларативной части программы, между `program` и `begin`, будет жестока караться и преследоваться.

Для чего пустой оператор нужен? Для чего хотите. Я сам не хочу, совершенно. А где его можно использовать? Пустой оператор может использоваться во всех местах программы, где может использоваться любой другой оператор. Сначала, плохие примеры. Некоторые пишут вот так:

```
if x>0 then y:=100 else;
```

Это глупость, но безвредная. Убираем лишнее, остаётся

```
if x>0 then y:=100;
```

А некоторые пишут вот так:

```
if x>0 then else y:=200;
```

Поскольку перед `else` точка с запятой не ставится, пустой оператор уменьшился до своего естественного вида, то есть до никакого. Так вот, это тоже глупость, но очень вредная. Программиста надо долго бить по рукам, затем повернуть ему мозги набок и написать так:

```
if x<=0 then y:=200;
```

Применим полученные навыки ничегонеделания к оператору цикла:

```
for i:=1 to 1000 do;
```

А вот тут не всё так просто. Пустой оператор не делает ничего. Написанный нами оператор цикла не делает ничего тысячу раз. Но нельзя сказать, что не делает совсем ничего – результата, конечно, нет, но жужжать-то он при этом жужжит. Иными словами, отбирает ресурсы и загружает процессор. На каждое выполнение цикла уходит хотя и очень маленькое, но время, даже если внутри цикла абсолютно ничего не делается - накладные расходы. Пять старушек, как известно, рупь. А цикл, ничего не делающий тысячу раз и тратящий на это *ничего* очень маленькое время, потратит этого очень маленького времени в тысячу раз больше и, вполне может быть, это будет уже не очень маленькое время. Переведём безделье на качественно другой уровень – ничего не делающий вложенный цикл.

```
for i:=1 to 1000 do  
  for j:=1 to 1000 do;
```

Вложенных циклов у нас пока не было. По плану, они будут позже. Просьба над этим пока особо не задумываться. Если интуитивно понятно – замечательно. Если непонятно – неважно. Помните, как Старик

Хоттабыч сваял телефон из цельного куска мрамора? Вот как к такому цельносваянному телефону и относитесь.

Обратите внимание – ничего не делает тысячу раз только второй цикл. Первый цикл при деле – он тысячу раз выполняет второй – поэтому после первого **do** точка с запятой отсутствует.

А всего мы имеем – тысяча умножить на тысячу – миллион раз отработавший впустую внутренний цикл. А это уже кое-что!

И вечный вопрос – а зачем? Зачем вложенные циклы, которые ничего не делают, только время жрут?

Вернемся к примеру про пять параллельных линий.

```
SetColor(Green);  
for i:=1 to 5 do begin  
    Line(100,200+10*(i-1), 300,200+10*(i-1));  
end;
```

Программа выполняется последовательно, сначала рисуется первая линия, затем вторая. Но появляются они на экране одновременно. Нетрудно понять, почему – появляются они на самом желе поочерёдно, но настолько быстро, что это кажется одновременным. А если мне хочется, чтобы медленно и по одной? Нельзя ли как-нибудь притормозить компьютер?

В теории всё очень просто. Есть процедура `Delay`. У неё один параметр, указывающий задержку в миллисекундах – тысячных долях секунды. То есть, если написать `Delay(1)`, то программа на этом месте задержится на одну миллисекунду, ничего не делая, а если `Delay(1000)`, то программа будет ничего не делать ровно секунду. В теории. На практике, всё это было давно. Турбо Паскаль разрабатывался под медленные, по нынешним понятиям, процессоры, и с нынешними скоростями не дружит. Короче, в наше время `Delay` не работает.

И вот тут приходят на помощь наши, казалось бы, бессмысленные опыты. Сразу чешутся ручонки сделать что-нибудь этакое:

```
for i:=1 to 5 do begin  
    Line(100,200+10*(i-1), 300,200+10*(i-1));  
    {здесь тормозим}
```

```
    for i:=1 to 1000 do
      for j:=1 to 1000 do;
end;
```

Сделали? Нехорошо вышло? А почему? А почему у нас подозрительно одноимённые переменные цикла? Короче, исправили, но всё равно не тормозит. На этот раз причина банальна – миллиона маловато будет для 2ГГц процессора. Меняем 1000 на 30000.

```
for i:=1 to 5 do begin
  Line(100,200+10*(i-1), 300,200+10*(i-1));
  {здесь тормозим}
  for j:=1 to 30000 do
    for k:=1 to 30000 do;
end;
```

Если хочется ещё медленнее, то просто увеличить константу не получится. А почему? Вспомните, какое максимальное число может уместиться в `integer`. Надо или написать три аналогичных вложенных цикла, или заменить `integer` на `longint` и увеличивать константу в своё удовольствие.

`Longint` во всех отношениях то же самое, что `integer`, но в него помещаются целые числа до двух миллиардов. А почему мы везде не пишем `longint`, а пишем `integer`? По традиции, наверное.

А может, мы просто жмоты. `Integer` занимает два байта, а `longint` аж четыре. Впрочем, иногда и это бывает важно. Открою мрачную тайну Турбо Паскаля – общий размер всех объявленных в программе переменных не может превышать 64К. Впрочем, как и всегда, есть другие обходные пути.

### Что-нибудь полезное

И не только полезное, но и красивое! Полезное, чтобы оно включало циклы, массивы и вот это – про торможение программы. А красивое, чтоб графика была. Хочу, чтобы было небо! И чтобы звёзды на нём! И, главное, чтобы звёзды моргали!

Конкретизирую. Чёрное-чёрное небо – в смысле, чёрный-чёрный экран. На нём сначала нарисовано штук тридцать, думаю, этого хватит, звёздочек. Звёздочки – это `PutPixel`. Затем какая-нибудь, совершенно

случайная звездочка гаснет, а в каком-нибудь другом, совершенно случайном месте экрана звездочка загорается. Медленно и печально. И так пока не надоест.

*А ведьма, между тем, поднялась так высоко, что одним только черным пятнышком мелькала вверху. Но где ни показывалось пятнышко, там звезды, одна за другою, пропадали на небе. Скоро ведьма набрала их полный рукав. Три или четыре еще блестили.* © Гоголь, Ночь перед рождеством

Какие выводы напрашиваются из этого технического задания? Поскольку процесс длительный, должен быть цикл. Цикл *пока не надоест* мы не умеем, пока. Напишем обычный цикл, до тысячи, например. Поскольку звезды загораются и гаснут случайным образом, обязательно возникнут Random и Randomize. Черное небо – Bar, Звезды PutPixel. А раз должно быть медленно – должна быть задержка. Это всё понятно. Дальше смутно маячат некоторые сложности. На сложности не обращаем внимания и программируем то, что нам уже понятно – таков общий подход.

```
program stars;
uses
  Graph;
var
  driver, mode      : integer;
  i,j,k,n          : integer;
begin
  Randomize;
  driver:=VGA;
  mode:=VGAHi;
  {предполагаем, что egavga.bgi в текущем каталоге}
  InitGraph( driver, mode, '');

  {чёрное-чёрное небо}
  SetFillStyle( SolidFill, Black);
  Bar( 0,0, 639,479);

  {нарисовать тридцать звездочек}

  for i:=1 to 1000 do begin
    {погасить случайную звезду}
    {зажечь где-нибудь}
    {задержка, подобрать по вкусу}
    for j:=1 to 1000 do
      for k:=1 to 30000 do;
  end;
```

```
    readln;  
    CloseGraph;  
end.
```

Нарисовать тридцать звезд нетрудно. Цикл до тридцати, внутри цикла PutPixel – и всё. В чём проблема? Проблема дальше – там, где звезду надо погасить. Гасить мы её будем применяя по ней тот же PutPixel, только черным цветом. А вот по какому месту – на экране - его применять? Постепенно приходим к осознанию факта, что местонахождение всех звёзд нам придется ещё и запомнить. Звёзд тридцать штук, значит, массив будет из тридцати элементов. Другой вопрос – тридцати каких элементов? К сожалению, на данном этапе нашего развития, могу только предложить два массива по тридцать каждый – один для хранения координаты X, другой для хранения координаты Y. Коряво как-то? Корявенько, да. А по-людски нельзя? По-людски, конечно, можно, но требует введения новых сущностей и усвоения новых знаний. А голова, она не резиновая. У меня – точно. Так что, делаем неэстетично, зато работоспособно.

- *Сеня, это же неэстетично!*

- *Зато дешевле, надёжно и практично* © к/ф Бриллиантовая рука

Вспоминаем, что нам придётся случайным образом выбрать гасимую звезду, и объявляем переменную для её номера и, заодно, и для её координат. Итого, в секциях объявлений и констант у нас добавляются:

```
const  
    maxStar = 30;  
var  
    starX      : array[1..maxStar] of integer; {это координаты}  
    starY      : array[1..maxStar] of integer;  
    star       : integer; {номер гаснущей звезды}  
    x, y       : integer; {координаты новой звезды}
```

Кстати, имена переменным я дал не очень хорошие. Почему? Объясню позже. Кому невтерпёж – читайте *Элементы стиля программирования*. Теперь у нас есть, где запомнить координаты наших пикселей, олицетворяющих звезды. Рисуем и запоминаем:

```
for n:=1 to maxStar do begin  
    x:=Random(640);  
    y:=Random(480);
```

```

    PutPixel( x,y, White);
    starX[n]:=x;
    starY[n]:=y;
end;

```

А почему переменная цикла вдруг называется N? Потому что порядочные девушки используют переменную цикла только для одной цели. Переменная I у нас использована для главного цикла, J и K организуют задержку, приходится задействовать N. Ничего, в запасе ещё M и L. Хотя, конечно, если до них дойдёт дело, значит что-то не так. Переменным ведь можно давать и осмысленные имена, не правда ли?

Что важнее, обратите внимание на Random(640). Значение в каком диапазоне он вернёт? Правильно, от нуля до 639. Именно то, что нам и нужно, экранные пиксели нумеруются именно так. К сожалению, не всегда так удачно совпадает. Теперь выбираем звезду случайным образом – одну из тридцати. Хочется вот так – star:=Random(30). Согласно описанию функции Random получим случайное число в диапазоне от 0 до 29. Надо бы на единицу больше. Куда эту единицу добавить? Многие пишут так – star:=Random(30+1). Так не надо, так плохо. Хорошо вот так - star:=Random(30)+1. Запомните, не раз ещё пригодится. Только вспомните, что у нас не 30, а константа maxStar.

Теперь осталось

- А) погасить звезду
- Б) выбрать координаты новой звезды - случайным образом, разумеется
- В) зажечь звезду
- Г) запомнить координаты

Переводим с русского на Паскаль. Русский текст оставляем в качестве комментариев:

```

{погасить звезду}
PutPixel( starX[star], starY[star], Black);
{выбрать координаты новой звезды}
x:=Random(640);
y:=Random(480);
{зажечь звезду}
PutPixel( x,y, White);
{запомнить координаты}
{обратите внимание! Откуда стёрли, туда и рисуем!}
starX[star]:=x;

```

```
starY[star]:=y;
```

Ну, вот вроде бы и всё. Но если можно без особого труда произвести впечатление, почему бы и нет? Пусть звёзды будут разноцветные, случайного цвета опять-таки.

Теперь теория. Теория такова. У нас есть палитра. В палитре 16 цветов. Нумеруются цвета от нуля до пятнадцати. Под нулевым номером идёт чёрный, под первым синий, по вторым зелёный и т.д. Когда мы пишем SetColor(Green), это равносильно SetColor(2), то есть выбрать из палитры цвет под номером два. При некотором желании можно во вторую ячейку палитры вместо зелёного записать какой-нибудь экзотический цвет, но пока нам этого не надо. И вообще, за всю программистскую жизнь, мне этого ни разу не понадобилось. А надо нам случайно выбрать цвет из палитры, один из шестнадцати. color:=Random(16), вообще-то подошло бы, но чёрный цвет нам не нужен.

Так что объявляем переменную и пишем:

```
color:=Random(15) + 1;
```

Ну а теперь собираем всё вместе, стараясь ничего не потерять по дороге.

```
program stars;
uses
  Graph;
const
  maxStar = 30;
var
  starX      : array[1..maxStar] of integer; {это
координаты}
  starY      : array[1..maxStar] of integer;
  star       : integer; {номер гаснущей звезды}
  x,y        : integer; {координаты новой звезды}
  color      : integer; {цвет звезды}
  driver, mode : integer;
  i,j,k,n    : integer;
begin
  Randomize;
  driver:=VGA;
  mode:=VGAHi;
  {предполагаем, что egavga.bgi в текущем каталоге}
  InitGraph( driver, mode, '' );

  {чёрное-чёрное небо}
  SetFillStyle( SolidFill, Black);
  Bar( 0,0, 639,479);
```

```

{нарисовать тридцать звездочек}
for n:=1 to maxStar do begin
  x:=Random(640);
  y:=Random(480);
  PutPixel( x,y, White);
  starX[n]:=x;
  starY[n]:=y;
end;

for i:=1 to 1000 do begin
  star:=Random(maxStar) + 1;
  {погасить звезду}
  PutPixel( starX[star], starY[star], Black);
  {выбрать координаты новой звезды}
  x:=Random(640);
  y:=Random(480);
  {зажечь звезду}
  color:=Random(15) + 1;
  PutPixel( x,y, color);
  {запомнить координаты}
  {обратите внимание! Откуда стёрли, туда и рисуем!}
  starX[star]:=x;
  starY[star]:=y;
  {задержка, подобрать по вкусу}
  for j:=1 to 1000 do
    for k:=1 to 30000 do
  end;

  readln;
  CloseGraph;
end.

```

Совсем забыли, как же клиент всё это великолепие прекратит, когда надоест? Смотрите и запоминайте, пригодится. Не только здесь пригодится, тут у нас не самый тяжелый случай – цикл выполняется долго, но не вечно. А бывает, что и вечно – не нарочно, разумеется. Ещё не умеете писать вечных циклов? Научим!

Когда программа входит в бесконечный цикл, про неё говорят, что она заиклилась. Это очень, очень плохо!

Как с этим бороться? Нажимаем **Ctrl/Break** и получаем поверх обычного экрана редактирования окошко с единственной кнопкой **Ok**. Нажимаем **Enter**. Вываливаемся в нашу программу, только одна строка, где-то в районе цикла задержки, выделена зелёным цветом. На самом деле мы, того не желая, перешли в режим отладки. Это очень, чрезвычайно

полезный режим, но о нём поговорим позже. А пока нажимаем Ctrl/F2 и всё. Вернулись.

Метод очень хороший, но срабатывает, только когда наша программа запущена из-под среды Турбо Паскаля. Если запущен готовый исполняемый файл, это не поможет, придётся убивать программу другими способами.

Что делать? Вообще говоря, программировать аккуратнее, не допуская заикливания. Конкретно в нашем случае, при каждом выполнении цикла проверять, не нажата ли клавиша Esc или что-то вроде того, и, в случае нажатия, прекращать исполнение. Как это сделать, скоро узнаем.

## Глава 6 Строки

### Просто строка

Знаменательное событие. Было у нас два типа данных – целые и дробные, `integer` и `single`, будет три. По-русски – строка, по-программистски – `string`.

*У старинушки три сына:*

*Старший умный был детина,*

*Средний сын и так и сяк,*

*Младший вовсе был дурак.* © Ершов. Или Пушкин? Нет, всё-таки, Ершов...

Короче, тип не совсем похожий на предыдущие типы. По сути, замаскированный массив. На самом деле, массив он и есть, и даже не очень замаскированный.

Теперь поступим, как всегда – объявим переменную типа строка, присвоим что-нибудь переменной типа строка, выведем переменную типа строка. Поехали.

```
var  
  s      : string;
```

Переменной цикла принято давать имя `I`, массив часто называют `A`, у строки обычная кличка – `S`. Набрав слово `string` без ошибок, обнаружим, что оно выделено белым цветом, то есть тоже относится к разряду волшебных зарезервированных слов, в отличие от `integer` и `single`. Почему – тайна.

Присваиваем:

```
S := 'woodpecker';
```

Напоминаем для озабоченных – если хочется, чтобы внутри строки была кавычка, надо вместо одной поставить две, вот так: `'wood''pecker'`. Выведено на экран будет `wood'pecker`. Максимальная длина строки – 255 символов.

Для любознательных. Строку можно объявить вот так, например: `s : string[30]`; Максимальная длина такой строки не 255, а только 30 символов. Зачем это надо? Если очень хочется сэкономить оперативную память. Под просто `string` выделяется по умолчанию 256 байтов, а под `string[30]` будет выделен 31 байт. Почему не тридцать? Для пытливых умов - есть ещё нулевой байт, `s[0]`. В нем хранится фактическая длина строки. То есть после оператора `s:='abcd'`; значение нулевого байта будет равно четырем.

Для любителей совать пальцы в розетку. Во времена Паскаля некоторые, чтобы узнать длину строки, вместо `Length(s)` предпочитали писать `Ord(s[0])`. Ещё чаще, при необходимости оставить от строки, например, первые десять символов, писали `s[0]:=Chr(10)`; И всё, в общем-то, у некоторых работало. И даже под Delphi первой версии работало. И как же эти некоторые огребли при переходе на Delphi 2... А что такое `Ord` и `Chr`, разужайте сами. Потом, конечно, я вам расскажу.

Если у нас есть две целые переменные, то мы можем поставить между ними знак арифметической операции и результат присвоить третьей целой переменной. Для дробных чисел всё так же, только операции немного другие. А для строк?

```
var
  s1,s2,s3      : string;
begin
  ClrScr;
  s1:='123 `';
  s2:='abcd `';
  s3:=s1 + s2;
  writeln(s3);
```

Получим на экране

123 abcd

С плюсом разобрались – он склеивает две строки в одну. А ещё? А всё! Больше операторов не предусмотрено. Почти. Придётся идти другим путём.

## Просто строка и её процедуры

Не совсем её процедуры. Скорее процедуры для неё. И не совсем процедуры. Скорее функции. Но неважно, сейчас перейдём к конкретике, и всё будет понятно.

Функция `Length`. Не очень интересная, но самая важная. Поскольку это функция, вроде `Sin`, то она возвращает значение, которое можно присвоить переменной. Только у `Sin` значение дробное, а у `Length` целое. Пишем вот такое:

```
var
  s          : string;
  len       : integer;
begin
  ClrScr;
  s:='12345';
  len:=Length(S);
  writeln(len);
```

Видим на экране цифру пять. Путём несложных умственных усилий приходим к немудрёному выводу, что функция `Length` возвращает нам количество символов в строке. А почему она самая важная? А это станет ясно в следующем разделе. Кстати, помните, что кавычка внутри строки кодируется двумя идущими подряд кавычками? В подсчёте символов эти две кавычки, как и следовало ожидать, считаются за одну.

А пока – функция `Copy`.

```
s1:='12345';
s2:=Copy(S, 2, 3);
writeln(s2);
```

В результате имеем 234. `Copy` выкусывает из строки часть, начинающуюся с символа имеющего номер, указываемый вторым параметром и в количестве символов, которое задаёт третий параметр. Обратите внимание, что эта функция в качестве результата возвращает не число, а строку.

Теперь функция `Pos`. Она возвращает целое число и занимается поисками одной строки в другой. Если занудно, то поисками подстроки в строке. Вот так:

```
s1:='12345';  
where:=Pos('2', s1);
```

После выполнения where будет равно, разумеется, двум. А если искать в строке что-нибудь эдакое, чего в ней нет, то в результате получим, как обычно принято, ноль.

Ещё совершенно необходимая – не функция – процедура Delete.

```
s1:='12345'  
Delete( s1, 3,2);
```

В результате в s1 останется только 125. Двойка в параметрах говорит, что из исходной строки надо удалить два символа, а тройка – что удалять символы надо, начиная с третьей позиции в строке. Если что-то можно удалить, то обычно где-то рядом должна быть и возможность обратного процесса.

Вот она, возможность – под названием Insert. Три параметра – что вставить, куда, в какое место. То есть:

```
s1:='12345';  
s2:='abc';  
Insert( s2, s1, 3);  
writeln(s1);
```

Получим 12abc345.

Вот и вся компания. Применяется этот зоопарк обычно не по одному, а в совокупности.

```
s1:='x=f(y)';  
s2:='z=x';
```

Задача – подставить во вторую строку значение X из первой.

```
s3:=s2;  
Delete( s3, Pos('=' ,s2)+1, 999);  
Insert( Copy(s1,Pos('=' ,s1)+1,999), s3, Pos('=' ,s3)+1);
```

В результате в s3 должны получить

```
`z=f(y)';
```

Магическое число 999 означает, что мы имеем в виду все символы до конца строки. Copy и Delete, если третий параметр больше, чем в строке осталось символов, понимают его, как количество символов до конца строки.

## Строка и цикл

С помощью процедур из предыдущего раздела и циклов можно решить любую задачу обработки строк, хотя часто и несколько противоестественным способом. Например, надо подсчитать количество пробелов в строке. В цикле – находим пробел с помощью Pos, увеличиваем счётчик, удаляем пробел с помощью Delete, снова находим пробел... И так будем продолжать, пока строка не кончится. Цикл, правда, нужен немного другой, об этом ещё будет позже. Потом. В любом случае, задача решаема, но в жизни подсчёт пробелов в строке решается совсем по-другому – просто берём и считаем. Без затей.

Важная идея заключается в том, что со строкой можно обращаться в точности, как с массивом. Или почти. Например: `s[5]:='a'`; - в пятом символе строки будет символ 'a'. Аналогично `if s[5]='b'` – проверяем – а не равняется ли случайно пятый символ строки символу 'b'?

Если мы написали `s:='12345'`; то у нас в строке ровно пять символов. Мы можем обращаться к символам от первого до пятого, но не более того. Нельзя написать `s[6]:='6'`, это плохо кончится. Надо так `s:=s + '6'`;

А теперь, вооружённые новыми знаниями, считаем пробелы. Перебираем все символы строки от первого до последнего. Проверяем – если пробел, значит наращиваем количество. Пишем цикл. Цикл от первого символа – это понятно. До последнего символа – вспоминаем функцию Length. Как проверять символ на равенство пробелу, вы только что узнали. В результате имеем:

```
num:=0;
for i:=1 to Length(s) do begin
  if s[i]= ' ' then num:=num + 1;
end;
```

А теперь задача намного сложнее – подсчитать, сколько раз в строке встречается определенный артикль 'the'. Задача похожа на только что

решённую, но проверять надо не только текущий символ, но и два последующих. Если текущий символ имеет индекс I, то у следующего будет индекс I+1, а у того, что после этого I+2. В целом получится что-то вроде:

```
num:=0;
for i:=1 to Length(s) do begin
    if (s[i]='t') and (s[i+1]='h') and (s[i+2]='e')
        then num:=num+1;
end;
```

Почему *что-то вроде*? Потому что когда переменная цикла I будет указывать на последний символ строки Length(s), мы попытаемся проверить ещё и следующий символ (I+1) и ничего хорошего из этого не получится. А с другой стороны – разве может слово из трех букв начинаться в последнем символе строки – ясно, что нет, ведь не уместится же. Не может начинаться и в предпоследней, по той же причине (так что рухнет наша программа ещё раньше, при I, равном Length(s)-1. А раз оно там начинаться не может, так зачем проверять? Урезаем цикл на два исполнения:

```
num:=0;
for i:=1 to Length(s)-2 do begin
    if (s[i]='t') and (s[i+1]='h') and (s[i+2]='e') then
num:=num+1;
end;
```

Может, у кого-то затесалась в голову мыслишка, что, ежели мы обнаружили слово “the”, начинающееся с позиции I, то проверять следующие две позиции на наличие символа ‘t’ – пустое расточительство? Немедленно топтать мыслишку коваными сапогами. Нам не надо, как лучше – нам надо чтобы работало. Работает – отойди, не трогай.

На самом деле при таком подходе мы сосчитаем и все слова типа then и there, содержащие внутри себя последовательность символов the, но не будем обращать внимание на такие мелочи. Пока.

Помните, как считается сумма элементов массива?

```
sum:=0;
for i:=1 to N do
```

```
sum:=sum + a[i];
```

У строки ведь тоже есть оператор “+”, хотя он выполняет не арифметическое сложение, а склейку строк. Простая задача – заполнить строку одним и тем же символом, например ‘-’. Всё аналогично. В начале цикла целой переменной присваивается значение ноль.

А теперь об очень важном! Что является аналогом ноля для строки? Пустая строка, то есть строка, в которой нет ни одного символа. Обозначается это чудо вот так ‘’ – две кавычки подряд, без пробела между ними. И вот текст программы:

```
s:='';  
for i:=1 to N do  
  s:=s + '-';
```

Обратите внимание, мы заполнили строку, в которой до того не было ничего. Если мы хотим заполнить каким-то символом строку из предыдущего примера, то есть уже заполненную чем-то, то выглядеть это будет так:

```
for i:=1 to Length(s) do  
  s[i]:='*';
```

Очень похоже на массив. Обдумайте. Ещё две задачи, которые в разных вариациях будут встречаться весьма часто.

Удалить из строки (например, ‘123 456 789’) все пробелы. Напрашивается быстрая модификация недавно созданной программы:

```
s:='123 456 789';  
for i:=1 to Length(s) do begin  
  if s[i]= ' ' then Delete( s, i, 1);  
end;
```

В результате имеем на выходе: ‘123456789’. Замечательно! А теперь из вредности подсовываем на вход вот такую строку – ‘123 456 789’ (между группами по два пробела). Результат – ‘123 456 789’. Что-то пошло не так, пробелы уменьшились в количестве, но всё-таки остались. Почему? Очень просто. Сначала проверили четвёртый символ, потом пятый. Четвёртый оказался пробелом, его удалили. Но при этом вся остальная

часть строки сдвинулась влево и нумерация оставшихся символов изменилась. Следующий пробел, бывший пятым, стал четвёртым. Но четвёртый символ мы уже проверили и переходим к пятому, а туда съехала цифра 4 из бывшей шестой позиции. Вот так всё неудачно получилось. Что делать будем?

Если мы начнём просматривать строку с конца к началу, то, удалив первый встреченный пробел - последний в строке, мы изменим нумерацию всех символов справа, но там-то мы уже были и туда нам больше не нужно. А с символами слева, куда мы направляемся, всё останется в порядке.

```
s:='123 456 789';  
for i:=Length(s) downto 1 do begin  
    if s[i]=' ' then Delete( s, i, 1);  
end;
```

Если у нас цикл движется от большего значения к меньшему, то, вместо **to**, должно использоваться **downto**. Запомните – удаление чего-то из чего-то пронумерованного обычно происходит, начиная с конца этого второго чего-то. Такое ненарушаемое правило.

А теперь задача - подсчитать количество слов в строке. Количество пробелов между словами – произвольное, то же с пробелами в начале и в конце строки, так что идея просто подсчитать пробелы нас к решению не приблизит. Задача это встречается чаще, чем кажется с первого взгляда. Но сначала придётся отвлечься и ввести ещё один тип данных. Тянул я с этим до последнего, но теперь пора.

### Ой, кто пришёл!

Встречаем – логические переменные, они же булевские. Зачем нужны? Да незачем, как обычно. Вполне можно обойтись. Молоток, он, в общем-то, излишняя сущность. Гвозди вполне можно и топором заколачивать. В переводе на реалии программирования – забываем про булевский тип, объявляем целую переменную и договариваемся, что будем присваивать ей строго только ноль или единицу. Причем, опять же для себя, решаем, что ноль значит *нет*, единица значит *да*. Ведь мы это уже делали – когда сортировали массив! А теперь ещё раз, проиллюстрируем на другом примере.

Хотелось бы знать – а не состоит ли наша строка полностью, только, и исключительно из одних пробелов? Вопрос очень реальный. Вспомнив, как мы считали пробелы в строке, можно додуматься до такого варианта: подсчитать пробелы в строке и, если их количество совпадёт с длиной строки – то да. А если нет – то нет.

```
num:=0;
for i:=1 to Length(s) do begin
  if (s[i]= ' ') then num:=num + 1;
end;
if num=Length(s)
  then writeln('Yes!')
  else writeln('No!');
```

Работает, конечно. Но как-то неязучно. И зачем делать лишнюю работу? Зачем считать пробелы и помнить (до выполнения условного оператора) сколько их? У нас же никто про это не спрашивал. Вопрос был совсем о другом. Пишем вот так - onlySpaces, понятно, целое и означает, что строка только из пробелов:

```
onlySpaces:=1;
for i:=1 to Length(s) do begin
  if (s[i]<> ' ') then onlySpaces:=0;
end;
if onlySpaces = 1
  then writeln('Yes!')
  else writeln('No!');
```

А теперь доведём наш шедевр до полного блеска булевскими переменными. Сначала теория, разумеется. Объявляем:

```
var
  onlySpaces: boolean;
```

Булевская переменная может получать только два значения:

```
onlySpaces:=true;   { да}           ИЛИ
onlySpaces:=false; { нет}
```

Использовать переменную в условном операторе можно аналогично целой:

```
if onlySpaces = true   { да}           ИЛИ
```

```
if onlySpaces = false    {нет}.
```

Но так никогда не пишут, поскольку ради булевских переменных разрешена, для наглядности, сокращённая форма записи:

```
if onlySpaces    { да}           ИЛИ  
if not onlySpaces {нет}.
```

Окончательный вариант - onlySpaces объявлено как boolean:

```
onlySpaces:=true;  
for i:=1 to Length(s) do begin  
    if (s[i]<> ' ') then onlySpaces:=false;  
end;  
if onlySpaces  
    then writeln('Yes!')  
    else writeln('No!');
```

Обратите внимание – это важно! Если нам надо определить, есть ли в строке хоть один пробел (для умных – квантор существования), то признак перед началом цикла устанавливается в ложь (false), а при встрече первого пробела устанавливается в истину (true). Напротив, если надо определить, все ли символы в строке пробелы (для умных – квантор общности), то символ устанавливается перед циклом в истину, а при встрече первого *не пробела* устанавливается в ложь.

### Считаем, наконец, слова

Строка состоит из слов, разделённых пробелами. Символы, из которых состоят слова – не пробелы. То есть, запятая у нас не разделитель, а часть слова. Впрочем, решение (когда мы его получим) тривиальным образом обобщается на совершенно произвольный набор разделителей.

Сначала решение на пальцах. Обнуляем счетчик слов. Движемся по строке от начала до конца. Как только начинается новое слово, увеличиваем счетчик на единицу.

Очень хорошее решение, главное – абсолютно правильное. Или выглядит правильным. Осталось разъяснить не вполне ясные моменты, до полного превращения их, неясных моментов, в моменты абсолютно ясные. Если вдруг окажется, что какой-то неясный момент не хочет разъясняться, значит, мы были неправы, и надо начинать сначала. Это ещё называется *метод пошаговой детализации*. Он же *разработка сверху вниз*.

Неясный момент у нас ровно один – как понять, что началось новое слово? Как вариант - встретился символ, в смысле не пробел, а до того был пробел - значит новое слово. Логично, но если перед первым словом нет ни одного пробела, получается особый случай. А особый случай на то и особый, что обрабатывать его приходится особо. Это плохо, не надо нам особых случаев.

Введём логическую переменную – есть слово или нет. Что значит *есть*? *Есть* – значит, слово встретили и движемся по нему. То есть, текущий символ, тот на который указывает переменная цикла – не пробел. Тогда наша логическая переменная имеет значение истина. А если движемся по пробелам, переменная имеет значение ложь. Таким образом, если переменная ложна и встретился не пробел, то мы нашли новое слово.

Думаем дальше. Наша логическая переменная принимает два значения, как ей и положено. Наш текущий символ, по сути, тоже принимает только два – или пробел, или не пробел. Подробности, какой именно не пробел, то есть, какой именно символ, нам совершенно не интересны. Итого имеем - два умножить на два - четыре сочетания.

Объявим переменные, напишем цикл по строке и по условному оператору на каждый вариант. А там поглядим.

```
var
  s           : string; {строка для разбора}
  inWord      : boolean; {признак - в слове или нет}
  count      : integer; {результат - количество слов}
  i           : integer; {это, понятно, переменная цикла}
.....
  count:=0;
  inWord:=false;
  for i:=1 to Length(s) do begin
    if inWord and (s[i]=' ') then begin {в слове и пробел}
  end;
  if inWord and (s[i]<>' ') then begin {в слове и не пробел}
  end;
  if not inWord and (s[i]=' ') then begin {не в слове и пробел}
  end;
  if not inWord and (s[i]<>' ') then begin {не в слове и не пробел}
  end;
end;
```

Теперь опять думаем. Как будет выполняться наша программа? Рано или поздно встретится первый не пробел. Это ситуация номер четыре. Наша реакция? Значит, началось новое слово – следовательно, увеличиваем счётчик на единицу. Слово началось, слово когда-то и кончится. То есть, мы находимся внутри слова и *внезапно* встретили пробел. Это ситуация номер один. Наша реакция – установить логическую переменную в ноль. Смотрим, что получилось.

```
count:=0;
inWord:=false;
for i:=1 to Length(s) do begin
  if inWord and (s[i]=' ') then begin    {в слове и пробел}
    inWord:=false;
  end;
  if inWord and (s[i]<>' ') then begin    {в слове и не пробел}
  end;
  if not inWord and (s[i]=' ') then begin{не в слове и пробел}
  end;
  if not inWord and (s[i]<>' ') then begin{не в слове и не пробел}
    inWord:=true;
    count:=count + 1;
  end;
end;
```

А дальше? А всё. Оставшиеся две ситуации нас не интересуют. А зачем мы их вообще выделили? Во-первых, для наглядности. Во-вторых, сейчас мы исходную задачу усложним.

Кстати, вспомогательная информация - время от времени встречаются операции типа увеличить счетчик на единицу. На этот случай предусмотрен особый оператор. Вместо count:=count + 1; можно написать Inc(count). Разумеется, предусмотрен и обратный вариант. Dec(count) уменьшит нашу переменную на единицу. И ещё, секций VAR может быть несколько. Только зачем?

А пока, оформите наш программный код в виде законченной программы и протестируйте её.

А теперь усложняем. Вернее, доводим до практического применения. Теперь наша задача – извлечь из строки слово с заданным номером. Если нет слова с заданным номером – вернуть пустую строку. Несколько позже мы (вы) оформим (оформите) это в виде процедуры (функции). Почти все переменные у нас уже есть, объявляем недостающее:

```

var
  needCount      : integer; {номер нужного слова}
  theWord        : string;  {результат - то самое слово}

```

Теперь надо как-то определиться с новыми переменными. theWord перед циклом должна быть проинициализирована пустой строкой, а значение needCount где-то перед этим уже введено пользователем программы. Снова глядим на уже обработанные ситуации.

Номер четыре – началось новое слово. В дополнение к тому, что уже там есть – проверить, а вдруг номер начавшегося слова совпадает с номером слова заказанного? Тогда текущий символ отправляем в результат. Номер один – ничего не изменилось. Кончилось слово, и ладно.

Первый символ слова мы уже пристроили. А остальные? Ясно, что номер третий для наших новых целей совершенно ни к месту. А вот номер два подойдёт – если мы по-прежнему в слове и номер слова тот, что надо, приклеиваем к результату текущий символ.

```

count:=0;
inWord:=false;
theWord:='';
for i:=1 to Length(s) do begin
  if inWord and (s[i]=' ') then begin {в слове и пробел}
    inWord:=false;
  end;
  if inWord and (s[i]<>' ') then begin {в слове и не пробел}
    if count = needCount
      then theWord:=theWord + s[i];
  end;
  if not inWord and (s[i]=' ') then begin {не в слове и пробел}
  end;
  if not inWord and (s[i]<>' ') then begin {не в слове и не пробел}
    inWord:=true;
    count:=count + 1;
    if count = needCount
      then theWord:=s[i];
  end;
end;
end;

```

Доделайте. Проверьте. Подумайте, как быть, если разделитель может быть не один? Например, пробел и запятая? Никаких проблем? А если разделители – пробел, запятая, точка, двоеточие, тире, точка с запятой, восклицательный знак, вопросительный знак?

## Глава 7, продолжение пятой Ещё циклы и массивы

### Массивы двумерные и далее

Массивы без циклов – вещь бесполезная, это я уже говорил, а двумерные массивы без вложенных циклов – ну даже и не знаю, нечто бесполезное в квадрате, и в высших степенях, в зависимости от размерности массива. Но раз я решил излагать в таком порядке – циклы сначала, массивы потом – значит так тому и быть.

Три составляющие применения любых данных – как объявить, как присвоить и как вывести. Ещё неплохо понять, что это вообще такое и зачем надо. Что такое и зачем целые и дробные переменные – понятно. Строки – в целом тоже. А массивы, тем более двумерные?

Массивы, как было упомянуто ранее, используются для хранения и обработки большого количества однотипных данных. Пример из любого учебника – если наша программа помнит и что-то делает с температурой воздуха сегодня, вчера и позавчера, достаточно объявить три переменные типа `single`. Если надо обрабатывать температуру за месяц, то объявлять тридцать одну переменную – ну неправильно это как-то.... Для этого массивы есть – `array[1..31] of single`;

А если всё ещё хуже и требуется хранить температуру не просто по каждому дню, но ещё и по каждому часу дня? – оцените, как плавно и ненавязчиво я подвёл разговор к теме двумерных массивов. Объявляем двумерный массив – почти как одномерный.

```
var  
    deg      : array[1..31,1..24] of single;
```

`of single` – это понятно, массив состоит из дробных чисел. А вот в квадратных скобках содержимое удвоилось. Индексов стало два. Первый может меняться от 1 до 31 – это, как можно догадаться, дни. Второй, от 1 до 24, очевидно часы. А сколько всего элементов массива? Правильно, 31 умножить на 24 равняется 744.

Теперь присваиваем

```
deg[1,1]:=10.5;
```

```
deg[31,24]:=12.5;
```

Всё точно так же, как и с одномерным массивом, только работать вручную стало ещё неудобнее – элементов в массиве намного больше. С выводом разберитесь сами. Переходим к циклам.

## Вложенные циклы

Напоминаем. Вот цикл:

```
for i:=1 to 3 do begin
  writeln('Au');
end;
```

Внутри цикла может находиться любой оператор, или несколько операторов. А раз любой – значит, этим оператором может быть и оператор цикла, внутри которого может быть... и так далее. Называется это – вложенный цикл. Тот цикл, что снаружи – внешний цикл, тот цикл, что внутри – внутренний цикл. На самом деле, не внутри оператора цикла, а внутри операторных скобок, – но какая разница...

Например, можно вот так:

```
for i:=1 to 3 do begin
  for j:=1 to 3 do begin
    writeln('Au!');
  end;
end;
```

После запуска получим:

```
Au!
Au!
Au!
Au!
Au!
Au!
Au!
Au!
```

Очень важно! Во внешнем и внутреннем циклах переменные цикла должны быть обязательно разные! По традиции, это I и J.

Как нетрудно убедиться, наша строка выведена ровно девять раз. Внешний цикл выполняется три раза, при каждом его выполнении три раза выполняется внутренний цикл - три раза по три будет девять. А если написать вот так:

```
for i:=1 to 9 do begin
  writeln('Au!');
end;
```

получим абсолютно тот же результат. Так зачем два цикла, если можно один? Совершенно верно, в данном конкретном случае абсолютно незачем. Слегка подправим программу:

```
for i:=1 to 3 do begin
  for j:=1 to 3 do begin
    writeln('Au!');
  end;
  writeln;
end;
```

Получим вот что:

Au!  
Au!  
Au!

Au!  
Au!  
Au!

Au!  
Au!  
Au!

Три группы по три строки, разделённые пустой строкой. Конечно, можно было бы слегка извратиться и обойтись один циклом – где-то вот так:

```
for i:=1 to 9 do begin
  writeln('Au!');
  if (i mod 3) = 0 then writeln;
end;
```

Но лично мне кажется, что вариант с двумя циклами естественнее – но допускаю, что я неправ. Тогда *ещё* чуть-чуть усложним – пусть в начале каждой непустой строки стоит номер группы, к которой она относится.

```
for i:=1 to 3 do begin
  for j:=1 to 3 do begin
    writeln(i, '.Au!');
  end;
  writeln;
end;
```

На выходе:

```
1.Au!
1.Au!
1.Au!
```

```
2.Au!
2.Au!
2.Au!
```

```
3.Au!
3.Au!
3.Au!
```

Можно, конечно, ещё слегка извратиться и опять обойтись одним циклом, но, по-моему, не стоит. Теперь простенькая программка. Смысл очевиден – выводим значения переменных цикла, смотрим, как они меняются и думаем.

```
for i:=1 to 3 do begin
  for j:=1 to 3 do begin
    writeln('i=', i, '      j= ', j);
  end;
  writeln;
end;
```

```
i=1  j=1
i=1  j=2
i=1  j=3
```

```
i=2  j=1
```

i=2 j=2

i=2 j=3

i=3 j=1

i=3 j=2

i=3 j=3

Если Вам это совершенно очевидно, то очень хорошо, иначе – думаем, думаем... А теперь пишем *три* вложенных цикла, выводим значения управляющих ими переменных и снова думаем, думаем... По традиции, третью переменную цикла зовут К.

### Пример посложнее

Номер 121 из задачника Культина. Вывести таблицу умножения девять на девять в виде квадрата, она же почему-то называется таблицей Пифагора. Берём тетрадку в клеточку, переворачиваем обратной стороной и вот она, таблица:

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	54	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Я это чудо только что нарисовал ручками в Ворде, и мне это занятие не понравилось. Попробуйте сами, чтобы оценить, насколько легче будет в Турбо Паскале. Конечно, те читатели, у кого есть хоть какой-то задачник, знают, что в конце задачника есть ответы. В задачнике Культина они тоже есть. Это, разумеется, не наш путь. Наша цель не в том, чтобы получить решение - оно ведь в ответах есть, а в том, чтобы понять, как до него, до решения, добраться наиболее естественным путём. Приступим.

Начнем с изготовления того, что внутри у таблицы, а строчку сверху и столбец слева, обозначающие, что на что умножать, пририсуем после. Первым делом, надо решить вопрос, как будем рисовать – по строкам или по столбцам? Мне кажется, что лучше построчно, аргументировать не буду. Внешний цикл выполняется девять раз, и каждый раз при этом выводится по одной *строке*, внутренний цикл выполняется девять раз и при этом каждый раз выводится по одному *числу*. Проверка: девять на девять равняется восемьдесят один, а в таблице чисел как раз столько. Я проверял. Всё сходится. Так что имеем:

```
for i:=1 to 9 do begin {цикл по строкам}
  for j:=1 to 9 do begin {цикл по столбцам}
    {здесь что-то делаем}
  end;
end;
```

Теперь не хватает только того, что должно быть внутри цикла – это и есть самое главное, правда. Начнем с малого и простого – выведем самую первую строку, ту в которой числа 1,2,3...9. Тут всё хорошо. Цикл от одного до девяти у нас уже есть, значение того, что надо вывести, удивительным образом совпадает со значением переменной цикла. Поскольку все числа выводятся в одной строке, подойдёт оператор `write(i)`; не переходящий после вывода на новую строку. Чтобы все числа не склеились в одно длинное слово, укажем количество выводимых знаков – у нас это будет четыре – одна/две цифры и три/два пробела спереди. А поскольку на следующую строку перейти всё-таки когда-то надо, добавим после завершения внутреннего цикла оператор `writeln`; без параметров.

```
for i:=1 to 9 do begin {цикл по строкам}
  for j:=1 to 9 do begin {цикл по столбцам}
    write(j:4);
  end;
  writeln;
end;
```

В результате получили повторенную девять раз одну и ту же строку –

```
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
```

и т.д.

Логично, ведь от переменной I, управляющей выводом по строкам, наш оператор `write(i:4);` никак не зависит. А как он должен зависеть? Элементарно – число на пересечении I-й строки и J-го столбца представляет собой их произведение. Делаем вот так - `write(i*j:4);`  
Гораздо лучше –

```
1  2  3  4  5  6  7  8  9
2  4  6  8 10 12 14 16 18
3  6  9 12 15 18 21 24 27
```

Собственно, то, что и хотели получить.  
Добавить строчку сверху труда не составляет, надо только не забыть добавить соответствующее число пробелов перед ней и пустую строку после неё.

```
write(`      `);
for j:=1 to 9 do
  write(j:4);
writeln;
writeln;
```

Переменная цикла J использована из принципиальных соображений – раз она у нас в основном цикле отвечает за перемещение по горизонтали – так путь отвечает за это везде. **begin** и **end** отсутствуют, напоминая, что можно и так. Почему `writeln` в двух экземплярах догадайтесь сами.

Теперь добавим вертикальную строчку. Вывести её в отдельном цикле – мысль не очень хорошая – а почему, в чём проблема?. Можно конечно теоретически использовать `GoToXY`, но зачем нам лишние приключения? Рассуждаем так. Где должна стоять цифра? Перед строкой с результатами. Одна строка – одна цифра. *Перед* строкой – значит *перед* циклом, выводящим строку. Цифра соответствует номеру строки, который соответствует значению переменной цикла I. И вот, наконец, программа:

```
write(`      `);
for j:=1 to 9 do
  write(j:4);
writeln;
writeln;

for i:=1 to 9 do begin {цикл по строкам}
```

```

write(i:4);
for j:=1 to 9 do begin {цикл по столбцам}
    write((i*j):4);
end;
writeln;
end;

```

А теперь, самостоятельно, бодро-весело рисуем шахматную доску. И подумайте, как посимпатичнее выбирать, в какой цвет какой квадрат покрасить. Посимпатичнее, не в смысле, чтобы красный и зеленый, а чтобы соответствующий условный оператор был бы не слишком страшен. Если сделаете легко и быстро, то напишите на каждой клетке её обозначение – E2, E4 и тому подобное.

### О самом важном. Всё сразу и побольше

Циклы и массивы созданы друг для друга. Двумерные массивы и вложенные циклы – вообще, наверное, символ мировой гармонии. Друг без друга им просто не жить.

Сначала несколько простых примеров, из серии *это должен знать каждый*. Массив предполагается объявленным как

```

var
    a          : array[1..N,1..M] of integer;

```

То есть, массив вовсе даже не квадратный – обратите внимание!

Задача номер раз. Заполнить массив нулями:

```

for i:=1 to N do
    for j:=1 to M do
        a[i,j]:=0;

```

Номер два. Посчитать сумму элементов массива:

```

sum:=0;
for i:=1 to N do
    for j:=1 to M do
        sum:=sum + a[i,j];

```

Три. Найти минимальный элемент массива:

```

min:=a[1,1];

```

```
for i:=1 to N do
  for j:=1 to M do
    if a[i,j] < min
      then min:=a[i,j];
```

Наверное, вам уже всё понятно. Вообще, двумерные массивы очень хорошо проецируются на математические реалии. Двумерный массив – это матрица, а количество математических манипуляций с матрицами воистину безгранично и даёт очень важные и полезные результаты. А ещё это напоминает мне о молодости и курсе высшей алгебры и, вообще, небо тогда было голубее, трава зеленее, но девки сейчас лучше! Или, говоря по другому, лучше сейчас!

А Вам две задачки для самостоятельной работы.

Первая. Повернуть квадратный массив набок. То есть, было:

```
1 2 3
4 5 6
9 8 9
```

Должно получиться:

```
9 4 1
8 5 2
9 6 3
```

Вторая задача сложнее. Заполнить квадратный массив *змейкой*, начиная с левого нижнего угла. Для массива размерностью 5x5 на выходе будет:

```
11 19 20 24 25
10 12 18 21 23
4 9 13 17 22
3 5 8 14 16
1 2 6 7 15
```

Разумеется, ваша программа должна работать с квадратным – только квадратным - массивом любой размерности. Как это технически будет реализовано? Как уже сказано, в Паскале размерность массива – константа. В некоторых других языках в этом месте может быть и переменная, но у нас допустима только константа. Что делать? Например,

объявить большой-большой массив, сто на сто, например, и передать в процедуру в качестве параметра его фактическую размерность, то есть какую часть из него заполнять. Некрасиво? Учите Фортран, там и переменные в этом качестве разрешены.

## Другие циклы

Циклы, с которыми мы встречались до сих пор, в одном отношении были совершенно одинаковы. Они могли выполняться пять раз, десять раз, сто, тысячу или сто тысяч миллионов раз, выполняться от единицы до миллиона или от миллиона до единицы, но число выполнений цикла нам было известно заранее, ещё при написании программы.

Упрощаю, конечно. Число выполнений цикла могло и не быть константой. Могло быть и переменной. Но с математической точки зрения – какая разница? Всё равно значение переменной этой известно *до начала выполнения* цикла. Это важно – количество выполнений цикла *нельзя* изменить внутри цикла. Прошу прощения у чистых математиков за вольную трактовку понятий переменной и константы. Конечно, если они, чистые математики, их вообще понимают... *Чистые* это не в смысле не *грязные*, это в смысле не *прикладные*.

А если число выполнений заранее неизвестно? А почему? Самый частый случай – когда есть живой пользователь. Например, запрограммировали мы игрушку. И пользователь наш радостно долбит по клавишам, устанавливая справедливость в одном отдельно взятом подвале. И когда он, зараза, утомится долбить по клавишам и захочет выйти из цикла и из игры, предугадать нам не дано. Вот тут и нужен цикл, который выполняется столько раз, сколько не знаем заранее.

К монстрам, подвалам и справедливости вернемся позже, а пока попытаемся придумать более удобное для программиста, то есть без живого пользователя, применение для такого рода циклов. К сожалению, на ум идёт исключительно математика, причем не простая, а всё как-то высшая. Постараемся, однако, обойтись без жертв. Напишем цикл, самый обычный:

```
for i:=1 to 5 do  
  writeln((1/i):5:3);
```

На выходе получим

1.000  
0.500  
0.333  
0.250  
0.200

Если бы мы вывели эти значения в виде не десятичных дробей, а натуральных, то получили бы следующее:

$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}$

Эта последовательность называется гармоническим рядом. Как её продолжить дальше, вопросов не вызывает –  $\frac{1}{6}, \frac{1}{7}, \frac{1}{8} \dots$  У неё есть одна особенность – если сложить достаточное количество его членов, то в сумме можно получить любое заранее заказанное число.

Чуть математики. Это значит, что ряд расходится. Полюбуйтесь на ряд  $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16} \dots$  Он сходится. Это не просто значит, что он не расходится. Это значит, что его сумма бесконечно и сколь угодно близко приближается к некоему числу. Осознание и даже доказательство этого факта лежит в пределах понимания любого не совсем деградировавшей особи. Слово *особь* не ругательное, а использовано исключительно для исключения возможности какой-либо гендерной сегрегации. Слово *гендерный* тоже не ругательное. Кстати, ряды бывают ещё не расходящимися и не сходящимися одновременно.

Поставим себе такую задачу, вовсе не искусственную, а вполне себе встречающуюся в реальной жизни. Ввести какую-то значение (сумму), а затем определить, сколько элементов ряда нам надо просуммировать, чтобы эту сумму достичь. Начинаем с простого – объявляем необходимые переменные, вводим ввод и выводим вывод:

```
program series;  
  uses  
    Crt;  
  var  
    sum           : single; {сколько надо получить в сумме}  
    factSum      : single; {сколько фактически имеем в сумме}  
    x            : single; {текущий член ряда}  
    howMany      : integer; {ответ - сколько членов просуммировали}
```

```

i          : integer; {а это переменная цикла, которая нам }
              {вообще не понадобится}
begin
  ClrScr;
  write( 'Sum = ');
  readln(sum);

  {а вот тут получаем ответ}

  writeln('how many = ' howMany);
end.

```

Если выразить человеческим языком то, что от нас требуется, получим примерно следующее: вначале ответ равен нулю. Затем, пока сумма меньше требуемой, вычисляем очередной член ряда, прибавляем его к сумме, увеличиваем ответ. Так *до тех пор*, пока фактическая сумма не превысит сумму требуемую. Обратите внимание на выделенные слова. Теперь запишем это на Паскале, и будем разбираться.

```

howMany:=0;
factSum:=0;
while (factSum < sum) do begin
  howMany:=howMany + 1;
  x:=1/howMany;
  factSum:=factSum + x;
end;

```

Извлекаем самое важное:

```

while (условие) do begin
  {чего-то делаем}
end;

```

Условие здесь – самое обычное условие, такое же, как и в условном операторе. То, что внутри цикла (там, где сейчас комментарий), будет выполняться до тех пор, пока условие истинно. В нашем случае – пока фактическая сумма не достигла требуемой величины. То есть – проверили условие, выполнили цикл, проверили условие, выполнили цикл, и так пока условие вдруг не окажется ложным. Обратите внимание – сначала проверили условие, затем выполнили цикл. Утром деньги, вечером стулья. А если нет денег, в смысле условие не выполняется? Не будет и выполнения цикла. Может быть и такая ситуация, когда цикл не выполнится ни разу. В нашем примере – если захотеть набрать нулевую или, того хуже, отрицательную сумму. Зачем что-то суммировать – ноль у нас и так уже с самого начала есть.

А можно утром стулья? Можно. Разработчики Паскаля, исходя из высших педагогических соображений, предусмотрели и этот вариант. На мой взгляд, можно было бы и обойтись. Лучше бы они предусмотрели оператор возведения в степень. Но это я так, к слову.

Цикл, который сначала выполняется, а потом думает – а стоило ли оно того? Выглядит он сложнее и загадочнее. Перепишем нашу программу с его применением.

```
howMany:=0;  
factSum:=0;  
repeat  
  howMany:=howMany + 1;  
  x:=1/howMany;  
  factSum:=factSum + x;  
until (factSum >= sum);
```

Исполняемая часть цикла теперь заключена между **repeat** и **until**. Условие поменялось на противоположное – это принципиально! Цикл теперь выполняется не до тех пор, пока условие *истинно*, а до тех, пока оно *ложно*. Как только условие станет истинным – исполнение цикла прекратится. Зато цикл выполнится как минимум один раз. Обратите внимание, возможно окажется полезным. Вот такой цикл **while** будет выполняться вечно:

```
while true do begin  
  {что-то вредное}  
end;
```

А вот такой цикл вообще ни разу:

```
while false do begin  
  {что-то полезное}  
end;
```

Вариант **repeat-until** и его вечно выполняющийся цикл

```
repeat  
until false;
```

Для цикла **repeat-until** вариант, не выполняющийся ни разу, по понятным причинам невозможен.

А теперь скрестим... Кого мы ещё не скрещивали? Ну, например двумерные массивы и только что освоенные *другие* циклы.

Есть такая игра – пятнашки. Напоминаю, кто не знал, да ещё и забыл – коробочка, рассчитанная на шестнадцать фишек, четыре на четыре. Но фишек не шестнадцать, а пятнадцать, одно место в коробочке остаётся пустым. На фишках числа – от единицы до пятнадцати. Фишки, тщательно перетасовав, помещают в коробочку. Задача – не вынимая фишек из коробки, только передвигая, пользуясь при этом для *маневра* или *манёвра* свободной клеткой, восстановить нормальный порядок вещей – расставить фишки по рядам от первой до пятнадцатой. Показываю на примере:

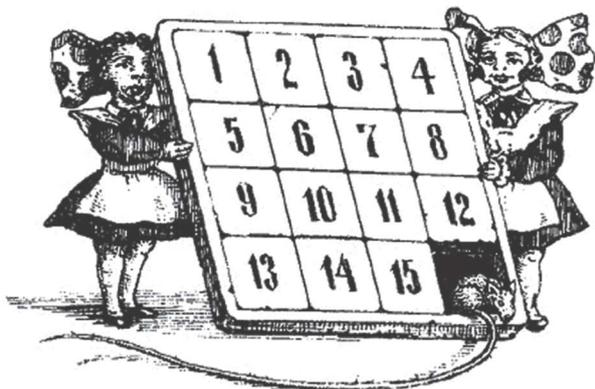
До того:

12	13	1	4
3	15	6	9
14	2	5	6
7	8	10	

После того:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Вот картинка, картинку я взял из книги Перельмана, того, не этого, *Живая математика*. Издание ещё 30-х годов, так сто никаких копирайтов быть не должно.



Это головоломка в финале, в собранном и упорядоченном состоянии. На данный момент, на начало игры, наша задача – перетасовать. То есть, случайным образом расставить в коробочке пятнадцать фишек. Какие варианты? Вариантов, как часто случается, ровно два.

Или перебираем все первые пятнадцать свободных посадочных мест в коробке. Случайным образом определяем на первое свободное место одну из фишек, вначале – одну из пятнадцати. Определив фишку на место, вычёркиваем её из числа доступных к расстановке, и, в следующий раз, уже выбираем одну из четырнадцати оставшихся. И далее пока все не кончатся. Вариант номер два. Берём фишку, от первой до пятнадцатой, пытаемся её поставить в коробочку, совершенно куда попало. Если место свободно, то уже хорошо. Если занято, пытаемся снова. Когда-нибудь, повезёт.

Мне больше нравится вариант номер два. В первом случае надо помнить, какая фишка уже поставлена, а какая ещё нет. Во втором случае у нас цикл по фишкам от первой до пятнадцатой – и помнить ничего не надо, потому что нечего помнить. Нет, конечно, надо помнить, где какая фишка уже стоит, но это ведь надо помнить в любом случае? Ведь это наша цель. Излагаем словесно:

Цикл по количеству фишек

Цикл пока не найдём свободного места для фишки

Попытаться фишку пристроить

Просто и ясно. Первый цикл, как уже много раз было сказано, от единицы до пятнадцати, значит - **for**. Для второго цикла очень даже подойдёт **repeat – until**. Ведь как минимум одну попытку расстановки предпринять мы должны.

Не совсем хорошая особенность нашего алгоритма – мы не можем гарантировать, что он завершится. Ну бывает же так, что каждый раз - и мимо! Конечно, вероятность незавершения нашего алгоритма за сколько-то заметное время сопоставима с вероятностью одномоментного выхода всех молекул кислорода из комнаты. С одной стороны, это вряд ли. С другой стороны, если бы мы программировали ракету на Марс, за такое программирование я лично поотрывал бы все программистские ручонки. Но у нас не ракета, сойдёт. В результате:

```
for i:=1 to 4 do
  for j:=1 to 4 do
    a[i,j]:=0;

for i:=1 to 15 do begin
  repeat
    col:=Random(4) + 1;
    row:=Random(4) + 1;
    if a[row,col] = 0 then begin
      a[row,col]:=i;
      nasli:=true;
    end
    else nasli:=false;
  until nasli;
end;
```

Обсуждение. Во-первых, ждите мою книгу, где я объясню теорию вероятностей весело, в подробностях и понятно для кого угодно. Далее. Числа 4 и 15 употреблены для большей наглядности нашей небольшой программы. В реальной жизни вместо них, конечно, должны быть константы. Массив А вначале заполнен нулями. Ноль означает, что в соответствующей ячейке фишки нет. Если фишка там присутствует, то элемент массива содержит её номер, то есть то, что на фишке написано. Обратите внимание на использование функции Random.

Продолжение последует после освоения работы с клавиатурой.

## Глава 8 Процедуры и функции

### Процедура без параметров

Помните мою печаль по поводу циклов без массивов? Их бессмысленности и бесполезности? В чём-то похожая ситуация с процедурами без параметров. Ситуация похожая, но не до конца – процедуры без параметров действительно жизненно необходимы. Но не тем, что они, подобно правильно применённым циклам, сокращают текст программы и, тем самым, труд программиста. Процедуры без параметров, в большинстве случаев, текст не сокращают нисколько, а вот труд программиста сокращают очень даже – но не на этапе кодирования, а на этапе отладки и даже позже. Поймёте тоже позже.

Процедуры *структурируют* программу. А по простому? А по-простому – делают её понятнее. А это важнее всякой краткости.

Теперь пример. Пример абсолютно бесполезный, бессмысленный, абстрактный, и, поэтому, на примере текстового вывода. Привожу всю программу целиком, поскольку здесь особо важна структура программы, в смысле – что за чем идёт.

```
program proc;
  uses
    Crt;
{-----}
procedure WriteSomething;
begin
  writeln('We don''t need no education');
  writeln('We don''t need no thought control');
  writeln;
  { (C) Pink Floyd 'The Wall' 1979, так, на всякий случай }
end;
{-----}
begin
  WriteSomething;
  WriteSomething;
  WriteSomething;
end.
```

На выходе имеем:

We don't need no education

We don't need no thought control

We don't need no education  
We don't need no thought control

We don't need no education  
We don't need no thought control

Пластинку заело, короче. Долго думаем. В целом понятно. Даже не знаю, что объяснять-то... Попробую всё же. Закомментируйте вот эти строки:

```
{WriteSomething;  
WriteSomething;  
WriteSomething;}
```

Запустите программу снова. Что случилось? Абсолютно ничего. Не в хорошем смысле, что ничего не изменилось, а в плохом – что ничего не вывелось. То есть, то, что находится между словами **procedure...begin...end**; само по себе не делает ничего. А называется это описанием процедуры, оно же тело процедуры. А вот то, что мы закомментировали, называется вызовом процедуры. Вызов процедуры без параметров эквивалентен тому, как если бы вместо него просто подставили тело процедуры - то, что между **begin** и **end**. Три вызова, как у нас – значит, тело процедуры подставили три раза.

Обратите внимание на поразительное сходство структуры процедуры и программы. Вся разница только в точке в конце процедуры. Или почти вся разница.... Как увидим чуть позже, в процедуре точно так же можно объявлять переменные. Это важно. Имя процедуре даётся по тем же правилам, что и программе и вообще переменным. На самом деле, текст процедуры не подставляется вместо её вызова. Но может и подставляться, в некотором смысле. Посмотрите на слово **inline** – возможно, когда-нибудь и пригодится.

Ну, вот, пожалуй, и всё о процедурах без параметров.

### То же и с параметрами

Сначала простенькая бесполезенькая программка для демонстрации идейки. Здесь и далее я буду писать только саму процедуру и её вызов, а

программу вокруг всего этого дорисуйте сами, до той степени, чтобы её можно было запустить.

```
procedure Out(      x : integer);  
begin  
    writeln(x);  
end;
```

Процедуру можно вызвать так:

```
Out(5);
```

На экране появится пятёрка.

А можно так:

```
y:=5;  
Out(y);
```

На экране опять появится пятёрка.

Теперь важные термины. `X : integer;` - это описание формального параметра. Сам `X` – это формальный параметр. То, что стоит в скобках при вызове процедуры – `5` или `Y` – это фактический параметр. Видим, что фактический параметр может быть константой или переменной. Имя этой переменной не обязано совпадать с именем фактического параметра, но, в общем случае, может. Об этом мы подробно поговорим позднее. Фактический параметр может быть и выражением:

```
y:=5;  
Out(y+1);
```

Выведется, понятное дело, шестёрка. Условие одно – фактический параметр, будь то константа, переменная или выражение, должен совпадать по типу с формальным параметром. В нашу процедуру нельзя передать дробное число или строку. Нельзя передать и выражение, принимающее дробное значение, например `5/2`. Соответственно, если формальный параметр строка, то фактический параметр обязан быть объявлен как **string** или быть строковой константой, но не как `integer` или `single`. Но если формальный параметр объявлен как `single`, на вход процедуре можно передать и целое число и целую переменную - целое в

данном случае рассматривается как частный случай дробного. Вспомните о правилах присваивания дробных и целых чисел. И здесь то же самое.

Теперь о сложном. Напишем вот так:

```
procedure Out(      x : integer);
begin
    x:=x*2;
    writeln('inside x ='x);
end;
.....
x:=5;
Out(x);
writeln('outside x =' ,x);
```

Что видим на экране?

```
inside x =10
outside x =5;
```

Во-первых, мы видим, что формальный параметр внутри процедуры можно использовать как самую обычную переменную, в том числе можно изменить его значение. Но изменение действует только в пределах самой процедуры. Вышли – забыли об изменениях.

Подробнее позже, очень скоро – в следующем разделе. А пока процедура посложнее – с несколькими параметрами.

Вспоминаем рисование квадрата методом опорной точки.

```
x0:=100;
y0:=100;
Line(x0,y0, x0+100,y0);
Line(x0+100,y0, x0+100,y0+100);
Line(x0+100,y0+100, x0,y0+100);
Line(x0,y0+100, x0,y0);
```

Переоформляем это в виде процедуры.

```
procedure Square(      x0,y0 : integer);
begin
    Line(x0,y0, x0+100,y0);
    Line(x0+100,y0, x0+100,y0+100);
    Line(x0+100,y0+100, x0,y0+100);
    Line(x0,y0+100, x0,y0);
```

```
end;  
.....  
Square(100,100);
```

Как легко догадаться, вам надлежит оформить это в виде законченной программы. Добавим третий параметр – цвет, которым надо рисовать квадрат. А какого цвет типа? Как несложно предположить из сказанного о цвете раньше – цвет целого типа, то есть `integer`;

Замечание в скобках. Догадка на самом деле неверная. То есть, цвет действительно целого типа, но не `integer`, а `word`. Разница в том, что `word` может принимать только положительные значения, зато в диапазоне от 0 до 65535. Тип `word` вам никогда не понадобится. Забудьте.

Добавляем цвет.

```
procedure Square(      x0,y0 : integer;  
                    color : integer);  
begin  
  SetColor(color);  
  Line(x0,y0, x0+100,y0);  
  Line(x0+100,y0, x0+100,y0+100);  
  Line(x0+100,y0+100, x0,y0+100);  
  Line(x0,y0+100, x0,y0);  
end;  
.....  
Square(100,100, Green);
```

Тип параметров указывается аналогично типу переменных. Однотипные параметры можно сгруппировать в одной строке, как вы уже заметили.

### А какие бывают параметры?

Теперь о сложных и серьезных материях. Кроме шуток. Какие бывают параметры у процедур? Двух видов.

С чем мы только что познакомились? В качестве фактического параметра можно передать хоть переменную, хоть константу или выражение, лишь бы оно совпадало по типу с формальным параметром. Или почти совпадало, как в случае целых и дробных. Если подать на вход переменную и изменить её внутри, по выходе из процедуры всё забудется. Это называется передача параметров по значению. Другой, незнакомый нам пока вариант – передача параметров по ссылке. Вообще

говоря, в других учебниках они, виды параметров, могут называться как угодно по-другому. Так что, забываем про глупости и встречаем по одежке. Как оно выглядит? Если так:

```
procedure Make( x : integer);
```

то это первый вариант, нам уже знакомый, передача по значению. На вход что угодно – переменная, константа, выражение, но изменения после выхода не сохраняются.

Если так:

```
procedure Make( var x : integer);
```

то это вариант номер два, нам пока незнакомый. Будем знакомиться. В качестве фактического параметра можно передать только и только переменную типа `integer`. Никаких констант, никаких выражений. Даже если бы формальным параметром был `single`, передать на вход целое всё равно нельзя. Зато если изменить значение переменной внутри процедуры, изменения сохранятся и после выхода наружу. Вернёмся к примеру из предыдущего раздела, только добавим `var`.

```
procedure Out( var x : integer);
begin
  x:=x*2;
  writeln('inside x ='x);
end;
.....
x:=5;
Out(x);
writeln('outside x =' ,x);
```

Что теперь мы получим на экране?

```
inside x =10
outside x =10;
```

Как этим счастьем пользоваться? Никогда нельзя допускать, чтобы в процедуру в качестве параметра с `var` входило одно значение, а возвращалось, в этом же параметре, другое значение. Это, разумеется, транслятором будет пропущено, но так делать низзя! Вход – параметр без `var`, выход – параметр с `var`. Получается, что параметры с `var` при входе в

процедуру вообще, по сути, не должны иметь никакого значения. Так оно и есть. Значение, всё равно присутствует, хотя бы и какое-нибудь мусорное, от этого никуда не деться. Но мы должны вести себя так, как будто его нет. Непонятно? Забудьте! Поймёте потом. Если станете программистом.

В качестве примера вспомним про когда-то подсчитанный нами факториал и изготовим из него процедуру. К факториалу нам придётся вернуться ещё минимум дважды – скоро, добравшись до функций, и нескоро, когда доползём до рекурсии. Напоминаю – факториал  $N$ , обозначается  $N!$ , представляет собой произведение всех целых чисел от 1 до  $N$ . И подсчитали мы его так:

```
N:=5;
fact:=1;
for i:=1 to N do
    fact:=fact * i;
```

Теперь изготовляем из этого процедуру с двумя параметрами.

```
procedure Fact(      N      : integer;
                  var result : integer);
var
    i      : integer;
begin
    result:=1;
    for i:=1 to N do
        result:=result * i;
end;
```

Обратите внимание. Мы впервые встретились с одновременным использованием параметров разных типов. И впервые – с объявлением переменных внутри процедуры.

Кстати, факториал какого (максимально) числа, мы можем подсчитать, если в качестве результата у нас используется переменная типа `integer`? Отмотайте назад, найдите какое максимальное значение может содержать `integer` и оцените. А потом оцените, целесообразна ли замена `integer` на `word`. По тому же алгоритму - отмотайте назад, найдите, сосчитайте... Не хочется? Скучно? Такова программистская жизнь...

## О грустном

Почему о грустном? Когда-то, когда я изучал Паскаль, совсем даже ещё не Турбо, всё было хорошо, просто и понятно. Всё, кроме передачи массивов в качестве параметров. Я как-то даже поверить не мог, что это настолько плохо и ожидал, что вот-вот и прочитаю в какой-нибудь книжке про человеческий способ сделать ЭТО. Не прочитал.

Что у нас есть в программе между **program** и первым **begin**'ом?

```
program
  uses
  const
  var
begin
```

Нам пока ни разу не понадобилась ещё одна секция. Начинается она словом **type**. Размещается обычно между объявлениями констант и объявлениями переменных. Это не строго обязательно, но обычно по другому никак. Предназначена секция для объявления пользовательских типов. Что это такое и зачем, подробно и осмысленно рассказывать здесь не буду, эта тема для совсем другой книги.

Сначала очень простой пример.

```
type
  TColor = integer;
```

Что это значит? Мы объявили новый тип по имени TColor. По существу новый тип является псевдонимом типа integer. Поняли? Нет? В сущности, пользовательские типы, то есть, типы, определённые пользователем, вещь абсолютно бесполезная, введённая исключительно из образовательно-дисциплинарных целей. Как ни старался, я не смог найти им ни малейшего применения. Преувеличиваю, конечно. Запись (record) вполне целесообразно объявить типом, и в дальнейшем ссылаться на него. Возможно, если постараться, можно вспомнить ещё пару-тройку случаев, когда пользовательские типы не совсем бесполезны. Но, в общем и целом, они таковой бессмыслицей оставались, до появления в Turbo Pascal 5.5 концепции объектно-ориентированного программирования. Тут, конечно, всё пошло совсем по-другому. Но об этом в другой книге. А пока, всё что мы получили от введения нового типа, это возможность вместо

```

procedure Square(      x0,y0 : integer;
                       color : integer);

```

написать

```

procedure Square(      x0,y0 : integer;
                       color : TColor);

```

Лично моё эстетическое чувство как-то травмировала необходимость передавать романтический цвет через какой-то математический `integer`. Теперь у нас всё красиво – цвет передаётся как цвет. К сожалению, если попытаться передать вместо `TColor` обычное целое, возражений не последует, как-то это неправильно. Сказано `TColor`, значит надо объявлять как `TColor`, и чтобы никаких целых. И обратите внимание на букву `T` в начале имени типа `TColor`. Есть хорошая традиция – все имена типов должны начинаться с буквы `T`. Так будет лучше, и вам, и окружающим.

А теперь объясню, к чему была вся эта теория. Захотели мы написать процедуру для подсчёта суммы элементов массива. Интуиция подсказывает, что сама процедура и её применение будут выглядеть примерно вот так:

```

const
  N = 5;
var
  a          : array[1..N] of integer;
  result     : integer;
{-----}
procedure Sum(      a          : array[1..N] of integer;
                  result : integer);
var
  i          : integer;
begin
  result:=0;
  for i:=1 to N do
    result:=result + a[i];
end;
{-----}
.....
{заполнили чем-то массив}
Sum( a, result);
{вывели ответ}

```

Что произойдёт при попытке транслировать этот, с виду вполне разумный, текст? Как говорится в народе, *пролетела птица обломинго* и приехал *литовский политический деятель Обломайтис*. Ничего хорошего не произойдёт. А как надо? А надо вот так:

```
const
  N = 5;
type
  TArray = array[1..N] of integer;
var
  a      : TArray;
  result : integer;
{-----}
procedure Sum(  a      : TArray;
               result : integer);
var
  i      : integer;
begin
  result:=0;
  for i:=1 to N do
    result:=result + a[i];
end;
{-----}
{заполнили чем-то массив}
Sum( a, result);
{вывели ответ}
```

Это надо просто запомнить. С многомерными массивами надлежит поступать точно так же. В смысле, сначала объявить их типами и только потом передавать как параметры.

### Скучная, но необходимая теория

Какие бывают параметры и как их правильно готовить, мы в целом разобрались. Теперь унылая тема под названием *Глобальные и локальные переменные*, тесно примыкающая к процедурам и параметрам. Когда-то тема эта была одной из важнейших при изучении, пожалуй что, любого языка программирования. По моему сегодняшнему мнению, тему надо открыть и тут же закрыть. В приличной программе глобальных переменных быть не должно. В приличной программе вообще много чего быть не должно. Подробнее на вопросе, чего не должен знать хороший программист, мы остановимся как-нибудь позже. Но чтобы глобальных переменных не допустить в программу, надо знать, что это такое. Врага надо знать в лицо!

Напишем такую совершенно абстрактную программу.

```
program global;
  var
    x,y          : integer;
  {-----}
  procedure MakeNothing;
    var
      x,z          : integer;
  begin
    x:=10;
    y:=20;
    z:=30;
    writeln('x = ',x:2, '   y = ',y:2, '   z = ',z:2);
  end;
  {-----}
  begin
    x:=1;
    y:=2;
    writeln('x = ',x:2, '   y = ',y:2);
    MakeNothing;
    writeln('x = ',x:2, '   y = ',y:2);
  end.
```

Запустите программу на выполнение. Посмотрите на результат. Подумайте. Сделайте выводы. Для читающих эту книгу не подходя к компьютеру и не вставая с дивана, привожу всё-таки результат её, программы, выполнения:

```
x = 1  y = 2
x = 10 y = 20  z = 30
x = 1  y = 20
```

Подумали? Сделали выводы? Первая строка вывода никаких вопросов не вызывает. Что присвоили, то и получили. И думать здесь не о чем. Вторая строка интереснее. С одной стороны, опять-таки, что присвоили, то и вывели. Но переменные наши здесь не совсем равноправны. Y объявлена в самой программе, *сверху*, так сказать. В процедуре она не объявлена. X объявлена и там и там. Z объявлена только в процедуре. А вот третья строка заставляет задуматься и, возможно, надолго.

Значение X не изменилось, хотя в процедуре оно было другим. Значение Y сохранилось. Z мы выводили и не пытались. Это и понятно, она объявлена только в процедуре.

Теперь объясняем, сначала практически, потом теоретически. Если переменная объявлена в программе, а в процедуре не объявлена, то любые её изменения – в программе ли, в процедуре ли – действуют, и навсегда. Если переменная объявлена в программе, и в процедуре объявлена тоже, то изменения сделанные в процедуре, в ней и останутся. По выходе из процедуры всё будет прощено и забыто. Если переменная не объявлена в программе, а объявлена только в процедуре – то всё становится гораздо проще - проще, в основном, для компилятора. Обратиться к этой переменной мы можем только в процедуре. Снаружи её не видно.

Теперь обещанная теория. Заранее предупреждаю, что моё толкование терминов может отличаться от приведённых в учебниках. В конце концов, я столько программирую, что имею право на личное мнение в отношении терминов. В моём коллективе я решаю, кто у меня локальная переменная!

Немного выше я сказал о переменной, что её не видно. Это не случайно. Наиважнейший термин – *область видимости*. Область видимости – откуда видно переменную. Если переменная объявлена в процедуре, то её область видимости ограничивается этой процедурой. Снаружи её не видно. Если переменная объявлена в программе, её область видимости поистине безгранична – переменную видно и в программе и во всех процедурах. Если переменная объявлена и в программе, и в процедуре, то, на самом деле, это совсем разные переменные. Ту переменную, что в процедуре, видно только из процедуры, ту переменную, что объявлена в программе, видно только в программе.

Переменные, объявленные в процедуре, называются локальными. Если переменная объявлена в программе, а мы сейчас в процедуре, то эта переменная для нас глобальная. Глобальные переменные – зло. То есть, они не запрещены, вполне могут существовать и это может даже от нас не зависеть – если программу и процедуру пишут разные люди. Это зло неизбежное. А вот зло, которого можно избежать – обращаться из процедуры к глобальной переменной. Не надо так делать.

А если процедур не одна, а две, и в каждой объявлены свои переменные? Всё так же, но область видимости переменной, объявленной в одной процедуре, этой процедурой и ограничивается, в другой процедуре эту переменную видно не будет.

Ещё одно порождение больной фантазии – процедура, вложенная в другую процедуру. То есть, имеем программу, в программе процедура, в процедуре другая процедура. Всё так же, как и было сказано выше, но роль программы выполняет первая процедура – а так всё сказанное о глобальных и локальных переменных и об области видимости остаётся в силе.

Вот так всё запутано.

## А теперь функция

Вернёмся к нашему факториалу. Помните?

```
procedure Fact(      N          : integer;
                  var result    : integer);
  var
    i          : integer;
begin
  result:=1;
  for i:=1 to N do
    result:=result * i;
end;
```

Соответственно, вызов этой процедуры и использование полученного результата будет выглядеть так:

```
Fact( N, result);
writeln( result);
```

А если пользоваться, к примеру, синусом, могли бы так:

```
x:=Sin(y);
writeln( x:8:2);
```

Или даже так:

```
writeln( Sin(y):8:2);
```

Так, разумеется, проще и нагляднее. И более естественно и похоже на реальные математические формулы. Вот таким же образом сейчас и оформим наш факториальчик.

```

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

```

Вот так это можно использовать:

```

k:=Fact(N);
writeln( k);

writeln( Fact(N));

```

Это называется функция. А теперь разглядываем нашу функцию внимательно. Принципиально важны две строчки. Первая – заголовок функции. Вместо

```

procedure Fact(      N      : integer;
  var result : integer);

```

пишем

```

function Fact(      N : integer) : integer;

```

Выходной параметр result исчез из заголовка. От него остался только тип integer, который переместился в конец заголовка функции. Зато result появился в списке объявленных внутри функции переменных. Это не обязательно, но удобно в случае, когда, как у нас, ранее написанная процедура преобразуется в функцию. И очень важная строчка

```

Fact:=result;

```

Слева – имя нашей функции. Справа, вообще говоря, может быть что угодно. Если мы напишем Fact:=5, то значением нашей функции при всех значениях аргумента будет неизменная пятёрка. Этот оператор присваивания обеспечивает возврат значения нашей функцией. Короче, без него ничего не получится. Только не промахнитесь и не напишите наоборот, result:=Fact. Транслятор проглотит, но результат выполнения будет нехороший. Это называется рекурсия. Так тоже можно, но только

если вы очень точно знаете, чего вы хотите. Впрочем, к этому мы ещё вернёмся.

Сочиним ещё какую-нибудь функцию. Для разнообразия, с результатом логического типа. Функция должна проверять, является ли положительное число точным квадратом. На вход – целое число, на выходе – логическая переменная. Проверять будем в цикле, тупым перебором всех целых чисел, меньших числа, заданного на вход. Для экономии будем прекращать проверку, как только квадрат проверяемого числа превышает заданное число. Как я замечал ранее и ещё замечу позже, оптимизировать программу по скорости – последнее дело. Впрочем, об это я ещё напишу. Оптимизировать программу стоит, только если экономия обещает быть очень значительной, у нас именно этот случай. Цикл наш должен будет завершиться в неизвестный заранее момент времени. Вообще-то, для таких случаев – приращение по одному, начиная с единицы, максимальное значение известно заранее, но цикл может прекратиться раньше – очень хорошо подходит обычный цикл **for** с использованием оператора **Break**. Но из педагогических соображений мы используем цикл **repeat-until**. И обратите внимание – нас совершенно не интересует, квадратом какого именно числа является число, поступившее на вход функции. Нас только интересует – да или нет? В результате вырисовывается такая вот функция:

```
function IsSquare(      N : integer) : boolean;
  var
    x
      : integer;
begin
  x:=0;
  repeat
    x:=x+1;
  until (x*x = N) or (x*x>N);

  if x*x = N
    then IsSquare:=true
    else IsSquare:=false;
end;
```

А почему я не написал  $x*x \geq N$ ? Для наглядности. Чтобы чётко были видны две причины нашего возможного выхода из цикла. Или нашли точный квадрат, или перебрали все числа, которые есть смысл перебирать. В качестве упражнения, оформите в виде функции написанный ранее программный код, который считал количество слов в

строке и выделял слово с заданным номером из строки. Заголовки функций должны быть примерно такими:

```
function WordCount( s : string) : integer;
function GetWord( s : string;
                 num : integer) : string;
```

И ещё, по какому-то странному недосмотру, в Турбо Паскале отсутствуют функции для выбора минимального и максимального из двух чисел. Исправьте это.

### А теперь тараканчик

Вот здесь я изобразил таракана. Как мог.

```
procedure Tarakan( x,y : integer;
                  color : integer);
begin
  SetColor(color);
  SetFillStyle( SolidFill, color);
  {тельце}
  FillEllipse( x,y, 60,30);
  {головёнка}
  Circle( x-60-20, y, 20);
  {лапки}
  Line( x,y-20, x,y-60);
  Line( x-20,y-20, x-50,y-60);
  Line( x+20,y-20, x+50,y-60);
  Line( x,y+20, x,y+60);
  Line( x-20,y+20, x-50,y+60);
  Line( x+20,y+20, x+50,y+60);
  {глазки}
  Circle( x-60-25, y-10,3);
  Circle( x-60-25, y+10,3);
end;
```

А вот здесь я оформил в виде процедуры задержку:

```
procedure Delay;
var
  i,j : integer;
begin
  for i:=1 to 1000 do
    for j:=1 to 30000 do;
end;
```

А теперь, чего я собственно хочу. А хочу я, чтобы таракан бегал. Ну, или для начала, перемещался по экрану. А как он может перемещаться?

Естественно, в цикле. Нарисовать, подождать, стереть, передвинуть и всё снова. Настолько просто, что не надо даже сначала записывать в комментариях (или, говоря правильно, в псевдокоде). Пишем сразу:

```
x:=500;
y:=200;
for i:=1 to 300 do begin
  Tarakan( x,y, Green);
  Delay;
  Tarakan( x,y, Black);
  x:=x-1;
end;
```

А теперь соберите из этого законченную программу и запустите. Должен забегать, хотя и помаргивая. На что обратить внимание? Переменная цикла *I* в процедуре и переменная *I* в программе – совсем разные переменные.

А как вы можете внести свой посильный вклад и поучаствовать в этом веселье? Хочу, чтобы таракан перебирал лапками. Ну, или как минимум, что бы каждая лапка могла быть в одной из трёх позиций. Напрашивается добавление параметра в процедуру рисования таракана – номер положения конечности. Это может быть даже переменная цикла – только к ней надо применить общепользовательский оператор **mod**. И ещё я хочу, чтобы, добежав до одной стенки (границы экрана), таракан разворачивался и бежал в обратную сторону. И, в конце концов, запустите тройку тараканов со случайными тараканскими скоростями, принимайте ставки и организуйте маленький ипподром. Надеюсь, эти просьбы не слишком сложны для вас.

В процессе перемещения таракан безобразно мерцает. Можно это победить? Можно. Надо только поступиться принципами, то есть разрешением экрана. Во времена DOS'a тоже была видеопамять. Размещалась она, правда, не на отдельной плате, а в основной памяти. И назначение её было несколько другим - она хранила образ изображения на мониторе. Тем не менее, размер её был ограничен и за её пределы - никак. Так вот, в режиме VGA (640x480), которым мы здесь везде пользуемся, в видеопамети помещается только одна страница с изображением. А если быть чуток поскромнее и спуститься до EGA (640x350), то этих страниц поместится уже две. А что это значит? Значит, первую страницу показываем зрителю, на второй тайком рисуем, затем

переключаемся на отображение второй, на первой рисуем и так далее. Практически, что для этого надо? Вместо VGA написать EGA. И прочитать про SetActivePage (выбирает, на каком экране рисовать) и про SetVisualPage (какой экран показывать). Это просто. Но, в сущности, не нужно.

### **А этот раздел просто больше некуда было вставить**

Честно говоря, единственная причина, по которой этот раздел оказался именно в этой главе – то, что результат предпочтительнее оформить в виде процедур, точнее функций. Причина весьма формальная, но пусть будет лучше здесь. Надо же где-то ему быть. Заодно в первом приближении познакомимся с понятием модуля, хотя бы на чисто эмпирическом уровне. Про эмпирический уровень – это когда что-то делаешь, оно работает, а почему работает – непонятно. О чём, собственно, речь? Или, говоря по-другому, в чём наша проблема? У вас нет проблемы? Сейчас убедим, что она у вас есть.

В игрушки играли? В некоторых старых игрушках управление мымриком было организовано даже не стрелками, а буквенной клавиатурой. Насколько помню, использовались клавиши Q,W,E,S. Кажется, так. Нажали E – мымрик побежал направо. Нажали S – вниз, или что он там делал в этом случае.

// Лирика

Кстати, как программист, преклоняюсь перед разработчиками игрушки, не знаю официального названия, исполняемый файл назывался goody.com. Запихать в шестьдесят четыре килобайта десятки экранов действия, а ведь там ещё и какой-то сюжет был, судя по-всему. Мы всем отделом за год так до конца и не добрались... Ещё раз – преклоняюсь.

Upd. Игрушка есть в американской Википедии. Так и называется Goody, платформер, разработчики испанцы, 1987. Через двадцать лет сделали римейк.

И, разумеется, о программировании игр я напишу в отдельной книге.

// конец Лирики

Однако, возвращаемся к нашей клавиатуре. Мы, конечно, можем понять, что нажата клавиша W – если после неё нажмут Enter. Но это совсем не то, что надо для игрушки. Enter нам не нужен. А без него мы пока не

умеет. Хуже того, мы даже не можем определить, что на клавиатуре что-то нажали – безотносительно того, что именно нажали. Будем учиться. Тут я нахожусь в затруднительном положении - *гусары, молчать!* С одной стороны, Турбо Паскаль предлагает соответствующие возможности, но как-то кривовато. С другой стороны, наша версия гораздо симпатичнее, но требует веры на слово. Поясню, о чём речь.

В Турбо Паскале есть две функции:

```
function KeyPressed : boolean;  
function ReadKey : char;
```

Попутно познакомимся с новым типом данных. Даже с двумя, но второй новый тип явится попозже. А пока первый новый тип – Char. Тип новый, хотя и очень простой, и даже, в каком-то смысле, уже знакомый. Char – это один символ. Строковый тип представляет собой, в некотором упрощении, массив элементов типа Char. Будем называть его (Char) символьным типом, или просто символом. То есть, если есть S, которое строка (**string**), то какое-нибудь отдельно взятое s[3] как раз и есть символ. Если объявлено ch : char;, то можно написать ch:=s[3]; или s[3]:=ch;, но, разумеется, нельзя написать ch:=s; или s:=ch;. Договоримся, что если в дальнейшем попадётся переменная по имени ch, то объявлена она как ch : char;.

Что всё это значит применительно к нашей задаче чтения с клавиатуры? Мы можем написать такой текст:

```
repeat  
  if KeyPressed  
    then ch:=ReadKey;  
  if ch = 'w'  
    then writeln('w key pressed');  
until ch = 'q';
```

Код вполне рабочий и даже почти полезный. Если нажата клавиша W мы, всего-навсего, выводим сообщение об этом факте. А могли бы отреагировать по-другому, повелев нашему игровому мымрику попрыгать.

Для любознательно-дотошных есть ещё возможность объявить что-то вроде as : **array[1..N] of char**; Потом долго думать (и проводить эксперименты) на тему совместимости этого типа с типом **string**.

Функции эти, вроде бы, делают именно то, что нам и надо. Первая проверяет, нажато ли что-то на клавиатуре. Если не нажали, ответ, как и ожидалось, отрицательный. Если нажали, то поведение функции чуть сложнее – она будет отвечать “Да” то тех пор, пока мы не прочитаем нажатую клавишу второй функцией. Тогда признак нажатия клавиши сбросится, и первая функция будет говорить “Нет”. Теперь программа чуть сложнее, но и универсальнее:

```
program Key;
uses
  Crt;
var
  ch          : char;
begin
  ClrScr;

  repeat
    if KeyPressed then begin
      ch:=ReadKey;
      writeln( Ord(ch));
    end;
  until ch = 'q';
end.
```

Программа проверяет, не нажали ли что. Если нажали, программа читает нажатый символ, а затем выводит символ на экран. Точнее, не символ, а его код. О функции Ord, как уже обещано, поговорим отдельно. Каждому символу однозначно соответствует код – число в диапазоне от нуля до 255. Выполнение программы прекращается при нажатии символа ‘q’. Это от слова Quit – в старых программах часто использовалось такое сокращение. Понаблюдайте за поведением программы при нажатии клавиш нестандартной ориентации – всяких стрелочек, функциональных клавиш и тому подобного.

Подводим итоги. В целом всё хорошо, но, как обычно, присутствуют нюансы. Нюанс мелкий – в текстах программ не случайно написано w и q, а не W и Q. Наша программа будет реагировать только на маленькие буквы. Сочетание клавиш w/Shift не будет распознано как нажатие клавиши w. Это даже нюансом назвать трудно – функция ведёт себя в точном соответствии со своим названием. Сказано прочитать символ – вот она и читает. Буква большая и та же буква, но маленькая – это ведь

разные символы. Возможно, это именно то поведение функции, которое нас лучше всего устраивает, но всё же – помнить об этом надо.

Теперь нюанс несравнимо серьёзнее. Есть клавиши хорошие – те, которые с буквами и циферками. Нажал на букву – и функция ReadKey вернула нам нажатый символ, готовый для дальнейшего использования. А есть клавиши нехорошие, но, по странному совпадению, как раз самые нужные и интересные. Те же стрелочки в первую очередь. При нажатии на такую клавишу ReadKey сообщит, что прочитана какая-то странная клавиша, вернувшая символ, кодируемый нулём. И если мы прочитали такой нулевой символ, то должны повторить чтение с помощью ReadKey ещё раз, и вот только тогда мы узнаем, что же такое было нажато. Вот это двукратное считывание кажется мне как-то не очень правильным или, по крайней мере, неудобным. Сразу возникает желание написать свою, улучшенную функцию ReadKey, или как мы её назовём.

DOS позволяет узнать нам, какой символ был нажат на клавиатуре. Но ещё DOS позволяет узнать, какая клавиша была нажата. А это, как говорят в определённых кругах, две большие разницы. Одна и та же клавиша при разных условиях может одарить нас разными одиночными символами, и наоборот – одна клавиша может сгенерировать два символа за одно нажатие – как мы видели чуть раньше. А вот с собственно клавишами всё железобетонно просто. Каждая клавиша генерирует свой скан-код. Только она и только его. Скан-код – звучит загадочно, но, на самом деле, очень просто. Когда-то, в совершенно неправдоподобно далёкие времена, кто-то незатейливо пронумеровал всю клавиатуру. Как положено – сверху вниз и слева направо. В левом верхнем углу оказался Esc, он и стал номером первым. Следующей, при тогдашней конструкции клавиатуры, оказалась клавиша с единицей. Она стала номером вторым. Ну и так далее. Вот эти номера клавиш и называются скан-кодами.

Соответственно, я предлагаю читать именно эти скан-коды, по-простому – номера. К сожалению, Турбо Паскаль не предоставляет простого способа прочитать скан-коды. Но Турбо Паскаль даёт возможность сделать почти всё, что мы хотим. Хотим читать скан-коды, значит, будем читать скан-коды. Только придётся немного потрудиться. Как всегда есть два варианта – для ленивых и для трудолюбивых. Трудолюбивые – это такие же ленивые, только они чуть-чуть поумнее ленивых и согласны

сегодня поработать немного больше, чтобы всю следующую неделю работать намного меньше.

В целом варианты весьма похожи. Первый. Добавляем в секцию **uses** имя Dos. Затем включаем в программу описание двух функций:

```
{-----}
function OurKeyPressed : boolean;
  var
    R          : Registers;
begin
  R.ah:=1;
  Intr( $16, R);
  if (R.flags and fZero) <> 0
    then OurKeyPressed:=false
    else OurKeyPressed:=true;
end;
{-----}
function OurReadKey : byte;
  var
    R          : Registers;
    ch         : char;
    sc         : byte;
begin
  R.ah:=0;
  Intr( $16, R);
  ch:=Chr(R.al);  sc:=R.ah;
  OurReadKey:=sc;
end;
{-----}
```

Что мы получили и как этим пользоваться? Первая функция заменяет стандартную турбопаскалевскую функцию KeyPressed. Вторая функция заменяет, соответственно, стандартный ReadKey, с поправкой на то, что наш OurReadKey читает не символы, а скан-коды.

И попутно - второй новый тип – byte. Он ещё проще, чем char. Это такая половинка от integer, точнее от word. Занимает он не два байта, а один, и вмещается в него диапазон целых чисел от 0 до 255. В большинстве случаев - и в нашем - вместо byte можно использовать обычный integer. То есть, можно писать k:=OurReadKey, где K объявлено как integer. Так зачем мы используем byte? Зачем, зачем, ну, получается что исключительно из принципа. На самом деле скан-код – это всегда один байт, вот мы и используем переменную, которая ровно один байт занимает. Если Вы нежадный, можете в предыдущем коде заменить sc :

byte; на sc : integer;. Главное, запомните общий принцип – переменной типа integer можно сколько угодно присваивать переменные типа byte, обратное весьма нежелательно. Как обычно, думать над этим три минуты.

Как этим пользоваться? Следующий код проверяет, нажата ли клавиша. Если какая-то клавиша нажата, выводим её скан-код. Если нажали Esc, прекращаем работу. Sc объявлено как byte или как integer, что вам больше нравится.

```
repeat
  if OurKeyPressed
    then sc:=OurReadKey;
    writeln('scan = ', sc);
until sc = 1;
```

Теперь немного о смысле того, а что мы написали. Всё это можно было написать на ассемблере – Турбо Паскаль разрешает ассемблерные вставки. Такой псевдопаскалевский/псевдоассемблерный стиль – следствие моих личных предпочтений. В смысле, мне так больше нравится.

Тип Registers – это запись (смотри далее) обеспечивающая доступ к регистрам процессора. Поля записи – отдельные регистры. Разумеется, обращение к записи не означает непосредственного обращения к регистрам процессора. Обращение происходит при вызове процедуры Intr. Она вызывает так называемое 21-е прерывание DOS, которое на самом деле никакое не прерывание. Первый параметр – номер функции прерывания, второй параметр – состояние регистров. Если регистры меняются во время вызова прерывания, изменённые состояния мы получим в этом же параметре. Подробнее об этом можно прочитать в *П.Нортон, Р.Уилтон “IBM PC и PS/2 Руководство по программированию”*. Питера Нортона программисты старой школы по традиции уважают, но ещё лучше прочитать об этом в... В чём? Как обычно, книжку из шкафа только что украли, а Интернет расходуется во мнениях. И как его только не называли... Жорден, Жордэн, Джурден, Жордейн, Джордейн. А потом оказалось, что это вовсе и не он, а такая специальная очень умная тётенька, и зовут его вовсе и не Роберт, а Роберта. Название приблизительно – *Справочник программиста PC XT и AT*. Когда я был молодым, а всё вокруг зелёное и цвело и кустилось...

Короче, в те времена ни один серьёзный программист не мог обойтись без этой книги.

Теперь вариант для трудолюбивых. Для тех, кому не хочется вставлять этот код в каждую свеженеписанную программу. Попутно знакомимся с новым понятием – модуль. Пока чисто прикладным образом, теория – в следующей главе.

Итак – <F10>, <File\New>. Появляется новое, абсолютно пустое окно редактирования без заголовка. Точнее, с заголовком понапе00.pas, что, в целом, ничем не лучше, чем без заголовка вообще. Набираем в этом пустом окне вот такой текст:

```
unit Scan;
{-----}
interface
{----- Scan Codes -----}
  const
    ArrowLeft = 75;    ArrowRight = 77;
    ArrowDown = 80;   ArrowUp    = 72;   Esc    = 1;
{-----}
function OurKeyPressed : boolean;
function OurReadKey : byte;
{-----}
implementation
  uses
    Dos;
{-----}
function OurKeyPressed : boolean;
  var
    R      : Registers;
begin
  R.ah:=1;
  Intr( $16, R);
  if (R.flags and fZero) <> 0
  then OurKeyPressed:=false
  else OurKeyPressed:=true;
end;
{-----}
function OurReadKey : byte;
  var
    R      : Registers;
    ch     : char;
    sc     : byte;
begin
  R.ah:=0;
  Intr( $16, R);
  ch:=Chr(R.al);  sc:=R.ah;
```

```

        OurReadKey:=sc;
end;
{-----}
end.

```

Всё почти так же, как и в ранее набранных функциях, включенных в тест нашей программы. Только добавились волшебные слова **unit**, **interface**, **implementation**. С волшебными словами подробно разберёмся в следующей главе. Ещё у нас появилась секция **const**, а в ней объявления нескольких констант. Всё это в целом называется модуль. Теперь нажимаем F2 и сохраняем наш модуль под именем Scan.pas. Обратите внимание, тот же подход, что и с программой. Имя модуля (то, что после **unit**), совпадает с именем файла, в котором этот модуль сохранён.

И как всем этим богатством воспользоваться? В разделе **uses** основной программы кроме Graph и Crt (если они нам нужны) дописываем наш Scan. Теперь, как всегда - нюанс. А могли бы мы назвать функцию не OurReadKey, а просто ReadKey, как и стандартную? Могли бы. Но в этом случае мы были бы должны аккуратно вписать его в *uses* *перед* Crt. Причина? А подумать? Понимаю, что сложно, но гипотезу высказать можно? В Crt уже есть и KeyPressed и ReadKey, и, вписав наш модуль раньше, мы занимаем для себя эти имена. Кто первый встал, того и тапки. Я понимаю, что это кажется очень простым. А в жизни пишут после Crt, и долго чешут репу. И зачем нам лишние хлопоты? Проще переименовать. Описания функций из основной программы выкидываем. Получается так:

```

uses
    Scan, Graph, Crt;
repeat
    if OurKeyPressed
        then sc:=OurReadKey;
        writeln('scan = ', sc);
until sc = Esc;

```

И что мы от этого имеем? Польза основная – вместо добавления текстов процедур каждый раз в текст основной программы, достаточно добавить одно имя в секцию **uses**. И не забыть при этом, что файл с текстом модуля Scan должен лежать в том же каталоге, что и сама программа.

На самом деле, это не совсем так строго. Модуль может лежать и в другом каталоге, но имя этого каталога должно быть указано в <Options\Directories\Unit directories>. И, обратите внимание, Dos в секции

uses основной программы уже не обязателен. На него ссылается непосредственно наш модуль Scan.

Польза вспомогательная – нам не надо помнить, что скан-код Esc равен единице. Оно у нас теперь константа. Там же, в разделе констант, определены и скан-коды стрелок на все четыре стороны. В Дополнении имеется полный текст модуля Scan. Можете вытаскивать оттуда скан-коды нужных клавиш по мере надобности, или набрать весь модуль сразу.

Кстати, что характерно, для применения в основной программе доступны только имена, описанные в модуле в секции **interface**. Это неспроста.

### Всем стоять и не разбежаться!

*Написав предыдущий раздел, меня замучила совесть © Антон Палыч Чехов, почти.*

А может, зря это я? Может, не надо скан-кодов? Может, надо быть проще и ближе к народу? Изложим предыдущее применительно к стандартным средствам Турбо Паскаля. То есть, как можно работать с клавиатурой, ничего не зная о скан-кодах и прочих ужасах, только с помощью KeyPressed и ReadKey. Напоминаю, там всё было бы хорошо, если бы не тот прискорбный факт, что в случае нажатия какой-нибудь непростой клавиши мы получаем два как бы нажатых символа - первый ноль, а второй означающий, что, собственно, мы нажали. Попробуем с помощью такой технологии различить нажатия на клавиши со стрелочками и пробел. Первые четыре – клавиши, порождающие два символа, пробел – клавиша честная, простая и незатейливая. В смысле, символ только один. В программе нашей мы просто будем в цикле выводить название нажатой клавиши. Завершится это незатейливое развлечением нажатием “q”.

```
repeat
  if keyPressed then begin
    ch:=ReadKey;
    if Ord(ch) = 0 then begin
      ch:=ReadKey;
      if Ord(ch) = 75 then writeln('Arrow left');
      if Ord(ch) = 77 then writeln('Arrow right');
      if Ord(ch) = 80 then writeln('Arrow down');
      if Ord(ch) = 72 then writeln('Arrow up');
    end
    else begin
      if Ord(ch) = 32 then writeln('SpaceBar');
```

```

        end;
    end;
until ch='q';

```

Обратите внимание, без функции Ord обойтись всё-таки никак не удалось. И ещё, если нажали что-нибудь отличное от заранее предусмотренных пяти клавиш, программа наша будет молчать, как рыба об лёд. Добавьте, для этого случая, вывод кода нажатой клавиши. И, что важнее, подумайте об оформлении всего этого в виде отдельного модуля, пусть он будет в минимальном варианте содержать одни константы кодов клавиш. Лично я этих кодов не помню и, не то, что не хочу, но даже и помнить не могу. И вас от этого категорически отговариваю.

### Применим к тараканчику

Вспомним тараканчика. Рисовался он процедурой с вот таким заголовком:

```

procedure Tarakan(    x,y    : integer;
                      color : integer);

```

А бегал посредством вот такой программы:

```

x:=500;
y:=200;
for i:=1 to 300 do begin
    Tarakan( x,y, Green);
    Delay;
    Tarakan( x,y, Black);
    x:=x-1;
end;

```

Хотелось бы, чтобы зверушка наша не скакала тупо и упрямо всегда влево, как здесь, а управлялась с клавиатуры. Стрелка влево – бежит налево, стрелка вправо – направо... Не будем двигаться по шагам, а приведём результат сразу. Он нам ещё много раз пригодится.

```

x:=500;
y:=200;
repeat
    if OurKeyPressed then begin
        sc:=OurReadKey;
        oldX:=x;
        oldY:=y;
        if sc = ArrowLeft then begin

```

```

        x:=x-1;
    end;
    if sc = ArrowRight then begin
        x:=x+1;
    end;
    if sc = ArrowUp then begin
        y:=y-1;
    end;
    if sc = ArrowDown then begin
        y:=y+1;
    end;
    Tarakan( oldX, oldY, Black);
    Tarakan( x, y, Green);
end;
until sc = Esc;

```

Обратите внимание, Delay здесь нам не нужен. Таракан теперь не в режиме автопилота, а на ручном управлении. Поразмышляйте о чувствах таракана,двигающегося кормой вперёд. Ну и, конечно, я хочу чтобы таракан головёнкой о края экрана не стучался. Я в курсе, что он не стучится, а молча уползает дальше, в голубую даль. А вы в курсе, что я в курсе. Сделайте же что-нибудь, жалко ведь зверушку...

## Глава 9 Совсем настоящая программа

### Про что программа?

Конечно, игрушка. Остальное просто недостойно нашего внимания. Не электронную же таблицу нам выстрегивать.

Тем не менее, оценим наши возможности. Morrowind, Civilization и НОММ пока программировать не будем. Morrowind – наполовину математически тосклив, наполовину просто тосклив. Тосклив для программиста, который пишет движок – но мы же тут программисты, и движок придётся писать именно нам. И тоскливо будет нам – а весело совсем другим категориям наёмных работников. И для 3D нужно очень-очень много математики.

Civilization (первый) и НОММ вполне реальны для начинающего программиста. Реальны при условии, что он усвоит ещё кое-что из технических подробностей, что мы ещё не усвоили, и, главное, будет знать, как правильно подойти к делу. Последнее, повторюсь, главное. Технические детали позволительно не знать - я не всё знаю, функции Windows API не то, что можно, нужно не знать - я, понятное дело, и не знаю и не хочу. Точнее, знаю, но со справочником. Это то же самое как знать английский язык со словарём.

Главное, освоить правильный подход. Как сказал, не знаю кто, *главное в работе руководителя – подбор исполнителей и проверка исполнения*. Детали и подробности оставьте подчиненным – не царское это дело. И всё стало хорошо, только исполнители почему-то злятся. Для вас разница в том, что пока исполнителями являются написанные Вами процедуры и функции, а проверкой исполнения - ваша же уверенность в том, что они выдают правильный результат. И будет Вам ЩЩастие. Почти.

Однако, к делу. Раз ничего из вышеперечисленного (пока) программировать мы не будем, что, собственно, нам остаётся? Пятнашки, Ханойские башни, Ним, Морской бой, Поле чудес, всяческие игры со словами. Теперь отстреливаем лишнее.

Поле чудес – без файлов никак.

Пятнашки, Ханойские башни – слишком просто, нет противника.

Ним – противник есть, но игра за него элементарна.  
Морской бой – оставим для Delphi, если получится.  
Игры со словами – пока просто не хочу, как-то уныло.

Предлагаю крестики-нолики. Присутствуют, какая-никакая графика, какое-никакое управление со стороны игрока и какой-никакой искусственный интеллект противника. А графическое решение открывает простор для необузданной фантазии.

### Отладка. Давно пора

Программы, которые мы до сих пор писали, были простыми. Или, выражаясь мягче, не очень сложными. Если что-то сочинялось не так, по нашему ли неразумению, или по простой опiske-опечатке, выловить ошибку удавалось упорным (тупым) изучением (разглядыванием) текста программы. И ошибка быстро обнаруживалась – я в этом уверен, честное слово!

Эта программа будет посложнее. И ошибки будут посложнее. И справиться с этими ошибками будет куда сложнее. Некоторые гиблые места мне известны заранее – соответствие строк и столбцов поля, координат расположения курсора и элементов двумерного массива, описывающего текущее состояние игры. Ошибаются все, проверено.

Поэтому для отлова и отстрела ошибок придётся задействовать тяжёлую артиллерию – отладочный режим. Тренироваться будем, как всегда, на кошках. В роли кошки – вот такая, совершенно бессмысленная программка. Циферки слева, ясное дело, к тексту программы не относятся, это номера строк. Они здесь, в книге, не в Паскале, для удобства ссылки на них. Набирайте текст:

```
01 program debug;
02   var
03     x,y           : integer;
04     i             : integer;
05 {-----}
06 procedure Incr( var x : integer);
07 begin
08   x:=x + 1;
09 end;
10 {-----}
11 begin
12   x:=1;
```

```

13     write('y = ');
14     readln(y);
15     Incr(x);
16     for i:=1 to 1000 do begin
17         if y > 0 then begin
18             x:=x + 1;
19         end
20         else begin
21             x:= x - 1;
22         end;
23     end;
24 end.

```

Как всегда нажимаем F9. А дальше не как всегда. Вместо Ctrl/F9 нажимаем F7. Образовался зелёный курсорщик во всю ширину экрана и установился на строке 11, то есть на первом **begin**'е. Снова нажимаем F7. Курсор переместился на следующую строку. Мы находимся в режиме пошагового выполнения программы, он же отладочный режим. Зелёный курсор показывает строку, которая сейчас будет выполняться. Внимание: эта строка ещё не выполнена, она будет выполнена только при следующем нажатии F7. Одно нажатие F7 – одна выполненная строка. И снова внимание – выполняется одна строка, а не один оператор. То есть, если в строке несколько операторов, выполнены будут все сразу, что не есть хорошо. Так что не жалейте заварку, в смысле пишите по одному оператору на строке.

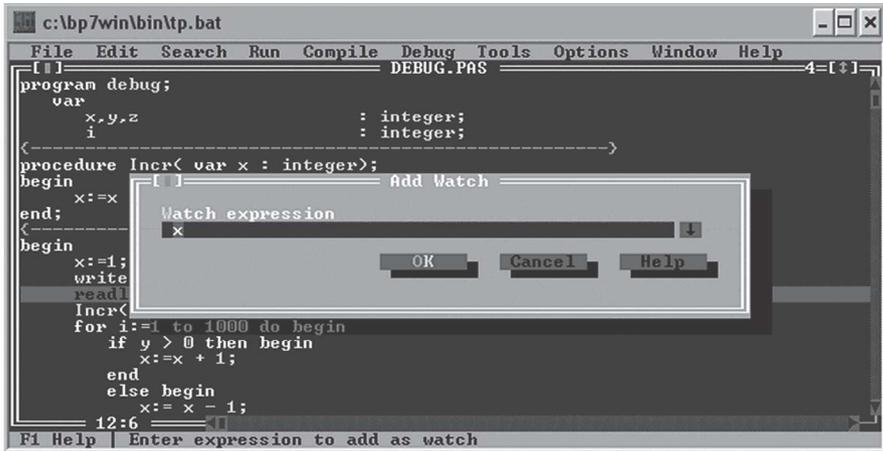
Добрались до строки 14. Нажали F7. Оказались на чёрном экране с приглашением ввести Y. В принципе, это естественно, никто за нас Y вводить не будет. А теперь, остановившись на строке 15, опять жмём F7. И попадаем в процедуру Incr. Затем тем же способом, через F7, из неё выбираемся. Вроде бы всё понятно и разъяснений не требует. Нажимаем Ctrl/F2. Зелёный курсор пропал – мы вышли из режима отладки.

Начинаем всё сначала, только вместо F7 будем нажимать F8. Всё происходит абсолютно так же, пока мы не доберёмся до строки 15. Теперь по нажатию F8 мы не входим в процедуру Incr, а перемещаемся на следующую строку 16. Смысл клавиши F8 должен быть вам понятен – такое же пошаговое выполнение, но без захода в процедуры и функции. Теперь главный вопрос – а зачем, собственно? Зачем это надо, и какие блага мы, собственно, от этого получаем? Некоторая польза обнаруживается на строке 17. В зависимости от введённого значения Y, мы видим, по какой из веток условного оператора движется наша

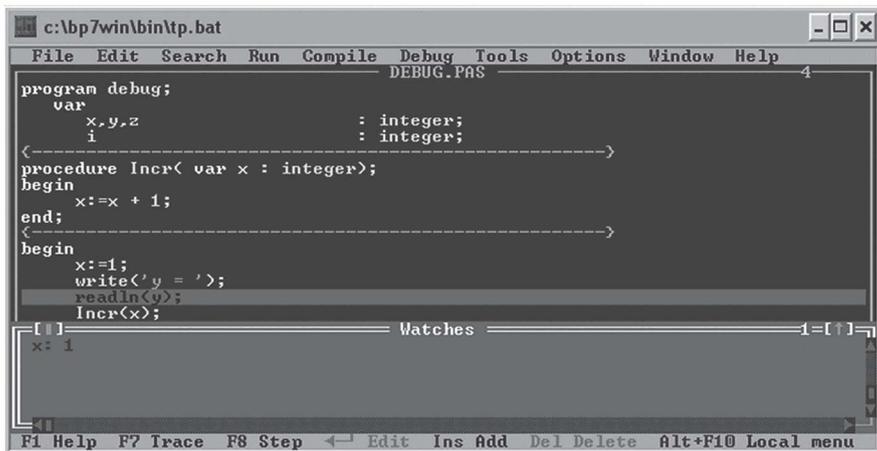
программа. Это уже очень полезно. В нашем случае всё и так очевидно, но ведь условие может быть гораздо сложнее и, что важнее, Y мог быть не введён нами только что, а рассчитан где-то давным-давно. А теперь желание посложнее, но более пользу приносящее. Хотелось бы узнать, чему равно значение переменной, и как оно меняется в процессе выполнения программы. Для этого придётся произвести ряд утомительных телодвижений. В связи с неочевидностью процесса, будем его иллюстрировать.

Начинаем всё сначала, отслеживать выполнение программы в пошаговом режиме. То есть, нажимаем F7, потом ещё и ещё, пока наш большой зелёный курсор не окажется на строке 14. Теперь приступаем к действиям. Пусть, к примеру, нас интересует судьба переменной X. Подкрадываемся курсором и устанавливаем его точно на переменной. Мы можем выбрать любое вхождение переменной X в программе, это неважно. Нажимаем Ctrl/F7.

Появляется вот такое окно:

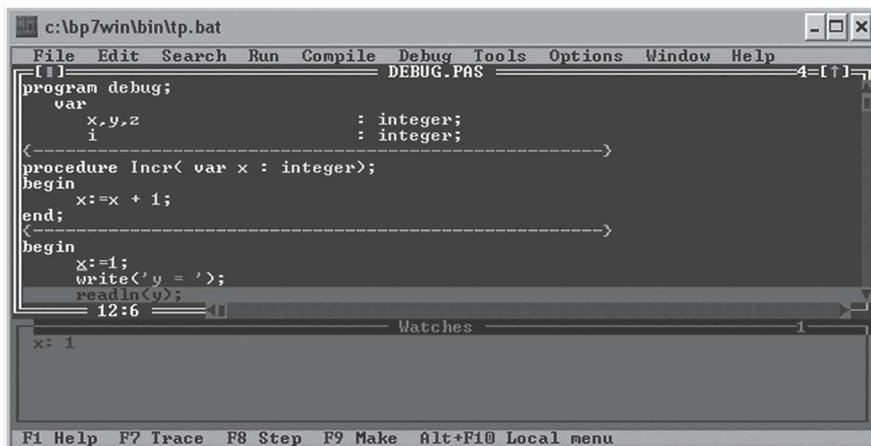


Вместо X мы можем ввести имя другой переменной, но, скорее всего, X это именно то, что нам и надо. Нажимаем Enter и получаем вот такое:



Мы видим значение переменной. Это хорошо. Но активным является окно со значением переменной. Это, всё-таки, плохо. Нам бы хотелось продолжать движение по программе, а для этого надо вернуться в окно с текстом программы. Нажимаем F6. Эта клавиша, если помните, поочерёдно перебирает все открытые окна. Даже картинку приводить не буду. Осталось только окно редактирования. Окно с переменными тоже осталось, конечно, но где-то сзади и нам недоступное и невидимое. Опять плохо.

Теперь выберите в меню <F10>, <Window\Cascade>. Или Alt/W, А. Все окошки выстроились и мы получили вот такой образцовый порядок:



А что ещё есть в запасе полезного? Нажав **Ctrl/F4** можно не только посмотреть значение переменной, но и его, значение, изменить. Мучительно пытаюсь припомнить, когда мне это понадобилось, и не могу.

А теперь, предмет гораздо более существенный, чем изменение значений переменных. Встали на строку 21 и нажали **Ctrl/F8**. Строка выделилась жирным красным цветом. Что это значит? Как это называется? Называется это – точка останова. Не остановки, а остановка! А кто говорил *остановки*, тех мои первые учителя программирования хватали за волосы и возили мордой по столу, приговаривая *А знаешь, как Ваньку Жукова учили?!*. Если попадался гуманный учитель, то просто гуманно бил линейкой по рукам. Извините, расчувствовался. Точка останова нужна, когда мы точно знаем, где хотим оказаться – в данном случае на строке 21, но при этом не хотим шлёпать туда пошагово, долбя по клавише **F7**. Запускаем программу на выполнение как обычно – через **Ctrl/F9**. Вводим отрицательное значение *Y*. Попадаем на точку останова – зелёный курсор на строке 21. Дальше можно двигаться в пошаговом режиме. Отменить точку останова можно, остановившись курсором на ней и снова нажав **Ctrl/F8**.

И помните – когда вам всё это надоест – просто нажмите **Ctrl/F2**. И ещё. Если программа работает в графическом режиме, то при переходе из окна редактирования в экран с графикой и обратно, графический экран будет выглядеть как-то так... Нехорошо... Грязновато... Это нормально и совершенно не страшно.

### **Ещё одна очень важная вещь. Модули**

Модуль – по-английски **unit**. Осваивать будем по той же схеме – сначала модуль, ничего не содержащий. Затем модуль хоть что-то делающий и как его применить. А затем я постараюсь объяснить, а Вы постараетесь понять, зачем они, модули, нужны. Модуль, в отличие, от, например, процедуры, вещь реальная. Модуль можно потрогать руками. Один модуль – один файл.

А теперь – абсолютно пустой модуль. Что, разумеется, не означает абсолютно пустой файл. <F10>, <File\New> - получаем новое окно редактирования. Набираем:

```
unit SomeUnit;  
interface  
implementation  
end.
```

Нажимаем F2 и сохраняем модуль под именем SomeUnit.pas. Имя модуля, разумеется, должно быть уникальным, как и все имена в программе. Модуль можно оттранслировать, как обычно нажав F9. Ошибок быть не должно. Запустить на выполнение, правда, не удастся. Модуль, подобно процедуре, сам по себе ничего не делает. Процедуру надо вызвать. Модуль надо подключить. То есть, главная ничего не делающая программа должна выглядеть вот так:

```
program Nothing;  
  uses  
    SomeUnit;  
begin  
end.
```

Обращаем внимание на определённое сходство структуры модуля и структуры программы. Сходство, как увидим в дальнейшем, ещё значительнее – модуль, как и программа, может иметь свои собственные секции **uses**, **const**, **type**, **var**. Теперь добавляем нашему модулю хотя бы минимальную функциональность. На самом деле даже не совсем минимальную – постараемся продемонстрировать весь, что называется, спектр возможностей.

```
unit SomeUnit;  
interface  
  uses  
    Dos;  
  const  
    Esc = 1;  
  type  
    TColor = integer;  
  var  
    x,y           : integer;  
  procedure DoSomething;  
{-----}  
implementation  
  uses
```

```

    Crt;
    const
      e = 2.718;
    var
      z          : integer;
{ -----}
procedure DoNothing;
begin
end;
{ -----}
procedure DoSomething;
begin
    z:=0;
    DoNothing;
end;
{ -----}
end.

```

Программа, использующая наш модуль – всё та же – Nothing. На наш модуль мы сослались в секции **uses**. В программе можно использовать константу **Esc**. Это хорошо. В программе можно объявить переменную типа **TColor**. Это тоже хорошо. В программе можно вызвать процедуру **DoSomething**. Это даже не то, что хорошо – это желательно и обязательно. Как правило, модули как раз и содержат различные процедуры и функции. По крайней мере, все модули Турбо Паскаля именно таковы.

В программе мы можем обратиться к переменным **X** и **Y**. Это плохо. Это те же глобальные переменные, только хуже. Когда-то это было распространённой практикой - создавался специальный модуль, где объявлялись все глобальные переменные и только они. На этот модуль ссылались все остальные модули и головная программа. Это я вам подробно объяснил, как не надо себя вести. Раньше это было неизбежным злом. С появлением объектов (Турбо Паскаль версии 5.5) такое поведение стало аморальным и местами даже преступным. Об объектах мы здесь говорить не будем, но, тем не менее, к вам это тоже относится. Впрочем, об объектах, если повезёт, ещё напишем.

Это всё было о том, что можно. Теперь о том, что нельзя. Нельзя из программы обратиться к переменной **Z** и константе **E**. Нельзя из программы обратиться к процедуре **DoNothing**. К кому обращаться можно, а к кому нельзя, определяют волшебные слова **interface** и **implementation**. Технически это секции. Секция интерфейса - кто бы подумал - и секция реализации.

Всё, что от слова **interface** до слова **implementation** видно снаружи модуля. Любая программа или модуль, указавшая `SomeUnit` в секции **uses**, может получить доступ ко всему нашему бараклу, упомянутому здесь, в секции интерфейса. Всё, что между **implementation** и концом модуля, видно только из секции **implementation** этого модуля.

И ещё. Если в программе есть ссылка на `SomeUnit`, это вовсе не значит, что в программе есть ссылка на модуль `Dos`. Её, ссылки, там нет. И это несмотря на то, что ссылка есть в секции интерфейса нашего модуля. О ссылке на `Crt` в секции реализации даже не говорю. Как там нас на истории средних веков учили – вассал моего вассала не мой вассал. Так и здесь. Хотите `Dos` – ссылайтесь явно.

А теперь о главном – зачем это надо?

Первый ответ банальный – чтобы программа не была слишком большой. Чтобы можно было попилить программу на части. А когда модуль станет слишком большим – его тоже попилить. Это не пустяк. Это важно. Поверьте. А каков максимальный размер программы или модуля? Лично мне кажется, что тысяча строк. В крайнем случае, две тысячи, если никак не пилится на части. Две тысячи – это для моего старого, измученного жизнью программистского интеллекта. Для вас и пятисот хватит. Попутно. А каков максимально рекомендуемый размер процедуры? Есть такое мнение, что процедура должна уместиться на одной странице. Страницы, конечно, разные бывают, но в целом я согласен. подход правильный.

А ещё зачем? В смысле, зачем надо? Говоря по-умному, для сокрытия реализации. Ещё раз – модуль состоит из двух секций - секция интерфейса (**interface**) и секция реализации (**implementation**). В секции интерфейса – только объявления и заголовки. В секции реализации – как оно на самом деле внутри устроено. Из секции реализации наружу дороги нет, как из чёрной дыры. Всё что там объявлено – константы, переменные, типы – всё там и останется. И это правильно. Программист слаб. Не может он думать обо всём сразу. А здесь мы ему жизнь облегчаем. Когда он пишет модуль – думать надо о секции реализации. Когда он модуль использует – думать только о секции интерфейса.

До совершенства эта концепция дошла с появлением объектно-ориентированного программирования.

Некоторые потаённые технические детали. Если в uses перечислено несколько модулей, а в их секциях интерфейса присутствуют одинаковые имена, проблема решаема. Перед именем надо через точку указать имя модуля - Graph.Green. Если модуль А ссылается на модуль В, а модуль В ссылается на модуль А, то так нельзя и вообще, это страшное преступление, именуемое circular reference. Однако, модуль А может ссылаться на секцию интерфейса модуля В, а модуль В может ссылаться на секцию реализации модуля А. Ничего не поняли? Учиться, учиться и учиться.

Практический совет. Есть программа, есть модуль. Редактируете программу, хотите запустить - нет проблем. Редактируете модуль, хотите запустить, или даже оттранслировать всю программу сразу - никак. Неудобно. Лечится просто - <F10>, <Compile\Primary File>. Выбираем имя нашей программы. А чтобы не на один раз - идём <Options\Save>. Только убедитесь, что сохранено будет в текущем каталоге, в том, который содержит наш проект.

### С чего начать?

С чего начать нашу программу? Про искусственный интеллект пока задумываться не будем – на этом этапе и никакого своего ещё не хватит. Ещё раз – начинать всегда надо с того, что очевидно. Читал я в детстве в какой-то книжке по физике, в целом замечательной, притчу о том, как папа организовал детишек на перетаскивание кучи камней. Сначала он заставил их перетаскать самые тяжёлые, а уже потом мелочёвку. И возблагодарили они папу и жили долго и счастливо. Такие программисты счастливо не живут. И даже долго не живут. Начинать надо с мелочёвки. С неё и начнём.

С чего начинается игра? С того, что кто-то чертит игровое поле – три на три. Значит, и мы с этого начнем.

Пошли дальше. Имеем цикл. До тех пор, пока игра не закончится. Не забыть, что игра может закончиться не только победой игрока или компьютера, но и ничьей. После цикла поздравления и фейерверки или наоборот. Внутри цикла: ходит игрок крестиком. Куда ходить, выбирает

курсором, которым управляет стрелками на клавиатуре. Если этим ходом игрок не выиграл, в ответ компьютер ходит ноликом.

Это наша программа в динамике. Теперь посмотрим в статике, какие составляющие-процедуры нам будут нужны. Нарисовать поле – процедура без параметров. Нарисовать крестик, нарисовать нолик – процедуры с параметрами, параметры – где рисовать. Нарисовать курсор – параметры, в данном случае – где рисовать. А поскольку курсор должен уметь бегать по экрану, ещё надо уметь его стирать, то есть рисовать цветом фона. Если без лишних усилий – чёрным. Для этого добавить в качестве параметра цвет. Или написать две процедуры – для рисования и для стирания.

А о каких параметрах мы сейчас говорили? Что значит – где рисовать? Понятно, что параметров будет два – нам надо задать расположение по горизонтали и по вертикали. Но в каких единицах? Говорю сразу – на экранные точки (пиксели) я категорически не согласен. Даже аргументировать не буду. Ну не согласен я, и всё тут. Координаты – это строка и столбец нашего расчерченного поля. Три строки на три столбца.

Процедура для проверки завершения игры. На вход – игровое поле, на выходе результат – победа, поражение, ничья. Ну и что-то там для торжественного финала. Также и пресловутый искусственный интеллект требует своего места под солнцем. Процедуру, в смысле, искусственный интеллект требует для проживания. Кстати, в процессе финала, неплохо бы нарисовать линию, перечёркивающую три крестика (или три нолика), которой обычно и завершается игра. Это не совсем тривиально.

Теперь подумаем о наших данных. Чтобы помнить, где что – массив три на три. Если крестик – в соответствующий элемент пишем единицу. Мысль кодировать нолик ноликом, отвергаем как идиотскую. Ноликом может кодироваться только отсутствие чего-либо, то есть, в нашем случае пустое поле. Нолик будет кодироваться двойкой. Ещё две целых переменных – запомнить, где у нас курсор. Ну и всякая вспомогательная мелочь.

Где-то позади переменных уныло тащатся константы. На этапе проектирования о константах думают не всегда, но мы хорошие, мы думаем. Вспоминаем про метод опорной точки – значит нам нужны X0

и Y0 – для левого верхнего угла. Думаем ещё. У клетки игрового поля есть размер. Размер, он только один – клетки-то квадратные. Крестики и нолики должны помещаться в клетках. Курсор по размерам тоже должен соответствовать. Операция проведения победной линии также не сможет обойтись без знания размера квадрата. Уговорили! Вводим константу S – размер стороны квадрата. Вроде бы, наша геометрия полностью определена.

Начинаем. Пишем пустые процедуры и их вызовы, там, где это очевидно. Содержимое добавим потом.

Теперь глубокая мысль, причём, что нечасто, полностью моя мысль. При прочих равных, следует стремиться к тому, чтобы программа была безошибочно транслируемой в любой момент. То есть, написав вызов процедуры, тут же бежим наверх и пишем её тельце, хотя бы и пустое. О том, что, написав **begin**, тут же обязательно надо написать **end**, и говорить нечего. На практике это значит, что в любой момент можно нажать F9 и получить сообщение об отсутствии ошибок. Понятно, что не в совершенно любой, это практически невозможно. Но пока ваша программа неспособна безошибочно оттранслироваться, идти пить чай вы не имеете права. И даже откинуться на спинку кресла права не имеете. И важное отсюда следствие – транслируйте чаще!

И ещё - стремитесь к тому, чтобы программа в каждый момент её написания была правильной. Даже изготавливая первый набросок программы, старайтесь, чтобы всё, вами написанное так и осталось неизменным. Заранее рассчитывайте, что добавлять вам очень много чего придётся, а вот убирать что-то – в идеале нет. Это, конечно, очень-очень в идеале. В реальности придётся и удалять и исправлять уже написанное и много-много раз. Но стремиться к идеалу надо. Теперь вперёд!

```
program Tic;  
{я в курсе, что по-английски это будет tic-tac-toe,}  
{но настаиваю на том, что имя программы и имя файла}  
{должны совпадать. Значит в имени не более восьми символов}  
uses  
  Graph, Scan;  
const  
  {мне кажется неправильным объявлять константу N = 3}  
  {если что-то надо сделать три раза,}  
  {то можно обойтись без циклов.}  
  {а с какого момента константы необходимы?}
```

```

        {вопрос философский...навверное, с четырёх. Кроме шуток}
        X0 = 200;
        Y0 = 100;
        S   = 100;    {циферки, что называется, от балды. Уточним
потом}
type
    TArray3x3 = array[1..3,1..3] of integer; {этот тип
непроста}
var
    a           : TArray3x3;    {самый главный массив}
    x,y         : integer;      {координаты курсора}
    sc          : byte;         {это для клавиатуры}
    result      : integer;      {не закончилась ли игра и чем}
    driver, mode : integer;      {понятно, для чего}
    gamover     : boolean;      {gamover}
    xkuda, ykuda : integer;      {а это куда выберет ходить AI}
    i,j,k       : integer;      {запас для циклов}
{-----}
{эта процедура рисует крестик}
procedure Cr(      x,y : integer);
{параметры - номера столбца и строки!}
begin
end;
{-----}
{эта процедура рисует нолик}
procedure Zero(   x,y : integer); {параметры - номера столбца
и строки}
begin
end;
{-----}
{эта процедура рисует курсор}
procedure DrawCursor(      x,y : integer);
{параметры - номера столбца и строки}
begin
end;
{-----}
{эта процедура стирает курсор}
procedure HideCursor(      x,y : integer);
{параметры - номера столбца и строки}
begin
end;
{-----}
{эта процедура рисует поле}
procedure DrawField;
begin
end;
{-----}
{эта процедура проверяет завершение игры}
{результат: крестики выиграли - 1}
{нолики выиграли - 2}
{ничья - 3}
{игра не окончена - 0}
procedure Check(      a           : TArray3x3;

```

```

                                var result : integer);
begin
end;
{-----}
{чего-то там делает}
procedure Finale(      result : integer);
begin
end;
{-----}
{а это великий и ужасный Искусственный Интеллект}
{на выходе - столбец и строка, куда ставить нолик}
procedure AI(      a      : TArray3x3;
                var xkuda, ykuda : integer);

begin
end;
{-----}
begin
    driver:=VGA;
    mode:=VGAHi;
    InitGraph( driver, mode, '');

    {тут всё самое главное}

    CloseGraph;
end.

```

Вот такой симпатичный скелетик нашей программы.

## Поле

Рисуем поле. Три на три. Не просто, а очень просто. Это у нас процедура без параметров.

```

procedure DrawField;
begin
    SetColor(Green);
    SetLineStyle( SolidLn, 0, ThickWidth);
    {горизонтали}
    Line( x0, y0+s,      x0+3*s, y0+s);
    Line( x0, y0+2*s, x0+3*s, y0+2*s);
    {вертикали}
    Line( x0+s, y0,      x0+s, y0+3*s);
    Line( x0+2*s, y0, x0+2*s, y0+3*s);
end;

```

Немного минималистки. Дорисуйте, чего не хватает.

## Крестик и нолик

Здесь чуть сложнее – процедуры с параметром. Начнём с нолика. Что использовать? Без вариантов – Circle. А где будет его центр? В центре клетки, понятно.

```
procedure Zero(    x,y : integer);  
  {параметры – столбец, строка}  
begin  
  SetColor(Red);  
  SetLineStyle( SolidLn, 0, ThickWidth);  
  Circle( x0+(x-1)*s+(s div 2),  
         y0+(y-1)*s+(s div 2),  
         (s div 2) -10);  
end;
```

Пояснения нужны? Тогда поясняем. Пояснения для координаты X. Для Y всё то же самое, что естественно.

$x0+(x-1)*s+(s \text{ div } 2)$ . X0 – это самое начало. Его прибавлять надо всегда, это не обсуждается. Если мы в третьей, например, клетке, нам надо прибавить размеры двух предыдущих и половину размера от текущей клетки, чтобы попасть ровно в центр третьей.

$(s \text{ div } 2) -10)$ . Зачем нужна десятка? Чтобы было красиво.

Крестик будет чуть посложнее. Сначала пишем безо всякой красоты.

```
procedure Cr(    x,y : integer);  
  {параметры – столбец, строка}  
begin  
  SetColor(Red);  
  SetLineStyle( SolidLn, 0, ThickWidth);  
  Line( x0+(x-1)*s,    y0+(y-1)*s,  
        x0+(x-1)*s+s, y0+(y-1)*s+s);  
  Line( x0+(x-1)*s+s, y0+(y-1)*s,  
        x0+(x-1)*s,    y0+(y-1)*s+s);  
end;
```

Теперь неплохо бы всё это проверить. Надо сразу отбросить наивную веру в осмысленность и правильность написанного нами текста. Пишем мы ерунду с кучей ошибок. Соответственно, проверять написанное надо как можно раньше и чаще. Написали процедуру – тут же проверили. Написать всё сразу и потом всё сразу проверять – глупость несусветная и необсуждаемая. Так что проверяем. Между InitGraph и CloseGraph, там, где у нас будет потом самая главная программа, пишем:

```

DrawField;
Zero( 1, 3);
Cr( 3,1);

```

Вспоминаем, что первый параметр у нас номер столбца, второй параметр – номер строки. А почему мы вызвали с параметрами (1,3), а не (1,1)? Для того, чтобы отловить случай перепутанных местами координат. Ошибка совершенно обычная и несимметричные параметры очень хорошо её ловят. Всё работает? А я пугал, что работать без проверки не будет? Ну так я ведь уже всё сам проверил...

Работать-то оно работает, но как-то неаккуратно. Крестик наезжает на разметку поля. Надо бы крестик слегка пообкусывать по краям, по всем четырём. Ход мысли должен быть примерно таким – левый верхний край должен быть короче, то есть находиться правее и ниже. Значит, к обоим координатам надо что-то прибавить. Прибавить, конечно, что-то одинаковое. А если одинаковое – значит нужна константа. А поскольку константа эта применяться будет только внутри процедуры – то внутри процедуры её и объявим. Получаем:

```

procedure Cr(      x,y : integer);
{параметры - столбец, строка}
  const
    d = 5;
begin
  SetColor(Red);
  SetLineStyle( SolidLn, 0, ThickWidth);
  Line( x0+(x-1)*s+d,    y0+(y-1)*s+d,
        x0+(x-1)*s+s-d, y0+(y-1)*s+s-d);
  Line( x0+(x-1)*s+s-d, y0+(y-1)*s+d,
        x0+(x-1)*s+d,    y0+(y-1)*s+s-d);
end;

```

Проверить и подобрать D по вкусу!

### Курсор и чтобы бегал

Сначала курсор, потом чтобы бегал. Курсор будет внизу клетки и, уже ясно, чуть короче. Вот так:

```

procedure DrawCursor(  x,y : integer);
{параметры - столбец, строка}
const

```

```

    d = 5;
begin
    SetColor(LightBlue);
    SetLineStyle( SolidLn, 0, ThickWidth);
    Line( x0+(x-1)*s+d,    y0+(y-1)*s+s-d,
          x0+(x-1)*s+s-d,  y0+(y-1)*s+s-d);
end;
{-----}
}

procedure HideCursor(    x,y : integer);
{параметры - столбец, строка}
const
    d = 5;
begin
    SetColor(Black);
    SetLineStyle( SolidLn, 0, ThickWidth);
    Line( x0+(x-1)*s+d,    y0+(y-1)*s+s-d,
          x0+(x-1)*s+s-d,  y0+(y-1)*s+s-d);
end;

```

А как проверить? А вот так (дописываем в хвост).

```

DrawField;
Zero( 1, 3);
Cr( 3,1);
DrawCursor(3,2);
readln;
HideCursor(3,2);

```

Деликатный вопрос – а зачем отдельные процедуры для рисования и стирания? А вот захотелось мне так. Точнее – я решил, что это будет правильно. Размер кода небольшой, можно и продублировать (почти), зато наглядно. Не нравится – напишите одну универсальную процедуру с указанием цвета рисования.

В статике работает. А теперь в динамике, чтобы бегало. Этот цикл у нас будет в основной программе. Выглядеть сначала он будет так же, как и цикл раньше для бегающего тараканчика. Прочитали клавишу – стёрли тараканчика, где стоит, поменяли позицию, нарисовали тараканчика. Не забыть только инициализировать позицию тараканчика/курсора перед циклом и нарисовать курсор. А когда цикл закончится? Когда закончится игра - для этого у нас припасена переменная `gameover`. Или когда игроку надоест и он нажмёт `Esc`.

Ещё раз – ЭТО у нас будет в теле основной программы.

```

DrawField;
x:=1;
y:=1;
DrawCursor(x,y);
gamover:=false;

repeat
  if OurKeyPressed then begin
    sc:=OurReadKey;
    if sc = ArrowLeft then begin
      HideCursor(x,y);
      x:=x - 1;
      DrawCursor(x,y);
    end;
    if sc = ArrowRight then begin
      HideCursor(x,y);
      x:=x + 1;
      DrawCursor(x,y);
    end;
    if sc = ArrowUp then begin
      HideCursor(x,y);
      y:=y - 1;
      DrawCursor(x,y);
    end;
    if sc = ArrowDown then begin
      HideCursor(x,y);
      y:=y + 1;
      DrawCursor(x,y);
    end;
  end;
until (sc = Esc) or gamover;

Finale(result);

```

Замечание в сторону. В финальную процедуру мы передаём результат как параметр. Хотя могли бы сэкономить и воспользоваться соответствующей переменной, ведь её из процедуры видно. Самой процедуре всё равно, текст её от этого не изменился бы ни на копейку. Но – глобальные переменные есть зло! И мы будем их уничтожать, там, где встретим. Теперь по существу. Курсор резво бегает. Это хорошо. Но бегает где попало. Это плохо. Никаких ограничителей на выход за пределы поля у нас нет. Надо добавить. Ещё такой подход называется *защита от дураков*. Главное, чтобы дураки это не прочитали.

Для стрелки влево будет так:

```

if (sc = ArrowLeft) and (x>=2) then begin
  HideCursor(x,y);
  x:=x - 1;

```

```
    DrawCursor(x,y);  
end;
```

Для трёх оставшихся случаев первая строка условного оператора модифицируется соответственно:

```
if (sc = ArrowLeft) and (x<=2) then begin  
if (sc = ArrowLeft) and (y>=2) then begin  
if (sc = ArrowLeft) and (y<=2) then begin
```

Аккуратненько внесите соответствующие изменения. Проверьте.

### Делаем ход

Теперь придётся немного подумать. Сделать ход в выбранное место несложно. Стрелками перемещаемся в нужную клетку, нажимаем пробел и рисуем крестик.

```
if sc = SpaceBar then begin  
    Cr( x,y);  
end;
```

Только одна проблема. Даже не проблема – обстоятельство, являющееся нормой жизни программиста. Есть объекты на экране – курсор, крестики, нолики. И есть отображение этих объектов в программе. Есть курсор на экране. И есть переменные X и Y, помнящие, где курсор сейчас находится. Если немного поразмыслить, придём к выводу, что эти две переменные и есть, в сущности, курсор с точки зрения программы. Если есть крестик на экране, должен быть и крестик в программе. Мы об этом уже позаботились, объявив массив A, размером три на три. Если в соответствующей ячейке массива ноль – значит в этой клетке ничего пока нет. Если там единица – значит там крестик, если двойка – то нолик.

А дальше как всегда. Не забыть проинициализировать массив нулями перед началом главного цикла:

```
DrawField;  
x:=1;  
y:=1;  
DrawCursor(x,y);  
Gamover:=false;  
for i:=1 to 3 do  
    for j:=1 to 3 do  
        a[i,j]:=0;
```

А теперь сосредоточьтесь. Первый индекс массива традиционно отвечает за номер строки, второй индекс – за номер столбца. Переменная X не менее традиционно отвечает за горизонталь, то есть за номер столбца. Y это, соответственно, номер строки. И ещё одно – а если клетка занята? Живёт там уже кто-то – крестик или нолик? Значит, сначала надо проверить, пустая ли клетка.

```
if sc = SpaceBar then begin
  if a[y,x]=0 then begin
    Cr( x,y);
    a[y,x]:=1;
  end;
end;
```

### А не выиграл ли кто?

Вообще-то, было бы очень неплохо, если бы, прежде чем следовать за извилистым течением моей мысли, вы попробовали бы сначала запрограммировать соответствующую часть программы самостоятельно. Не получилось самостоятельно – идём дальше совместно. Вот заготовка процедуры проверки на завершение игры. Смотрим на неё и думаем.

```
{эта процедура проверяет завершение игры}
{результат: крестики выиграли - 1}
{нолики выиграли - 2}
{ничья - 3}
{игра не окончена - 0}
procedure Check( a : TArray3x3;
                 var result : integer);
begin
end;
```

Думаем об объявлении процедуры и о типе параметра A. Или просто вспоминаем, что это и зачем. Думаем, а откуда будет вызываться эта процедура. Думаем об этом даже раньше, чем мы эту процедуру написали, и это правильно. Мне кажется, что вызывать её надо сразу после хода крестиком. И как-то отреагировать на её результат. Как, кстати, отреагировать? Если её результат ноль, то никак. Игра продолжается. Впрочем, *никак* не совсем подходящее слово. Реакцией будет ход ноликов. А вот во всех трёх остальных случаях реакция, в сущности одинаковая – завершить игру, выйти из основного цикла и что-то такое специальное сделать. Не сейчас сделать, после цикла. А что означает выйти из цикла? `GameOver:=true`. А как мы узнаем после цикла,

чем игра закончилась? А мы дадим себе честное слово, что не испортим значение переменной result до выхода из цикла. Не испортим, разумеется, в том только случае, когда gamover истинно.

```
if sc = SpaceBar then begin
  if a[y,x]=0 then begin
    Cr( x,y);
    a[y,x]:=1;
    Check( a, result);
    if result = 0 then begin
      {жизнь продолжается}
    end;
    if (result=1) or (result=2) or (result=3) then begin
      gamover:=true;
    end;
  end;
end;
```

Вроде бы выглядит неплохо, правдоподобно. Начинаем думать о главном на текущий момент – что же мы такое напишем в нашу процедуру проверки?

А как могут выиграть крестики? В смысле, как крестики при этом могут быть расстановлены? Три крестика в первом ряду. Или три во втором. Или три в третьем. В первом, втором или третьем столбике. Наконец, по диагонали из левого верхнего угла в правый нижний угол. Или из левого нижнего в правый верхний. Кажется, ничего не пропущено. Итого восемь вариантов.

Приступаем

```
if ((a[1,1]=1) and (a[1,2]=1) and (a[1,3]=1)) or {первая строка}
  ((a[2,1]=1) and (a[2,2]=1) and (a[2,3]=1)) or {вторая строка}
  {ещё шесть раз}
  .....
then result:=1;
```

Невесело как-то получается. А как веселее, я не знаю. Если узнаете, расскажите мне. Можно слегка улучшить, если слегка подумать. Нам надо, чтобы каждое из трёх чисел равнялось единице. А чему, в таком случае, равняется их произведение? Единице, конечно! Исходя из этого пишем, сразу и для ноликов – для них только единицу меняем на восьмёрку.

Но лучше сразу подумать и про третий вариант – когда ничья. Что такое ничья в крестиках-ноликах? Когда некуда больше ходить. То есть, все клетки заполнены. По другому - все элементы массива не нули. И финальная деталь – их произведение не равно нулю.

```

procedure Check(      a      : TArray3x3;
                    var result : integer);
begin
    result:=0;
    if (a[1,1]*a[1,2]*a[1,3] = 1) or
        (a[2,1]*a[2,2]*a[2,3] = 1) or
        (a[3,1]*a[3,2]*a[3,3] = 1) or
        (a[1,1]*a[2,1]*a[3,1] = 1) or
        (a[1,2]*a[2,2]*a[3,2] = 1) or
        (a[1,3]*a[2,3]*a[3,3] = 1) or
        (a[1,1]*a[2,2]*a[3,3] = 1) or
        (a[3,1]*a[2,2]*a[1,3] = 1)
    then result:=1;

    if (a[1,1]*a[1,2]*a[1,3] = 8) or
        (a[2,1]*a[2,2]*a[2,3] = 8) or
        (a[3,1]*a[3,2]*a[3,3] = 8) or
        (a[1,1]*a[2,1]*a[3,1] = 8) or
        (a[1,2]*a[2,2]*a[3,2] = 8) or
        (a[1,3]*a[2,3]*a[3,3] = 8) or
        (a[1,1]*a[2,2]*a[3,3] = 8) or
        (a[3,1]*a[2,2]*a[1,3] = 8)
    then result:=2;

    if a[1,1]*a[1,2]*a[1,3]*
        a[2,1]*a[2,2]*a[2,3]*
        a[3,1]*a[3,2]*a[3,3] <> 0
    then result:=3;
end;

```

Помните, написав процедуры для рисования крестика и нолика, мы их аккуратно и тщательно (почти) протестировали? Как вы думаете, какая процедура сложнее, рисования нолика или процедура, только что написанная? Правильно. Так что настоятельно рекомендую бросить всё и написать таки программку для тестирования вот этой самой процедуры.

### Вражеский интеллект

Вот она, наша заготовка для вражбых мозгов:

```

procedure AI(      a      : TArray3x3;
                  var xkuda, ykuda : integer);
begin

```

```
end;
```

На входе – игровое поле с расставленными уже крестиками и ноликами. На выходе – столбец и строка, куда конкурирующий интеллект считает нужным поставить нолик. Начинаем думать. Какие возникают вопросы?

А откуда эту процедуру вызвать? Мне так кажется, что вызывать её надо в обработчике нажатия на пробел. То есть, ход моей мысли - и нашей программы - такой. Если клетка свободна, ставим крестик. Проверяем на завершение игры. Если завершилась, то завершилась. Если не завершилась, обращаемся к искусственным мозгам, мозги дают ответ, куда ходить. Ходим за ноликов. Опять проверяем, чем дело кончилось. Если что, устанавливаем gamover. И, собственно, всё. Программа готова.

```
if sc = SpaceBar then begin
  if a[y,x]=0 then begin
    Cr( x,y);
    a[y,x]:=1;
    Check( a, result);
    if result = 0 then begin
      {игра продолжается}
      AI( a, xkuda, ykuda);
      Zero(xkuda,ykuda);
      a[ykuda,xkuda]:=2;
      Check( a, result);
      if (result=1) or (result=2) or (result=3) then
begin
          gamover:=true;
        end;
      end;
    if (result=1) or (result=2) or (result=3) then begin
      gamover:=true;
    end;
  end;
end;
```

Посмотрели? Понравилось? Лично моё тонкое эстетическое чувство оскорблено этими повторяющимися строками со словом result. Так оставаться не должно и не будет. Варианты? Убрать gamover вообще, а главный цикл завершать по соответствующему значению result. Можно, но gamover жалко. Слово красивое. Да и вообще – использовать для завершения главного цикла программы переменную, которая применяется ещё и в других целях – пошло и потенциально опасно.

Кстати, практически в любой программе есть свой Главный Цикл.

Другое предложение – добавить в процедуру Check третий параметр. Не очень изящно. Можно преобразовать Check в функцию. Мне эта идея не нравится – функция, которая кроме своего значения возвращает ещё и модифицированные параметры – неправильно это, мягко говоря. Придётся всё-таки добавлять третий параметр. Заголовок процедуры Check с этого момента выглядит следующим образом:

```
procedure Check(      a      : TArray3x3;
                   var result : integer;
                   var gamover : boolean);
```

Вот эти строчки переезжают в конец процедуры Check - не забыть проинициализировать gamover в начале процедуры:

```
if (result=1) or (result=2) or (result=3) then begin
  gamover:=true;
end;
```

А в том тексте, что наверху, эти строки просто исчезают:

```
if sc = SpaceBar then begin
  if a[y,x]=0 then begin
    Cr( x,y);
    a[y,x]:=1;
    Check( a, result, gamover);
    if result = 0 then begin
      {игра продолжается}
      AI( a, xkuda, ykuda);
      Zero(xkuda,ykuda);
      a[ykuda,xkuda]:=2;
      Check( a, result, gamover);
    end;
  end;
end;
```

Эстетическое чувство всё равно беспокоит, но так гораздо лучше. Теперь другой вопрос. Как думает искусственный интеллект? Давайте думать сами – а как бы мы походили на его, интеллекта, месте. Мысль номер один - если можно выиграть, то нужно выиграть! Если у нас где-то уже есть два нолика в ряд, пристраиваем к ним третий – и всё! А если выиграть нельзя?

Мысль номер два. Если не можем выиграть сами, не дадим выиграть врагу, то есть живому игроку. Не съём, так понадкусываю! Если где-то стоят два крестика в ряд, ставим туда в качестве третьего наш вредный нолик. И противник уже не выиграл. Хорошо. А если ни того, ни другого – ни двух ноликов, ни двух крестиков?

Мысль третья. Если некуда ходить, походим в какое-нибудь хорошее место. Какое? Есть такое мнение, что лучший ход – в центр. Если центр занят – ходим в среднюю боковую клетку, их много - четыре. Если и они все заняты, ходим в угол. Угол наверняка свободен – если бы всё было занято, определилась бы ничья и мы бы в это место программы вообще не попали.

Ищем ход, которым могут выиграть нолики. Если нашли – ходим.  
Если не нашли:

Ищем ход, которым могут выиграть крестики. Если нашли – ходим.

Если не нашли:

Ходим в центр. Если нельзя:

Ходим в боковую клетку. Если нельзя:

Ходим в угловую клетку.

Всё хорошо. Осталось уточнить мелкие детали. Что значит – *Ищем ход, которым могут выиграть нолики*? Как и обычно, я предлагаю самый тупой вариант. Поочерёдно попробовывать поставить в каждую свободную клетку нолик, и, если мы выигрываем – значит мы выигрываем... Кстати, процедура, проверяющая, не выиграли ли мы, у нас уже есть. Называется Check, если кто забыл. Если ответ положительный – запоминаем, куда ходить и возвращаем в качестве результата. Вот только не надо использовать в процессе переменную gamover в качестве одноимённого параметра. Gamover – это святое, его портить нельзя. Хотя, конечно, тождественность имён так и тянет вписать его в вызов процедуры. Тем не менее, объявим для этих целей специальный, временный gamover. Нюанс – если мы что-то нашли, дальше искать уже не надо. Напрашивается логическая переменная, отвечающая за это дело – непродолжение, в смысле.

Ещё уточняем - цикл три на три, пытаемся в каждую свободную клетку поставить нолик, вызываем Check, смотрим на результат. Если всё, то всё, в смысле – ходим туда, иначе – продолжаем искать.

```

procedure AI(      a      : TArray3x3;
                 var xkuda, ykuda : integer);
var
  b      : TArray3x3;
  nashli : boolean;
  tmpGamover : boolean;
  i, j    : integer;
begin
  nashli:=false;

  for i:=1 to 3 do begin
    for j:=1 to 3 do begin
      if a[i,j] = 0 then begin
        b:=a;
        b[i,j]:=2;
        Check( b, result, tmpGamover);
        if result = 2 then begin
          xkuda:=j;
          ykuda:=i;
          nashli:=true;
        end;
      end;
    end;
  end;

  if not nashli then begin
    {продолжаем разговор}
  end;

```

Обратите внимание, я взял на себя инициативу и ввёл вспомогательный массив В. А что было бы, если не? Попробуйте, хотя бы на абсолютно пустом массиве А. В процессе экспериментов он очень быстро замусорился бы двойками (ноликами) и мы получали бы совершенно неуместные радостные сообщения о победе. Я прошёл через это. Для избежания такого безобразия и нужен массив В, который не жалко испортить для очередного эксперимента – всё равно на следующем шаге мы вернём его а исходное состояние.

Ещё обратите внимание на оператор присваивания В:=А. Да, такое возможно, но только при условии, что массивы объявлены абсолютно идентично. И когда я говорю “абсолютно идентично”, именно это я и имею в виду, и даже хуже. Нельзя объявить в двух строчках массивы А и В как array[1..3,1..3] of integer. С точки зрения Паскаля, это разные объявления. Вы можете не соглашаться с Паскалем, но его мнение

решающее. Идентичные – это когда объявляются одним словом – string, integer, single, TArray3x3.

Теперь быстро изготавливаем продолжение – для того случая, когда мы можем помешать крестикам выиграть. Почти то же самое.

```
if not nashli then begin
  for i:=1 to 3 do begin
    for j:=1 to 3 do begin
      if a[i,j] = 0 then begin
        b:=a;
        [i,j]:=1;
        Check( b, result, tmpGamover);
        if result = 1 then begin
          xkuda:=j;
          ykuda:=i;
          nashli:=true;
        end;
      end;
    end;
  end;
end;
```

Вроде бы, всё понятно и дополнительных комментариев не требует. Осталось собраться с силами и дописать вариант про то, не знаю что, – куда ходить, когда некуда ходить. Предлагаю просто и без затей. Вопросы, конечно, возникнут. Ознакомьтесь с текстом, затем ответы на предполагаемые вопросы.

```
if not nashli then begin
  if a[2,2] = 0 then begin
    xkuda:=2;
    ykuda:=2;
    nashli:=true;
  end;
  if not nashli then begin
    if a[2,3] = 0 then begin
      xkuda:=3;
      ykuda:=2;
      nashli:=true;
    end;
    if a[2,1] = 0 then begin
      xkuda:=1;
      ykuda:=2;
      nashli:=true;
    end;
    if a[1,2] = 0 then begin
      xkuda:=2;
    end;
  end;
end;
```

```

        ykuda:=1;
        nashli:=true;
    end;
    if a[3,2] = 0 then begin
        xkuda:=2;
        ykuda:=3;
        nashli:=true;
    end;
end;
if not nashli then begin
    if a[1,1] = 0 then begin
        xkuda:=1;
        ykuda:=1;
        nashli:=true;
    end;
    if a[3,3] = 0 then begin
        xkuda:=3;
        ykuda:=3;
        nashli:=true;
    end;
    if a[3,1] = 0 then begin
        xkuda:=1;
        ykuda:=3;
        nashli:=true;
    end;
    if a[1,3] = 0 then begin
        xkuda:=3;
        ykuda:=1;
        nashli:=true;
    end;
end;
end;
end;

```

А теперь – возникшие у вас вопросы и сомнения. Попытаюсь телепатически угадать и не телепатически ответить.

А нельзя это сделать короче? Можно, только я не знаю как. Если знаете, скажите мне. Я не гордый, всегда готов научиться новому фокусу. У себя на работе, сотрудников, которые учатся фокусам, я стараюсь загнубить, сгноить и изничтожить.

А зачем в последних четырёх условных операторах мы усердно пишем nashli:=true, ведь nashli нам уже больше не понадобится? Во-первых, чтобы не думать. Во-вторых, для симметрии. В-третьих, сегодня не понадобится, а завтра кто его знает. Случаи, как известно, разные бывают.

Если кто-то думает, что я шучу, он заблуждается. Никакого юмора, только суровые будни профессионального программиста.

А зачем мы, найдя свободной, например, первую же угловую клетку, ищем дальше, проверяя остальные три? Вопрос серьёзный. В некоторых случаях - не в нашем - это означает дополнительную трату времени. Если свободны несколько угловых клеток, мы всегда вернём последнюю, а не первую. В некоторых случаях - опять-таки, не в нашем - это важно. Так зачем? Во-первых – работает и ладно. Во-вторых – а как? Бесконечно вложенный условный оператор? Нафиг-нафиг. Есть несколько более изящный метод, но, сугубо из педагогических соображений, я оставил его на потом. И, ещё раз о главном, работает и ладно. Добро пожаловать в реальный мир.

А хорошо ли, что если все четыре угловые клетки свободны, искусственный интеллект всегда ходит в левую верхнюю? Плохо. Но это я оставил для вас.

### **Имеем в результате**

Ну и что осталось сделать? Мне – ничего. А вам – аккуратненько собрать всё написанное в единый текст программы и доработать напильником. Начнём с того, чем закончили предыдущий раздел - случайный выбор одной из нескольких свободных клеток. Вся печаль в том, что клеток мало, максимум четыре. И получаем известную картину стрельбы из пушек по воробьям. Было бы клеток штук сто... Стандартный ход мысли – составить список пустых клеток, попутно узнав их количество, затем выбрать случайную клетку. Для наших скромных масштабов – ужас какой-то получается. Поразмышляйте – возможно, вас посетит Идея.

Ну и когда всё это закончилось - в смысле, основной цикл - как-то надо отреагировать на происшедшее. Для этого у вас есть переменная `result`. Как минимум, напишите кто проиграл, а кто выиграл. Или изобразите фейерверк. Или нарисуйте линию, перечёркивающую три крестика или три нолика. А лучше, всё сразу.

## Глава 10

### Файлы

#### Коротенько. Почему это очень важно

А действительно, кто такие файлы, и почему они нам так важны - а они действительно так важны. Вам не казалось, что программы наши носят несколько эфемерный характер? Отработала программа, и что осталось после неё? Всё прошло, всё забыто, всё начинать по новой. При этом любая игрушка после запуска не начинает жизнь сначала, а предлагает восстановить прежнее состояние, или, по крайней мере, ведёт таблицу рекордов. Многие программы изначально заточены на обработку внешних данных, например Word или Photoshop. Да тот же Турбо Паскаль, в конце концов, не заставляет ведь каждый раз набирать программу заново. Помнит он её где-то.

Помнит он её в файле. Я говорил уже о важнейших понятиях программирования – переменная, массив, указатель. К ним добавляется и файл. От других понятий файл отличается тем, что он вещь реальная, существующая независимо от нашей программы. Реальная, разумеется, в компьютерном смысле слова, но уж в этом-то смысле реальная безусловно – программа работать закончила, а файл остался. Вот о них-то, о файлах, мы сейчас и поговорим.

И о главном – файлы бывают текстовые и бинарные. На самом деле, все они, разумеется, бинарные, но разделение такое общепринято и вполне разумно.

#### Найти и снова найти

Но начнём мы не с того, что у файлов внутри а, так сказать, с виду сверху. Файл, как сказано, вещь реальная. Или он есть, или его нет. Или он есть, но их много. Отсюда вопрос – как узнать - есть он, или его нет, или их много? Ещё более другими словами – как узнать, какие файлы существуют? На этот вопрос немедленно возникает вопрос встречный – а где существуют? Упрощаем вопрос - как узнать, какие файлы существуют в каталоге, из которого запущена наша программа?

Узнать несложно, но, как бы это помягче выразиться – методом немного корявым. Сначала объявляем переменную типа SearchRec.

```
var
    SR          : SearchRec;
```

SearchRec – запись, а записи у нас по плану (моему) в следующей главе. Так что не обращаем внимания и делаем вид, что всё понимаем. На самом деле, записи - это очень просто. Вот только доберёмся до следующей главы... А вот и сама программа:

```
FindFirst( '*.*', AnyFile, SR);

while DosError = 0 do begin
    writeln( SR.name);
    FindNext(SR);
end;
```

На выходе получаем список файлов, расположенных в текущем каталоге – том каталоге, из которого запущена программа.

Два странных файла с именами в виде одной и двух точек соответственно – особенность DOS'а. Если в каталоге присутствуют подкаталоги, то в списке окажутся и они. Управляет этим второй параметр процедуры. AnyFile означает, что будут найдены все файлы, включая каталоги. С точки зрения DOS'а, каталог – это такой специальный файл. Почитайте в справочной системе на эту тему, в частности, как искать только файлы. Первый параметр, самый важный, - маска поиска. Почитайте что-нибудь про DOS, в частности о команде dir. Если нас интересует какой-то конкретный файл, то в качестве первого параметра надо указать его имя. И почитайте в справочной системе описание структуры SearchRec. В ней присутствует и другая полезная информация.

Если слово DOS вам скучно и неинтересно, то Windows внутри себя тот же DOS. Не совсем, конечно, но во многих отношениях.

Обратите внимание, что команда поиска файлов используется в двух вариантах. Первый раз – в полной – FindFirst. Второй и следующий в сокращённой – FindNext. Подразумевается, что первые два параметра из FindFirst незримо присутствуют в FindNext с теми же значениями. И ещё загадочный DosError. Это такая переменная, не нами объявленная. Пока очередной искомый файл находится - и вообще всё хорошо - переменная эта равна нулю. Как только что-то становится не так, значение становится ненулевым.

Обратите внимание на организацию цикла **while**. У нас есть некоторое действие (Find), в зависимости от результата которого выполнение цикла может быть прекращено. И есть обработка результатов этого действия - вывод имени файла, в нашем случае. Первый Find производится перед циклом. А в цикле всё идёт в обратном порядке – сначала обработка, затем действие. Это типично.

И не думайте, что всё это можно немедленно забыть, потому что это DOS и Turbo Pascal. Напоминаю, при программировании под Windows всё почти так же, только хуже.

### **Файлы текстовые и никому не нужные**

Запустите Блокнот. Напишите в нём какую-нибудь ерунду. Сохраните. Запустите FAR, Total Commander, Проводник, или что-нибудь аналогичное. Полубуйтесь на созданный файл. Существует? Безусловно. Расширение у файла будет .txt. Такой файл называется текстовым – не потому, что у него такое расширение, и даже не потому, что Проводник обозвал его *Текстовым документом*, а потому что он содержит только текст.

Более формализовано – текстовый файл содержит символы с кодами от какого-то значения до какого-то значения (я, естественно, этих значений не помню), в диапазон которых (кодов этих символов) как раз и попадают буквы, циферки и тому подобные видимые символы. Из невидимых символов в правильных текстовых файлах присутствуют ещё возврат каретки с кодом 13 и перевод строки с кодом 10. Все эти ужасы со странными именами – наследие давно забытых дней. Просто примите как данность. Встречаются в природе и неправильные текстовые файлы, у которых переход на новую строку кодируется по-другому. Иногда попадают в остальном правильные файлы, у которых в самом конце присутствует символ, кодирующий признак конца файла. Не обращайтесь внимания, если не собираетесь писать свой собственный текстовый редактор.

И ещё – не всё, что выглядит как текст, является текстовым файлом. Документ Word'a выглядит вполне текстово, но к текстовым файлам не

относится. И совсем напоследок. То, о чём мы говорим, в смысле, не сам файл, а его содержимое, по-английски обычно называется *plaint text*.

Хотя текстовые файлы сами по себе большого интереса для программиста не представляют и большой пользы не приносят, тем не менее техника работы с ними очень похожа на технику работы с бинарными файлами. А вообще, и то, и другое, очень просто, как и всё в Паскале. Объявляем текстовый файл:

```
var  
    Txt           : Text;
```

Работаем с ним – создаём новый файл и записываем в него текст. Как правило, во всех языках работа с файлом делится на три этапа – создать или открыть файл, сделать что-то с его содержимым, закрыть файл.

```
{открыть файл}  
Assign( Txt, 'a.txt');  
ReWrite(Txt);  
  
{записать что-то в файл}  
writeln( Txt, '12345');  
writeln( Txt, '67890');  
  
{закрыть файл}  
Close(Txt);
```

Комментируем.

Открытие файла в Паскале выполняется в два этапа. Assign связывает объявленную в программе файловую переменную Txt с физическим файлом на диске – уже существующим или тем, которого ещё нет, но который будет создан в процессе выполнения программы. Имя может включать в себя и путь, то есть файл может быть открыт и в другом каталоге. Разумеется, расширение файла не обязательно будет .txt. Немного другими словами – у нас есть переменная. Тип этой переменной – не целое, не строка. Тип её – текстовый файл. Переменная существует, как и любая переменная, только и только в пределах и во время выполнения нашей программы. Assign выполняет действие небывалой важности и новизны – указывает, что нашей эфемерной переменной соответствует реальный объект реального мира – файл. Ну, естественно, если вы верите в реальное существование файла...

Запись в текстовый файл поразительно напоминает вывод на экран. Только в скобках появился дополнительный, первый параметр – имя файловой переменной. Обратите внимание – не имя реального файла – того, что на диске – а имя переменной. Ещё раз извиняюсь, если вы давно всё уже поняли, а я всё долблю, как дятел. Точно так же, как и на экран, можно выводить в текстовый файл и значения переменных.

```
writeln( Txt, int:5, ' ', float:8:3, ' ', stroka);
```

Закрытие файла вопросов не вызывает (надеюсь).

Теперь чтение. S – строковая переменная. Имеем почти то же самое, что и для записи:

```
{открыть файл}  
Assign( Txt, 'a.txt' );  
ReSet( Txt );
```

```
{прочитать что-то из файла}  
readln( Txt, s );
```

```
{закрыть файл}  
Close( Txt );
```

Только ReWrite поменялся на ReSet, и вместо writeln написан readln, почти точно такой же, как и знакомый readln из текстового режима – только добавилось имя файла (повторюсь – имя файловой переменной).

В качестве самостоятельного упражнения модернизируйте программу из предыдущего раздела. Пусть она выводит список файлов не на экран, а в текстовый файл. Имя файла запросить у пользователя . Через каждые десять строк в список вставлять пустую строку. В конце списка вывести общее число файлов.

Уже модернизировали? Вот и славно. А теперь ещё упражнение. Вернёмся к нашим крестикам-ноликам. Хочу, чтобы после завершения игры, результат - кто кого победил - записывался в текстовый файл. Если файла нет, он должен быть создан. Если файл есть, новый результат дописать ему в хвост. А как это, дописать в хвост? У текстовых файлов есть счастливая возможность – кроме ReSet и ReWrite их также можно открыть командой Append. После этого все новые записи в файл будут

приклеиваться к уже имеющемуся в файле. Проверку, существует ли файл, производим с помощью FindFirst .

**Первый вариант:**

```

procedure Finale(      result : integer);
  const
    ftxtname = 'ktokogo.txt';
  var
    Txt           : Text;
    SR            : SearchRec;
begin
  { тут то, что вы уже понаписали. С фейерверком, надеюсь }

  FindFirst( ftxtname, AnyFile, SR);

  if DosError = 0 then begin  { файл есть }
    Assign( Txt, ftxtname);
    Append(Txt);
    if result = 1 then writeln( Txt, 'You Win!');
    if result = 2 then writeln( Txt, 'You Lost!');
    if result = 3 then writeln( Txt, 'Draw!');
    Close(Txt);
  end
  else begin  { файла нет }
    Assign( Txt, ftxtname);
    ReWrite( Txt);
    if result = 1 then writeln( Txt, 'You Win!');
    if result = 2 then writeln( Txt, 'You Lost!');
    if result = 3 then writeln( Txt, 'Draw!');
    Close(Txt);
  end;

```

Ни в коем случае не являясь сторонником сокращения текстов программ до минимально возможного размера, тем не менее должен заметить, что меня не очень радуют монотонно повторяющиеся строки с writeln. А не вынести ли их нам в отдельную процедуру? И вынесем! Вариант улучшенный:

```

procedure Finale(      result : integer);
  const
    ftxtname = 'ktokogo.txt';
  var
    Txt           : Text;
    SR            : SearchRec;
  { ..... }
  procedure WriteResult( var Txt      : Text;
                        result : integer);
begin

```

```

    if result = 1 then writeln( Txt, 'You Win!');
    if result = 2 then writeln( Txt, 'You Lost!');
    if result = 3 then writeln( Txt, 'Draw');
end;
{.....}
begin
    {тут фейерверк}

    FindFirst( ftxtname, AnyFile, SR);

    if DosError = 0 then begin
        Assign( Txt, ftxtname);
        Append(Txt);
        WriteResult( Txt, result);
        Close(Txt);
    end
    else begin
        Assign( Txt, ftxtname);
        ReWrite( Txt);
        WriteResult( Txt, result);
        Close(Txt);
    end;
end;

```

Комментарии. В процедуру передаются в качестве параметров файл и результат. Это правильно. Пользоваться изнутри процедуры внешними переменными нехорошо. Глобальные переменные – зло. Передаётся в процедуру не имя файла, а сам файл. Подумайте – опять - об этом. Файл передаётся с **var**, хотя сам файл не меняется, меняться будет его содержимое. Думать об этом не надо. Если это кажется вам нормальным, так тому и быть. Если нет, запомните, что файлы в качестве параметров *всегда* передаются с **var**.

Возможно у вас самозародились мысли о дальнейшем уплотнении программы. Вынести Assign выше, перед условным оператором. Отказаться от объявления процедуры, а её текст однократно вписать после условного оператора. Туда же отправить и Close. Короче, сегодня я добрый, и промолчу, но вам должно быть стыдно. Не надо так делать. Никогда.

## Бинарные

Бинарные файлы бывают типизированные и нетипизированные. Типизированные – для слабых духом. Для нас – только нетипизированные файлы.

Объявляем, открываем, записываем, закрываем.

```
var
  f           : file;
  int        : integer;
  a          : array[1..10] of integer;

{открыть файл для создания}
Assign( f, 'a.bin');
ReWrite( f, 1);

{записать что-то в файл}
BlockWrite( f, int, 2);

{закрыть файл}
Close(Txt);
```

Открытие бинарного файла очень похоже на таковое же открытие текстового. Только заметьте волшебную единицу в ReWrite. Так надо, поверьте! Заккрытие файла ничуть не изменилось. А вот на записи остановимся подробнее.

Первый параметр – имя файла. Это понятно. Второй параметр – имя переменной, значение которой мы хотим записать в файл. Точнее, это так, но только в первом приближении. На самом деле второй параметр – это адрес, начиная с которого производится запись в файл, не взирая ни на какие переменные. Но об этом пока можно забыть. Третий параметр - это размер записываемой переменной, или, точнее, количество записываемых в файл байтов.

На самом деле и это не так. Циферка 1 в открытии файла означает размер блока в байтах, то есть минимального количества данных, которое можно записать в файл и, соответственно, прочитать из него. Третий параметр в BlockWrite означает на *самом деле* количество записываемых блоков. Кстати, если вы забудете задать второй параметр в ReWrite, то размер блока будет по умолчанию установлен в 128, что почти наверняка является не тем, на что вы надеялись.

Ответственным за соответствие размера переменной и количества записываемых байтов назначаетесь вы. Это вы должны помнить, что целая переменная занимает именно два байта. Впрочем, этот труд можно облегчить.

Вместо

```
BlockWrite( f, int, 2);
```

пишем

```
BlockWrite( f, int, SizeOf(int));
```

Незатейливая, но очень полезная функция `SizeOf` возвращает количество байтов, занимаемых переменной. Если мы хотим записать в файл весь массив `A` целиком, то закодируем следующее:

```
BlockWrite( f, a, SizeOf(a));
```

или

```
BlockWrite( f, int, 20);
```

Но если мы напишем

```
BlockWrite( f, int, 10);
```

то в файл попадут только первые пять элементов массива. Ещё раз – `BlockWrite` игнорирует, где кончается одна переменная, и где начинается другая.

А теперь чтение:

```
Assign( f, 'a.bin');
```

```
ReSet( f, 1);
```

```
BlockRead( f, int, 2);
```

```
Close(Txt);
```

Вроде бы всё понятно. `ReWrite` поменяли на `ReSet`, `BlockWrite` на `BlockRead`. Далее о маленькой разнице между `ReSet` и `ReWrite`. `ReWrite` – если файл есть, то он его переписет, как будто файла и не было – уничтожив всё содержимое. Если файла нет, то он его создаст. `ReSet` – если файл есть, то он его откроет на чтение. Если файла нет – всё кончится очень плохо. Так что проверьте существование файла перед открытием с помощью `FindFirst`.

Отметим существенную разницу с текстовым файлом. Нам не нужно знать, что именно было записано в текстовом файле, для того, чтобы его прочитать. Достаточно знать, что этот файл текстовый. При каждом считывании из файла мы получаем одну строку. Единственное, что важно

– вовремя остановиться, то есть не попытаться прочитать что-нибудь ещё после того, как из файла всё уже, собственно, прочитано. С бинарным файлом не так. Чтобы прочитать его содержимое, мы должны абсолютно точно знать, что же именно в этот файл было записано – какие переменные, каких типов, какого размера, в каком порядке. Малейшая ошибка – и на выходе куча мусора.

Так в чём преимущество бинарных файлов? А в них можно записать всё. Нет, не так – ВСЁ! И вы это ещё оцените.

### Бинарные файлы. Задача

Задача простенькая. Скопировать существующий файл под другим именем. Если бы в модуле Dos была соответствующая процедура, то это было бы делом одной строки. Но процедуры нет. Будем делать собственными ручками. Дело нехитрое – проверить, что файл существует, прочитать файл, записать файл. Прочитать всё сразу мы пока не умеем, так что слона будем есть по частям, мелкими кусками. Организуем цикл по количеству кусков – прочитать кусок, записать кусок. Где-то так:

```
Ввести имя входного файла
Ввести имя выходного файла
Если входной файл существует
    Цикл по количеству кусков
        Прочитать кусок из входного файла
        Записать кусок в выходной файл
```

Куски, безусловно, должны быть одинакового размера. Было бы хорошо, чтобы файл делился на куски без остатка. А на какое число делится всё на свете без остатка? Правильно, на единицу! Следовательно, куски будут по одному байту. Цикл – по количеству байт в файле. А размер файла узнаем из записи SearchRec, он, размер, там есть.

Проверку на наличие файла выполним через единственный, без продолжения, FindFirst. И не забыть – объявить файлы, открыть файлы, закрыть файлы.

```
program filecopy;
uses
    Dos;
```

```

var
  inname, outname           : string;
  infile,outfile           : file;
  SR                        : SearchRec;
  b                          : byte;
  i                          : integer;
begin
  write('in name = ');
  readln(inname);
  write('out name = ');
  readln(outname);
  FindFirst( inname, AnyFile, SR);

  if DosError = 0 then begin
    Assign( infile, inname);
    ReSet(infile,1);
    Assign( outfile, outname);
    ReWrite(outfile,1);

    for i:=1 to SR.size do begin
      BlockRead( infile, b, 1);
      BlockWrite( outfile, b, 1);
    end;

    Close(outfile);
    Close(infile);
  end;
end.

```

И небольшой комментарий. Читаем из файла мы в переменную типа `byte`. С тем же успехом могли бы использовать `char` (тоже один байт), хотя это вызвало бы нехорошие сомнения у читателей нашей программы. Какие такие читатели? Тот программист, которому досталось её сопровождать. Хотя обычно проще программу выкинуть и написать новую.

### К чему-нибудь прикрутим

Вернёмся к крестикам-ноликам. Какая там может быть польза от файлов? Предложение сохранять и восстанавливать игру в процессе можно рассматривать только как издевательство в особо изощрённой форме. Но давайте хотя бы посчитаем, сколько игр за всю историю закончилось победой крестиков, а сколько – ноликов.

Была там у нас зарезервирована процедура `Finale`, внутри которой у нас – у вас - уже есть фейерверк и запись в текстовый файл. Из всех переменных нашей программы процедуре этой нужна только переменная

result, чтобы узнать кто выиграл. Вот в неё, эту процедуру, мы и приспособим наше файловое хозяйство.

Как оно должно быть устроено? При работе с файлами всегда надо помнить, что файла этого может ещё/уже и не быть. Так что вначале проверяем, существует ли файл. Имя файла, разумеется, находится в разделе констант. Если файл есть, читаем из файла информацию о числе побед крестиков и ноликов. Информация – два целых числа, других предложений ведь не будет? Дальше, что существенно, наша деятельность протекает совершенно независимо от того, нашли мы наш разлюбезный файл, или нет. Обновляем информацию, в зависимости от результатов текущей игры. Записываем файл как новый, вне зависимости от того, существовал он, или нет. Программа практически готова, кроме шуток. Формализуем чуть формализованнее:

Если файл есть

    Прочитать файл

Обновить данные

Записать данные в файл

Дальше тривиально. Не забыть объявить две переменные, для побед и для поражений – мы, конечно, смотрим с точки зрения крестиков. И, что не менее важно, не забыть их проинициализировать, на случай отсутствия файла. Ту часть, где мы писали результаты в текстовый файл, я опускаю, чтобы не отвлекала внимание.

```
procedure Finale(      result : integer);
  const
    fbinname = 'itogo.dat';
  var
    f          : file;
    SR         : SearchRec;
    win, lost  : integer;
begin
  {тут ваш фейерверк}

  {тут мы писали в текстовый файл}

  {а тут наш бинарный файл}
  FindFirst( fbinname, AnyFile, SR);

  win:=0
  lost:=0;
  if DosError = 0 then begin
```

```

    Assign( f, fbinname);
    ReSet( f, 1);
    BlockRead( f, win, 2);
    BlockRead( f, lost, 2);
    Close(f);
end;

if result = 1
    then win:=win + 1;
if result = 2
    then lost:=lost + 1;

Assign( f, fbinname);
ReWrite( f, 1);
BlockWrite( f, win, 2);
BlockWrite( f, lost, 2);
Close(f);

{а тут порауйте пользователя, сообщите, кто впереди}
end;
```

Мы негласно за собственной спиной договорились сами с собою, что в начале файла у нас идёт число побед, за ним следует число поражений. Константу с именем файла лучше бы объявить в разделе констант самой программы, в процедуре она объявлена только для наглядности, чтобы всё под рукой было. Для проверки существования файла также можно использовать функцию FSearch. Assign теоретически можно было написать один раз в начале – но не надо жмотничать!

Как отобразить результаты? Согласно Моему Стратегическому Плану Обучения, выводить числа в графическом режиме вы научитесь в следующей главе. Пока хотя бы нарисуйте пару столбиков, высота которых пропорциональна количеству побед крестиков и ноликов. А в остальном всё хорошо...

# Глава 11

## Всякие глупости, она же

### Глава очень длинная

#### Записи. И как мы только без них обходились!

Записи – это редкий пример простой и полезной вещи. В порядке исключения не будем начинать с объявления записи, в которой нет ни одного поля. Сразу объявим правильную запись и применим её к делу. Помните программу с мигающими звёздочками? Нам надо было хранить информацию об их координатах. Для этого мы объявили два массива – для координаты X и для координаты Y отдельно, и по отдельности, но всегда рядом, этими массивами пользовались:

```
starX           : array[1..maxStar] of integer;
starY           : array[1..maxStar] of integer;
.....
    starX[i]:=x;
    starY[i]:=y;
.....
    PutPixel( starX[star], starY[star], Black);
```

Массив, как вы очень хорошо усвоили, объединяет в себе несколько, обычно много, однотипных элементов. Запись объединяет несколько, обычно немного, разнотипных элементов. Массив может объединять записи. Запись может включать в себя массивы.

Наш первый пример записи не вполне показателен – оба поля записи будут однотипны. Помните мы упоминали объявление типов, в связи с передачей массивов в качестве параметров? В случае записей настоятельно рекомендуется объявлять их как типы. Это значительно повышает читаемость программы, особенно в нашем случае, когда используется не просто переменная, объявленная как запись, а массив записей.

```
type
  TPos = record
        x           : integer;
        y           : integer;
      end;
var
  thestars         : array[1..maxStar] of TPos;
```

```

.....
  thestars[i].x:=x;
  thestars[i].y:=y;
.....
  PutPixel( thestars[star].x, thestars[star].y, Black);

```

Пример записи посложнее. Встречавшийся уже SearchRec из поиска файлов. Вот объявление этой записи:

```

type
  SearchRec = record
    fill      : array [1..21] of byte;
    attr      : byte;
    time      : longint;
    size      : longint;
    name      : string[12];
  end;

```

Здесь имеем типы в широком ассортименте, включая один массив.

Разработчики Турбо Паскаля не знали ещё, что имя типа – у правильных программистов - должно начинаться с буквы T.

Повторим с самого начала, стараясь всё повторяемое как-то систематизировать. Объявление переменной типа запись начинается с нового слова **record** и кончается старым словом **end**. Между ними объявляются поля записи. Объявление отдельно взятого поля ничем не отличается от объявления одной отдельно взятой переменной. Поле может быть любого разумного типа. Надеюсь, что ваша фантазия не выйдет за пределы разумного. Вот так, без объявления нового типа, можно объявить одну переменную как запись:

```

var
  st      : record
            x      : integer;
            y      : integer;
          end;

```

Наш массив тоже можно было объявить непосредственно:

```

var
  thestars      : array[1..maxStar] of record
                    x      : integer;
                    y      : integer;
                  end;

```

Лично я от таких объявлений пугаюсь и пытаюсь спрятаться. Лучше объявлять через тип. Правила идентичности типов, о которых мы говорили в связи с массивами, распространяются и на записи. К записи, как и к массиву, можно обратиться как к единому целому. То есть, для предыдущего объявления мы можем написать так:

```
thestars[i]:=thestars[i+1];
```

Это равнозначно такому коду:

```
thestars[i].x:=thestars[i+1].x;  
thestars[i].y:=thestars[i+1].y;
```

Чаще происходят обращения к отдельным полям записи, как мы наблюдали на примере SearchRec. Обратите внимание – у нас есть переменная X и есть поле записи X и никому это не мешает. Главное, конечно, что это не мешает транслятору.

О записях в плане их интимных отношений с файлами. Запись записи в файл:

```
BlockWrite( f, st, SizeOf(st));
```

Чтение, соответственно:

```
BlockRead( f, st, SizeOf(st));
```

## Указатели

Как и обещано – самое-самое важное и самое-самое ужасное. Не то, что бы оно было тут очень нужно. Многие учебники начального уровня обходятся без указателей. Трудности видны сразу, а польза неочевидна. Но мне кажется, надо представлять, с чем придётся встретиться в реальной жизни. Дальше будет много непонятных слов и загадочных телодвижений. Смотрите и наслаждайтесь.

Как и файлы, указатели бывают типизированные и нетипизированные.. И, как и в случае с файлами, нас интересуют только нетипизированные.

Как объявить указатель? Очень просто.

```
var
    p           : pointer;
```

А присвоить указателю значение? Это сложнее. Надо всё же объяснить сначала, что такое указатель. Указатель, как и следует из своего имени, указывает. На что указывает? На область памяти. Понятно? Нет? Неважно, продолжаем. Точнее, возвращаемся к вопросу присваивания указателю значения. Целой переменной можно присвоить другую целую переменную, вернее значение другой целой переменной, строке можно присвоить другую строку. Указатель не исключение. Объявляем:

```
var
    p1,p2       : pointer;
```

Теперь можем написать

```
p1:=p2;
```

А смысл? Смысл в том, что указатель p1 имеет теперь тоже значение, что и p2 и указывает на ту же область памяти. А на что указывает p2?

Первый способ присвоить значение указателю (сначала добавим объявлений):

```
var
    p1,p2       : pointer;
    int1,int2,int3 : integer;
    a           : array[1..10] of integer;

    p1:=@int1;
    p1:=Addr(int1);
```

Способ я пообещал один, но операторов написал два. Причина в том, что это синонимы. Оба выражения дают один и тот же результат – указатель p1 указывает на переменную int1. Разумеется, это не означает, что указатель имеет тоже значение, что и int1. Значение указателя вообще имеет мало отношения к значению переменной, на которую он указывает. Указатель – это адрес переменной.

```
p1:=@a;
p2:=@a[1];
```

На что указывают эти указатели? Первый – на массив А, второй – на первый элемент этого массива А[1]. В результате, значения этих двух указателей одинаковы, ведь массив начинается с первого элемента. Понятно? Непонятно? Продолжаем.

Второй способ проинициализировать указатель – выделить память. Выделение памяти – операция симметричная, в смысле, если память выделили, то её надо будет освободить. И ответственный за это вы. Выделяем:

```
GetMem( p1, 10);
```

Мы запросили и получили десять байтов. На эти десять байтов теперь указывает указатель p1. А что находится в этих десяти байтах? Да ничего.

Напоминаю – если память выделили, надо и освободить! Вот так:

```
FreeMem( p1, 10);
```

Тот же указатель, и тот же размер памяти. Никакого контроля нет! Отвечаете за всё вы! Вы должны помнить, сколько байтов и для какого указателя были выделены. Постарайтесь не перепутать. Еще раз – в переменной типа указатель хранится вовсе не значение переменной, на которую этот указатель указывает. Там хранится адрес этой переменной. А если написать вот так: P^ - то это будет уже область памяти, на которую указатель указывает. Использовать это обозначение P^ *как есть* можно только в очень небольшом количестве случаев, в частности в BlockWrite и BlockRead.

Ещё одна новость. Для целых и дробных переменных есть нулевое значение. Для строки есть пустая строка. Указатель ничем не хуже. Для него существует специальный как бы указатель, константа, которая никуда не указывает. Выглядит присвоение пустого указателя вот так:

```
p:=nil;
```

**nil** – заранее определённая в Паскале константа. Это указатель, который никуда не указывает. Внутри у него банальный ноль. В случае обращения к памяти, на которую *как бы* указывает этот указатель, всё кончается очень и очень печально.

Для того, чтобы хоть что-то стало понятно, запрограммируем небольшую задачу. Задача уже была поставлена и решена – скопировать файл (небольшой) под другим именем. Решим её по-другому, избавившись от цикла. Зачем здесь нужен цикл? Зная размер файла, мы читаем один байт из входного файла и записываем этот один байт в выходной файл. Наверное, было бы лучше прочитать весь входной файл сразу целиком, и сразу записать всё в выходной файл. Прочитать целиком – это хорошо, только куда? Указатель тут приходится как нельзя более кстати. Вот этот текст из первой версии программы изымается:

```
for i:=1 to SR.size do begin
  BlockRead( infile, b, 1);
  BlockWrite( outfile, b, 1);
end;
```

И заменяется на такой:

```
GetMem( p, SR.size);
BlockRead( infile, p^, SR.size);
BlockWrite( outfile, p^, SR.size);
FreeMem( p, SR.size);
```

Переводим с паскалевского на русский:

Первая строка – выделена память в количестве SR.size байт. Адрес выделенной памяти содержится в указателе P. Вторая строка – прочитано такое же количество байтов из входного файла. Прочитано по адресу, на который указывает указатель P, то есть P^. Третья строка – аналогично, но теперь данные записаны в выходной файл. Четвёртая, и последняя – память освобождена.

О технике безопасности. Выделенная память должна быть освобождена, и в точности в том же количестве. Сначала выделить, потом освободить. Не перепутайте. Нельзя два раза выделять память по одному указателю, не освобождая её перед вторым выделением. Внешне всё будет хорошо, а на самом деле плохо. Называется это утечка памяти. Несколько таких утечек и, когда в следующий раз вы попросите ещё немного памяти, вы её не получите. Вся кончилась. Утекла, в смысле.

Нельзя два раза освобождать память по одному указателю. Вот здесь - `BlockRead( infile, p^, SR.size);` - должно быть именно `P^`. Если написать просто `P`, транслятор скушает. `BlockRead` отработает, ему всё равно. О последствиях даже говорить не буду, настолько они будут ужасны. А почему я уточнил, что копировать мы будем небольшой файл? А потому, что в Турбо Паскале одному указателю можно выделить не более 64К памяти, точнее даже чуть меньше.

Итак, трудности и непонятности стоят во весь рост, польза где-то прячется. Но это так только кажется с первого взгляда.

### Round, Ord, Chr и другие пустячки

Вещицы разные и мелкие. О некоторых уже упоминалось, о некоторых нет.

`Round`. Как следует из названия – округляет. Округляет до целого. Позволяет присвоить целой переменной дробную переменную. Ну не совсем дробную, а целую часть от неё. Понятно, в общем.

```
var
    int           : integer;
    fl            : single;

fl:=1.23;
int:=Round(fl);
```

В результате целая переменная будет равна единице.

`Ord` и `Chr`. Возможно, у разработчиков были какие-то возвышенные намерения в отношении этих функций. Изначально они предназначены для работы со всеми перечисляемыми типами. В реальности их используют только при работе с символами, причём в какой-то полухакерской манере. Систематизируем ранее сказанное и ранее только упомянутое. Сначала повторим тип `Char`.

`Char`. Один символ. Переменная строкового типа – по сути массив, каждый элемент этого массива – `char`. `Char` занимает один байт, но для наших целей это неважно (пока). Каждый символ имеет свой код в диапазоне от 0 до 255. Например, большая английская буква `A` имеет код 65, цифра `0` – код 48. Более того, внутри символа этот самый код и находится. Если записать в файл переменную типа `char`, например, ноль

(символьный), а прочитать это из файла в переменную типа байт, то в этой переменной окажется значение 48.

Вот для облегчения подобных манипуляций и используются эти две функции. Ord возвращает код символа. Если написать `int:=Ord('A')`; целая переменная получит значение, как легко догадаться, равное 65. По сути функция переводит символ в байт. Chr выполняет обратную операцию – переводит байт в символ. Пишем `ch:=Chr(65)`. Символьная переменная получает значение A. По сути, обе функции вообще ничего не делают. Как было значение в байте 65, так и осталось. Мы только определяем, что это такое – символьная переменная или однобайтовое целое. Тяжёлое наследие разработки Паскаля как учебного языка со строгим типированием.

А вот пример относительно мирного и полезного применения. Задача – проверить, состоит ли строка только из цифр. Очевидный вариант:

```
onlyNumbers:=true;
for i:=1 to Length(s) do begin
    if (s[i]<>'1') and (s[i]<>'2') and (s[i]<>'3') and
       (s[i]<>'4') and (s[i]<>'5') and (s[i]<>'6') and
       (s[i]<>'7') and (s[i]<>'8') and (s[i]<>'9') and
       (s[i]<>'0')
    then onlyNumbers:=false;
end;
```

Вариант покороче использует тот факт, что цифры в таблице кодировки расположены подряд, начиная с 48(0) и кончая 57(9).

```
onlyNumbers:=true;
for i:=1 to Length(s) do begin
    if not ((Ord(s[i]) >= 48) and (Ord(s[i]) <= 57))
    then onlyNumbers:=false;
end;
```

Я не считаю такой подход очень уж правильным, ведь мы навсегда попадаем в рабство принятой таблице кодировки. Тем не менее это работает, и, по крайней мере в чужих программах, вы будете нередко встречать подобное.

Ещё две полезные (действительно полезные!) процедуры. Были бы функциями – цены бы им не было. Они похожи на только что

прокомментированные. Только Chr и Ord преобразовывают символ в число и обратно, а эти две функции работают не с отдельным символом, а с целой строкой.

```
var
  s1, s2      : string;
  int         : integer;
  fl          : single;
begin
  int:=10;
  fl:=12.345;
  Str( int:3, s1);
  Str( fl:5:2, s2);
  writeln(s1);
  writeln(s2);
```

Получим

```
10
12.34
```

Число преобразуется в своё символьное представление. В качестве первого параметра может использоваться выражение или константа. Числа после первого параметра несут тот же смысл, что и в операторе writeln. Второй параметр всегда переменная. Одно из применений – вывод чисел на экран в графическом режиме, ведь там нет оператора writeln.

Например, так. Полностью, с объявлениями.

```
var
  stroka      : string;
  x           : single;
begin
  x:=3.14;
  {а можно вот так}
  x:=Pi;
  Str(x:8:3, stroka);
  SetColor(Green);
  SetTextStyle( 1, HorizDir, 5);
  OutTextXY( stroka, 100,100);
```

Процедура Val устроена посложнее.

```
var
```

```

s1,s2      : string;
fl         : single;
int        : integer;
code       : integer;

s1:='123.45';
Val( s1, fl, code);
if code = 0 then begin
    {что-то делаем}
end
else begin
    writeln('bad');
end;

s1:='123.45zzzzzz';
Val( s1, fl, code);
if code = 0 then begin
    {что-то делаем}
end
else begin
    writeln('very bad');
end;

```

Val выполняет обратную операцию – преобразовывает строку (первый параметр) в целое или дробное число (второй параметр). Тонкая разница в том, что число в строку можно перевести всегда, а строку в число – отнюдь нет. Для того и нужен третий параметр. Если он равен нулю, значит, число перевелось успешно, если не ноль – всё пропало. Точнее, ненулевой третий параметр указывает на номер символа строки, в котором возникли проблемы – но кому оно надо?

Очевидное применение – ввести число с экрана в графическом режиме. Упражнение – введите число с экрана в графическом режиме. В каком, собственно, смысле – ввести? В том смысле, что вы у нас будете вместо подсистемы компьютера, за текстовый ввод отвечающей. Определяем и запоминаем текущее положение курсора. Рисуем курсор. Далее – цикл до нажатия клавиши Enter. Если нажато что-то отображаемое – буква, цифра – нарисовать в текущей позиции, курсор переместить на одно знакоместо вправо - стереть в текущей позиции, увеличить позицию по горизонтали на единицу, нарисовать в новой позиции. Если нажали забой (BackSpace) – стереть последний введённый символ на экране и в памяти, переместить курсор на одну позицию влево. А если наконец Enter – использовать Val и перевести в число. Разумеется, если что не так и опять набрали какую-то фигню, громко кричать и возмущаться.

Это ничего, что я употребляю страшные слова вроде *знакоместо*?

### Есть такая штука – множество

Красивый безобидный пустячок. Не сравнить с указателями и рекурсией. Знаете ли Вы, что такое множество? Если нет, то отползайте, отползайте... Вам не надо. Если знаете, то продолжаем. Множества в Паскале почти как настоящие. Только элементов у них не более 256 – прописью - двухсот пятидесяти шести. И элементом такого множества может быть только тип, принимающий не более 256 – прописью - двухсот пятидесяти шести значений. И ещё кой-какие ограничения.

Символ (char) проходит по всем критериям. Объявляем множество символов:

```
var
  s                : string;
  dgs              : set of char;
  onlyNumbers     : boolean;
  i                : integer;

  {справа от оператора присваивания – пустое множество}
  dgs:=[];

  {а теперь заталкиваем туда циферки}
  dgs:=['1', '2', '3', '4', '5'];

  {а теперь суммируем множества}
  dgs:=dgs + ['6', '7', '8', '9', '0'];

  {а теперь та же задачка – состоит ли строка из цифр?}
  onlyNumbers:=true;
  for i:=1 to length(s) do begin
    if not (s[i] in dgs)
      then onlyNumbers:=false;
  end;
```

Пустое множество и суммирование множеств здесь не нужны и добавлены чисто из педагогических соображений. Суммировать можно только однотипные множества. Нельзя сложить множество дней недели и множество целых чисел. **in** проверяет, принадлежит ли элемент множеству. Если принадлежит, то выражение **a in b** является, как нетрудно догадаться, истинным. А если нет, то нет. Обратите внимание, слева от **in** элемент множества без квадратных скобок, справа – множество как переменная или непосредственно заданное в квадратных

скобках. Такой способ определения, состоит ли строка из цифр, кажется мне наиболее симпатичным.

### Совсем глупость – про музыку

Сначала, как выражаются в определённых кругах, disclaimer. По-русски это значит, что фирма Микрософт не за что не отвечает. Фирма Борланд, впрочем, тоже. А я чем хуже? В смысле, чем лучше? Музыка я не учился, музыкального слуха у меня нету, в музыкальной терминологии я путаюсь. Так что просьба скрипачам, пианистам и прочим бездельникам – не стрелять в программиста. Программист играет как умеет.

В списке желательных умений программиста, перечисленных в начале книги, нотная грамотность не значится. Но сейчас я ожидаю, что вы знаете как ноты записываются на линейках. Больше от вас ничего знать не потребуется.

Итак, музыка в Турбо Паскале есть. Правда, исполняется она на встроенном динамике компьютера, но всё же исполняется. Так что проверьте, что динамик у Вас вообще подключен и чего-то пищит - и вперед! Сначала техническая подробность. Для хоть какой-то музыки нужно управление высотой звука и его, звука, длительностью. За длительность у нас отвечает процедура Delay, не очень хорошо работающая, мягко говоря. Придётся написать свою такую же. Собственно, мы уже писали, но тогда она обеспечивала задержку фиксированной, неизменной длительности. Для наших нынешних целей хотелось бы задержку переменной длины. Примерно так:

```
procedure OurDelay(      howmany : single);
  var
    i,j                    : longint;
begin
  for i:=1 to Round(howmany) do
    for j:=1 to 100000 do;
end;
```

В первоначальном Delay параметр это время задержки в миллисекундах. В нашем, в первом приближении, где-то как-то тоже что-то очень отдалённо похожее. Можете – и даже обязаны - константу подкрутить под скорость вашего компьютера, но это не принципиально важно. Главное для нас не абсолютная точность измерений, а чтобы

OurDelay(1000) был в два раза длиннее, чем OurDelay(500). А с этим у нас всё хорошо, даже лучше чем в казённом Delay. А если константы не те – вместо *Largo* получите *Presto*. Если вы, конечно, вообще понимаете, о чём я говорю.

Обратите внимание, параметр у нас дробного типа, и внутри процедуры перед употреблением его приходится округлять. А зачем такие сложности? Чуть позже увидите.

А теперь, вот она, музыка. Убеждаемся, что в секции `uses` прописан модуль `Crt`, затем пишем:

```
Sound(1000);
```

Запускаем. Если с динамиком всё в порядке, наслаждаемся мелодичным звуком высотой 1000Hz. Нравится? То-то же. Теперь учимся звук выключать:

```
Sound(1000);  
NoSound;
```

И чего? А ничего! Звук включили, звук выключили, причём тут же, немедленно. Добавляем свежеизготовленную задержку:

```
Sound(1000);  
OurDelay(500);  
NoSound;
```

Теперь имеем тысячегерцовый звук длиной в полсекунды. Можно приступать к изготовлению реальной музыки. Приступим сразу, безо всякой подготовки, а в процессе поглядим. Берём совершенно настоящие ноты. Вот:



Что видим?

Вообще-то, в высшем смысле, видим мы русскую народную песню *Вдоль да по речке*. Выбрали мы её ввиду полной незащищённости автора от нарушения его, автора, авторских прав. Ввиду полной его, автора, неизвестности. Сейчас будем нарушать, в смысле аранжировать для компьютерного динамика.

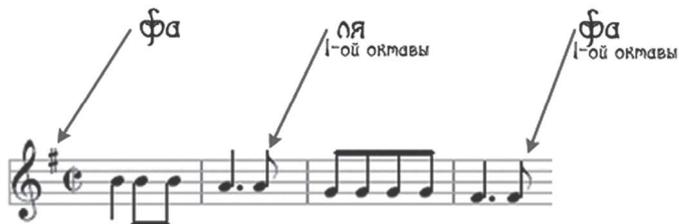
Но пока конкретно видим мы ноты. Нарисованы они на разных линейках, а также над, под и между. Хотя мы этого и не видим, но знаем, что у нот есть названия – до, ре и прочие ля. А то и какой-нибудь ля диэз. Мы, со своей стороны (компьютерной), можем предложить взамен звук определённой частоты, измеряемой в герцах. Надо связать между собой их ля и наши герцы. Из околomuзыкальных источников получаем вот такую табличку:

До	C	262 герц
До диэз	C#	278 герц
Ре	D	294 герц
Ре диэз	D#	311 герц
Ми	E	330 герц
Фа	F	349 герц
Фа диэз	F#	368 герц
Соль	G	392 герц
Соль диэз	G#	415 герц
Ля	A	440 герц
Ля диэз	B	465 герц
Си	H	494 герц

Название ноты, традиционное буквенное обозначение, частота в герцах.

Нот формально семь, но реально двенадцать. Между основными семью, имеющими персональные имена, кое-где напихано ещё пять. Нота, которая выше ре, но ниже ми, называется ре диэз. Музыканты, конечно, утверждают, что всё не так, но утверждают это только ради того, чтобы запутать программистов. Для наших целей наша трактовка самая подходящая. К сожалению, двенадцати нот музыкантам мало. Табличка относится к нотам так называемой второй октавы. Если надо сыграть что-то повыше, сверху размещается третья октава. Названия нот абсолютно те же самые, а частоты в герцах ровно в два раза больше. То есть ля второй

октавы это 440 герц, ля третьей октавы – 880 герц. Снизу находится первая октава. Там частоты всех нот ровно в два раза меньше, то есть ля первой октавы равно 220 герц. Теперь свяжем всю эту абстракцию с нашими реальными нотами.



Нота между второй и третьей линейкой – ля первой октавы, как и написано. Следующая нота, на третьей линейке, стало быть, ми первой октавы. Диез после ключа нарисован на месте фа первой октавы, но великий диезный смысл в том, что действует он на все ноты фа всех октав, повышая их на полтона и превращая в фа диез. Дальше сами увидите.

Теперь, вооружённые всей суммой знаний, записываем нашу песенку нотами, но словесно. Мы же не музыканты, нам программу надо. А словесно – это уже почти программно. Записывать так: A1# - значит ля диез первой октавы. Замечаем, что все ноты относятся к первой октаве.

Кстати, слова в этих двух строчках вот такие:

*Вдоль да по речке, вдоль да по Казанке, сизый селезень плывёт.  
Вдоль да по бережку, бережку крутому, добрый молодец идёт.*

Это для контроля. Если слова и слоги на ноты как-то укладываются, хотя бы по количеству совпадают – это уже хорошо.

Записали ноты словесно? Вот и славно.

H1 H1 H1 /A1 A1 /G1 G1 G1 G1 /F1# F1# /E1 E1 /D1 E1 F1 /G1 G1 /G1/

H1 H1 H1 /A1 A1 A1 /G1 A1 G1 A1/F1# F1# /E1 E1 /D1 E1 F1#/ G1 G1 /G1/

Главное, следите внимательно, чтобы я где-нибудь чего-нибудь не переврал.

Комментарии. Такты разделены только для удобства чтения, в дальнейшем вся информация о них будет утрачена. Фа диез вместо ожидаемого фа – следствие одинокого диеза после скрипичного ключа. Все фа стали на полтона выше. Кстати, диез на ноте фа стоит, а тональность называется соль мажор. Хотя, с другой стороны, Муму Тургенев написал, а памятник почему-то Пушкину поставили.

Но радоваться рановато. Опять мучительно всматриваемся в ноты и замечаем, что они не только сидят на разных линейках, но в придачу и сами по себе какие-то разные – черные, белые, с палочками, с крючочками... Так обозначаются длительности нот. В отличие от высоты – раз сказано 440Hz, так оно и будет – длительность понятие относительное. Та нота, которая с крючочком, длится в два раза больше той ноты, которая с двумя крючочками, вот и всё. Для справки картинка:



Хотя ещё не всё. Видите в нашей песне точки после некоторых нот? Их, нот, длительность увеличивается в полтора раза. То есть без точки она  $\frac{1}{4}$ , а с точкой уже  $\frac{1}{4} + \frac{1}{8} = \frac{3}{8}$ .

Снова переписываем песню словами, но теперь уже с учётом длительностей. Длительности в скобках, они у нас будут параметрами. Только мы будем писать не  $\frac{1}{4}$ , а просто 4. Означать это будет, что при исполнении ноты мы устанавливаем задержку в длительность целой ноты, делённой на 4. Нам, программистам, так удобнее. Следуя этой логике,  $\frac{3}{8}$  закономерно превращаются в  $\frac{8}{3}$ . Не пугайтесь.

H1(4) H1(8) H1(8) /A1(8/3) A1(8) /G1(8) G1(8) G1(8) /F1#(8/3) F1#(8)

E1(4) E1(4) /D1(4) E1(8) F1(8) /G1(4) G1(4) /G1(2)/

H1(4) H1(8) H1(8) /A1(4) A1(8) A1(8) /G1(8) A1(8) G1(8) A1(8)/

F1#(8/3) F1#(8) /E1(4) E1(4) /D1(4) E1(8) F1#(8)/ G1(4) G1(4) /G1(2)/

А теперь наконец-то попрограммируем! У нас есть запись песни в виде последовательности нот – напомним для каждой ноты по процедуре, что может быть проще? Или написать одну процедуру, а ноту передавать как параметр? А какая, собственно, разница. Поразмышляйте, и придёте к тому же выводу. Одна из причин такого моего решения – а как передавать в универсальную процедуру играемую ноту? Как параметр какого типа? Строка? Именованная константа целого типа? Пользовательский тип? Рассмотрите эти варианты и подумайте.

Другая причина – наглядность. Наглядность – дело во многом субъективное, но всё же. С точки зрения моей персональной субъективности запись исполнения до-ре-ми в виде

```
c(8);  
d(8);  
e(8)
```

предпочтительнее чем

```
pl(c,8);  
pl(d,8);  
pl(e,8);
```

Сколько раз повторялось, что использование глобальных переменных – преступление? Вот-вот. А сейчас мы их того... используем... Поймите меня правильно, я от своих принципов не отказываюсь. Но иногда отступаю. А куда деваться, в этом случае? Как уже сказано, длительность – понятие относительное. Недаром в нотах обычно задаётся темп, словесно или численно. Хотелось бы иметь возможность написать вот так:

```
one:=1000;
```

То есть мы указали, что целой ноте соответствует длительность в тысячу миллисекунд. А как передать значение *one* в процедуру для ноты? Как параметр? Была бы процедура одна, а их очень много. Очень мусорный текст получается. Так что придётся поступиться принципами. В результате, для многострадальной ноты ля второй октавы (она же первая струна на пятом ладу) пишем:

```
procedure a(      t : single);
  var
    ti          : single;
begin
  if t > 0 then begin
    ti:=one / t;
    Sound(440);
    OurDelay(t);
    NoSound;
  end;
end;
```

На что обратить внимание. В процедуру передаётся длительность не в миллисекундах, а в музыкальных понятиях – половинная, четвёртая, восьмая. Пересчитываем в миллисекунды уже внутри, с использованием пресловутой переменной *one*. Условный оператор – защита от идиота, передавшего ноль. Не сомневайтесь, идиот найдётся.

Дальше, по идее, надо программировать ля диез, но чем он отличается? Только высотой. А не вынести ли нам всю внутренность нашей процедуры в процедуру отдельную, а уже ей передавать заказанную высоту и длительность? Очень похоже на предлагавшуюся раньше универсальную процедуру, только теперь она зарыта глубоко внутри нашей иерархии процедур. Конечному пользователю, который будет записывать музыку, процедура уже не видна. Получается вот так:

```
procedure MS(      hz : integer;
                  t   : single);
  var
    ti          : single;
begin
  if t > 0 then begin
    ti:=one / t;
    Sound(hz);
    OurDelay(t);
    NoSound;
  end;
end;
```

```

end;

procedure a(      t : single);
begin
    MS( 440, t);
end;

```

MS это не Microsoft, а MakeSound.

Самостоятельно программируем пару-тройку октав (про запас). Для нетрудолюбивых текст модуля в приложении. Что ещё осталось? Когда музыканту надоедает пикивать по скрипочке, он отдыхает. Не играет, в смысле. Это называется пауза. В зависимости от тунеядистости пикикальщика паузы классифицируются в точности так же, как и ноты – половинная, четвёртая. В модуле они предусмотрены, хотя сейчас нам и не понадобятся.

Теперь опробуем. Напишем кусок мелодии:

```

h1(4); h1(8); h1(8); a1(8/3); a1(8);
g1(8); g1(8); g1(8); g1(8); f1d(8/3); f1d(8);

```

Исполним. Звучит. Но звучит как-то того – *legato*, что ли. В смысле всё слиплось и одна нота плавно переходит в другую. Если это то, о чём вы мечтали, то ладно, а иначе – решаем проблему. Сделаем чуток *staccato*, если вы опять-таки понимаете, о чём я говорю. Пусть нота звучит не сколько надо, а чуть меньше. От длительности ноты откусывается одна восьмая. Семь восьмых нота звучит как надо, последнюю восьмую не звучит. Константу можно поменять для большей стаккатиности. Бывают случаи, когда в нотах явно обозначено связанное исполнение. Этот вариант в модуле тоже предусмотрен. Всякие большие фантазии вроде *триолей* мы игнорируем.

Вот наша песня в финальном варианте:

```

one:=3000;

h1(4); h1(8); h1(8); a1(8/3); a1(8);
g1(8); g1(8); g1(8); g1(8); f1d(8/3); f1d(8);
e1(4); e1(4); d1(4); e1(4); f1d(4);
g1(4); g1(4); g1(2);

h1(4); h1(8); h1(8); a1(4); a1(8); a1(8);

```

```
g1(8); a1(8); g1(8); a1(8); f1d(8/3); f1d(8);  
e1(4); e1(4); d1(4); e1(8); f1d(8);  
g1(4); g1(4); g1(2);
```

Я прочел книгу и многое понял. Понял, откуда вообще берутся ноты, каждая со своей конкретной частотой, что такое гамма и что такое октава. Главное, книга эта не для музыкантов, а для математиков. И вам очень советую:

*Популярные лекции по математике, выпуск 37*

*Г.Е.Шилев Простая гамма. Устройство музыкальной шкалы*

*Государственное издательство физико-математической литературы  
М, 1963*

### Оно надо? Рекурсия.

*Настоящее произведение искусства, впрочем, совершенно бесполезное* – записал в своём дневнике начальник немецкого Генерального штаба сухопутных войск генерал-полковник Франц Гальдер. Речь шла о гигантской восьмисотмиллиметровой пушке Дора, изваянной сумрачным германским гением. С рекурсией та же фигня. Красиво, но бесполезно. Но красиво. Но бесполезно. Хотите написать – напишите. Но помните, что немцы кончили хреново.

Что такое рекурсия? Это очень просто. Это когда процедура вызывает сама себя. Или, понятное дело, функция вызывает сама себя. Как обычно, демонстрируем рекурсию на совершенно бессмысленном примере.

```
procedure Recurs;  
begin  
  Recurs;  
end;
```

Написали. Вызвали. Да, не очень хорошо. Модернизируем.

```
procedure Recurs;  
begin  
  writeln('Muche trabajo!');  
  Recurs;  
end;
```

В общем-то, немногим лучше. Но хоть видно, что работает. Вся проблема с рекурсией в том, что когда-нибудь она должна кончиться. Нет, не в том смысле, что всё когда-нибудь кончается. Сама по себе рекурсия не кончается. Это мы должны обеспечить её конец.

Всегда и везде, если речь заходит о рекурсии, её иллюстрируют на примере вычисления факториала. А мы что, не люди, что ли? И мы начнём с факториала. Напоминаю (какой-то уже раз): факториал от  $N$  – это произведение всех целых чисел от единицы до  $N$ . Обозначается  $N!$ . Факториал мы уже считали, вот таким образом:

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

Применяем рекурсию. Сначала, как принято, думаем. Что такое факториал от 5?  $5! = 1*2*3*4*5$ . А что такое  $1*2*3*4$ ? Правильно, это факториал от 4. То есть  $5! = 4!*5$ . Очевидно, что эта забавная особенность присуща не только пятерке.  $N!=(N-1)!*N$ . И вот тут вползает рекурсия.

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

Имя нашей функции появилось в правой части оператора присваивания. Вот оно, зримое проявление рекурсии! А главное здесь, что рекурсия не вечна. При  $N$ , равном единице рекурсия завершается. Обратите внимание, раньше был цикл. А теперь цикла нет, но есть рекурсия. Рекурсия, по сути, замаскированный цикл. И обычно её можно циклом заменить. Но иногда сделать это не то, что трудно, но коряво и неудобно.

Чуть-чуть технических подробностей. Подробность первая – теоретическая. Есть процедура А, она вызывает процедуру А, саму себя, в смысле. Именно это мы сейчас и наблюдали. Это называется *прямая рекурсия*. Но если есть процедура А, она вызывает процедуру В, а процедура В вызывает, в свою очередь, процедуру А, то это называется *косвенная рекурсия*.

Теперь подробность практическая. У нас в процедуре объявлена всего одна переменная, занимающая жалких четыре байта. Повезло. Переменных могло быть больше и размер у них мог быть потолще. Чем это чревато? Зайдите в пункт меню <Options>\<Memory Sizes>. Посмотрите на пункт Stack Size. Что вы там видите? Скорее всего, число соответствующее шестнадцати килобайтам. Турбо Паскаль, и не только он, делит всю доступную память на две части – кучу (Heap) и стек (Stack). Кучу оставим пока в стороне. На что используется стек? Переменные, объявленные внутри процедуры (локальные переменные), размещаются в стеке. В исходной версии нашей процедуры расчёта факториала было объявлено две переменных, общим объёмом шесть байт. При входе в процедуру из стека забираются эти шесть байт, при выходе из процедуры шесть байт освобождаются и возвращаются в стек. Из чего попутно следует, что значения локальных переменных забываются при выходе из процедуры – не только рекурсивной.

В рекурсивной версии нашей процедуры только одна переменная размером четыре байта. Но теперь процедура не завершается а, скорее всего, вызывает сама себя. И эти четыре байта выделяются снова. Если мы захотим посчитать факториал от тысячи, то потратим из стека тысячу, умноженную на четыре, байтов.

На самом деле, всё немного хуже. Передача параметров в процедуру и возвращение значения функции тоже используют стек. Так что память в стеке может закончиться даже быстрее, чем планировалось. Что делать? Экономить стек. Или увеличить его размер – в том самом пункте меню. Или не применять рекурсию.

А теперь ещё одна классическая задача на рекурсию. Реализуете её самостоятельно. Называется – рекурсивный обход дерева.

Дерево, которое мы собираемся обходить – дерево каталогов. Мы знаем, как вывести оглавление каталога. Но только одного. А если в этом каталоге есть подкаталоги? Например, в каталоге `c:\bpcascal` имеются подкаталоги `bin`, `units` и `bgi`. И мы хотим получить оглавление не только основного каталога `bpcascal`, но ещё узнать, какие файлы содержатся в подкаталогах. Разумеется, подкаталоги эти могут содержать свои собственные подкаталоги, которые тоже могут... и так далее.

Новое знание нам потребуется только одно – команда перехода в другой каталог. Называется она `ChDir`. Параметр у неё один, строкового типа. Значение параметра – каталог, в который мы хотим перейти. Имя каталога может содержать и символ, обозначающий диск. В этом случае сменится не только каталог, но и диск. Но для нашей задачи это лишнее.

Как будем действовать? С помощью `FindFirst` и `FindNext` получаем и выводим список файлов, содержащихся в текущем каталоге. В списке этом будут и каталоги, так что для каждого файла проверяем – а не каталог ли он случайно? Сделать это можно, проанализировав соответствующее поле записи, описывающей файл. Если у нас есть объявление

```
var
    SR                : SearchRec;
```

то этим полем будет `SR.Attr`. Отдельные биты этого поля отвечают за отдельные атрибуты файла. Один из битов, если не ошибаюсь пятый, установлен в единицу тогда и только тогда, когда наш файл является каталогом. Какой именно по номеру бит я, разумеется, точно не помню, потому что на это имеется соответствующая константа. Пользуются этой константой вот так:

```
if (SR.Attr and Directory)
    then writeln('Каталог. Радость-то какая!');
```

Это называется побитовое сложение, но для нас это сейчас не важно – мы тут рекурсией занимаемся, а не чем-то ещё. Итого – мы умеем переходить в другой каталог и умеем узнать, является ли найденный файл каталогом. Пишем процедуру с одним параметром. Параметр – имя каталога, оглавление которого, включая все его подкаталоги, нам надо вывести. В процедуре – сначала переходим в заданный каталог, затем в цикле

выводим (куда-то) список файлов. Если файл вдруг неожиданно оказывается каталогом, вызываем для него нашу процедуру. Вот и рекурсия! Собственно, и всё. Приступайте к реализации.

### Меряем время

Предмет, рассматриваемый в данном разделе, прост, как не знаю что. Узнать который час. И всё. Но для запутывания интриги, сначала узнаем, какой сегодня день. Зачем? Вы и так знаете? У вас и справка есть? Ну мало ли что...

```
var
    year, month, day, dayofweek    : word;

    GetDate( year, month, day, dayofweek);
```

Для изучавших французский, перевожу. Четыре переменные – год, месяц, день, день недели, в смысле, номер дня недели. Из особой щепетильности переменные объявлены как word. Понятно, что отрицательными они быть по смыслу ну никак не могут, но всё равно раздражает. Потому что забываешь про этот word.

Вроде бы всё кристально ясно и прозрачно. Теперь о времени. Сходство чрезвычайное.

```
var
    hour, minute, second, sec100  : word;

    GetTime( hour, minute, second, sec100);
```

Первые три параметра – часы, минуты, секунды. Четвёртый по уму должен был бы быть миллисекундами, но оказался сотыми долями секунды. Можно предположить, что разработчиков мучила совесть и они не хотели вводить в заблуждение программистов – измерение времени здесь очень неточное, и ни о каких миллисекундах речи не идёт. С другой стороны, измерение настолько неточное, что и десятки миллисекунд под большим вопросом, а неудобство с сотыми долями налицо.

Теперь давайте не просто узнаем, который час, а померяем интервал времени. Это будет чуть сложнее. Чтобы мерить что-то осмысленное и с пользой, предлагаю взять две процедуры определения факториала – обычную с циклом и рекурсивную – и померить, кто быстрее.

Какие проблемы и проблемки ждут нас на этом пути? Процедуры нужно переименовать – не могут же они сосуществовать в одной программе с одинаковыми именами. Если попробовать определить факториал хотя бы десяти, мы получим вот такое сообщение об ошибке:

Error 201: Range check error

Догадайтесь о причине. Вспомните, какое максимальное число помещается в двухбайтовое `integer`. Чтобы несколько расширить наши вычислительные возможности, заменим `integer` на `longint`. Выясните, математически или путём наступания на грабли, факториал какого числа мы теперь можем вычислить.

Затем объявляем четыре переменные для получения и хранения времени. А может быть, восемь? Четыре для времени начала расчета и четыре для времени окончания... А как мы будем считать, сколько времени прошло? Сделать это, оперируя часами, минутами и секундами, задача, безусловно, полезная и развивающая навык работы с условными операторами, но мы сейчас занимаемся немного другим. Моё мнение – оба времени надо перевести в миллисекунды и узнать, сколько времени истекло, простым вычитанием. И вот их-то, миллисекунды в смысле, и запоминать. Объявляем две переменные типа `longint` – время до и время после. Вот что имеем для начала:

```
program factor;
  uses
    Crt, Dos;
  var
    h,m,s,s100           : word;
    time0, time1         : longint;
    N                     : integer;
    rez                  : longint;
    i                    : longint;
  {-----}
  function OldFact(      N : integer) : longint;
  var
    result              : longint;
    i                   : integer;
begin
  result:=1;
  for i:=1 to N do
    result:=result*i;
  OldFact:=result;
```

```

end;
{-----}
function NewFact(      N : integer) : longint;
  var
    result                : longint;
begin
  if N = 1
  then result:=1
  else result:=N*NewFact(N-1);
  NewFact:=result;
end;
{-----}
begin
  ClrScr;
  {-----}  обычный способ  {-----}
  { тут меряем}
  {-----}  рекурсивный способ  {-----}
  {и тут меряем}
  readln;
end.

```

## В чём заключается само измерение?

```

Получить текущее время
Перевести в миллисекунды и запомнить
Посчитать факториал
Получить текущее время
Перевести в миллисекунды и запомнить
Вывести разницу между временами

```

Повторить для способа с рекурсией

Сделайте, попробуйте, посмотрите на результат, подумайте. У меня получилось, что уже для 13! время расчёта является нулевым. А факториал большего числа в longint не вмещается. Можно было бы расширить доступный диапазон, используя какой-нибудь плавающий тип. В этом случае мы, разумеется, получили бы только приближённое значение факториала. Но нас это не спасёт – компьютер всё равно считает слишком быстро.

Другой вариант – посчитать один и тот же факториал не один раз, а десять. Или сто. Лично у меня ненулевые значения появились при миллионе повторений, а программа приобрела вот такой вид:

```

N:=13;
skoka:=1000000;
{-----}  обычный способ  {-----}

```

```

GetTime( h, m, s, s100);
time0:=h*60*60*1000 + m*60*1000 + s*1000 + s100*10;

for i:=1 to skoka do
  rez:=OldFact(N);

GetTime( h, m, s, s100);
time1:=h*60*60*1000 + m*60*1000 + s*1000 + s100*10;

writeln( 'обычный способ. время = ', time1-time0);
{----- рекурсивный способ -----}
GetTime( h, m, s, s100);
time0:=h*60*60*1000 + m*60*1000 + s*1000 + s100*10;

for i:=1 to skoka do
  rez:=NewFact(N);

GetTime( h, m, s, s100);
time1:=h*60*60*1000 + m*60*1000 + s*1000 + s100*10;

writeln( 'рекурсивный способ. время = ', time1-time0);
writeln(N, '!=', rez);

```

Померяйте время. Получите результат. Подумайте. Сделайте выводы.

А теперь хочу, чтобы с графикой. Нарисуйте часы. С тремя стрелками – часовой, минутной и секундной. И чтобы ходили, разумеется. И тикали. Подумаем сначала о часах стоящих. Мы знаем время, в часах, минутах и секундах. Задача - нарисовать часы и три стрелки в соответствующем положении. Сами часы – окружность, Circle. Стрелки – линии. Один конец линии – центр окружности, другой конец – где, собственно? А вот здесь нас подкарауливает математика в лице геометрии и тригонометрии. Теперь поглядим на часы идущие. Стрелки должны двигаться. Когда? Когда изменилось время. Самая быстрая стрелка у нас секундная, значит передвигать стрелки надо, когда изменилось число секунд. Как мы можем узнать, когда изменилось число секунд? Нам доступен только один способ – бесконечно отслеживать, который час (*который час* здесь просто фразеологизм, отслеживать будем секунды). Как только изменилось число секунд – перерисовываем стрелки. Часовую стрелку и минутную мы при этом, как правило, будем перерисовывать совершенно зря – часы и минуты, скорее всего, не изменились – но куда деваться? И вообще, компьютер железный, а мы устали.

В результате имеем что-то такое:

Нарисовать циферблат  
Запомнить время  
Цикл пока не надоест  
    Если время изменилось  
        Стереть стрелки  
        Нарисовать стрелки  
        Запомнить время

Теперь обещанные геометрия с тригонометрией. Определяем константы, все целые. Центр циферблата –  $cX$ ,  $cY$ . Длины часовой, минутной, секундной стрелок –  $rh$ ,  $rm$ ,  $rs$ . Всё измеряется в экранных точках, естественно. Чтобы нарисовать стрелку, надо провести линию. Один конец линии – центр циферблата, он константа, другой конец линии надо рассчитать. Вот формулы, для случая, когда центр циферблата совпадает с началом координат:

$$X = R * \sin(\alpha)$$
$$Y = R * \cos(\alpha).$$

$R$  – это, понятно, радиус, то есть длина стрелки.  $\alpha$  – угол между вертикалью и нашей стрелкой. С углом имеем некоторые нюансы. Начнём с секундной стрелки. В любой момент нам известен не угол, а время в секундах. Соображаем, что когда стрелка указывает на цифру три, иными словами прошло пятнадцать секунд, стрелка образует прямой угол с вертикалью. То есть, пятнадцать секунд соответствуют девяноста градусам. Одна секунда – пятнадцать градусов. Умножаем секунды на пятнадцать, получаем угол в градусах. Это ничего, что я так неторопливо объясняю? Угол, как верно подмечено, в градусах. Но функция  $\sin$  на входе ждёт только углы в радианах. Вспоминаем, что  $2\pi$  радиан это 180 градусов. Добавляем в константы коэффициент пересчёта

```
coef = 3.14159/180;
```

и записываем свежеиспечённую формула для расчёта координат конца стрелки

```
x:=cX + Round(rs*sin(6*s*coef));  
y:=cY - Round(rs*cos(6*s*coef));
```

$S$  – количество секунд. Координаты центра ( $cX, cY$ ) мы учли. Подумайте, почему в одном случае плюс, а в другом минус. С минутной стрелкой получаем абсолютно тоже самое, с заменой секунд на минуты:

```
x:=cX + Round(rm*Sin(6*m*koef));
y:=cY - Round(rm*Cos(6*m*koef));
```

С часовой придётся подумать, но совсем немного. Во-первых, девяносто градусов соответствуют трём часам, то есть коэффициент пересчёта будет равен тридцати. Во-вторых, что три часа, что пятнадцать, циферблат у нас только один на двенадцать часов. Учитывая вышесказанное, получаем:

```
if h>12
  then h:=h-12;
x:=cX + Round(rh*Sin(30*h*koef));
y:=cY - Round(rh*Cos(30*h*koef));
```

А теперь все вместе и хором! В смысле бодренько дописываем ещё недописанное, и получаем вот такую программу:

```
program Clock;
uses
  Crt, Graph, Dos;
const
  cX=320;
  cY=240;
  rs=200;
  rm=170;
  rh=120;
  koef=3.14159/180;
var
  mode, driver           : integer;
  m, h, s, s100         : word;
  mOld, hOld, sOld      : word;
  x,y                   : integer;
  i,j                   : integer;
begin
  driver:=VGA; mode:=VGAHi;
  InitGraph( driver, mode, '');

  SetLineStyle( SolidLn, 0, ThickWidth);
  SetColor(White);
  Circle(cX,cY, rs+5);

  sOld:=0;

  repeat
    GetTime(h,m,s,s100);
    if s <> sOld then begin
      { seconds }
      SetLineStyle( SolidLn, 0, NormWidth);
      SetColor(Black);
```

```

x:=cX + Round(rs*Sin(6*sOld*koef));
y:=cY - Round(rs*Cos(6*sOld*koef));
Line(cX,cY, x,y);
SetColor(Red);
x:=cX + Round(rs*Sin(6*s*koef));
y:=cY - Round(rs*Cos(6*s*koef));
Line(cX,cY, x,y);
sOld:=s;

{ minutes }
SetLineStyle( SolidLn, 0, ThickWidth);
SetColor(Black);
x:=cX + Round(rm*Sin(6*mOld*koef));
y:=cY - Round(rm*Cos(6*mOld*koef));
Line(cX,cY, x,y);
SetColor(Yellow);
x:=cX + Round(rm*Sin(6*m*koef));
y:=cY - Round(rm*Cos(6*m*koef));
Line(cX,cY, x,y);
mOld:=m;

{ hours }
if h>12
  then h:=h-12;
SetLineStyle( SolidLn, 0, ThickWidth);
SetColor(Black);
x:=cX + Round(rh*Sin(30*hOld*koef));
y:=cY - Round(rh*Cos(30*hOld*koef));
Line(cX,cY, x,y);
SetColor(Green);
x:=cX + Round(rh*Sin(30*h*koef));
y:=cY - Round(rh*Cos(30*h*koef));
Line(cX,cY, x,y);
hOld:=h;
end;
until KeyPressed;

CloseGraph;
end.

```

Проверьте. Добейтесь, чтобы часы тикали – вы ведь освоили звук!

## Страшная сила

А сейчас, дети, мы выучим плохие слова, которые вы не должны произносить ни в коем случае. Или начнём по другому. Они есть! И никуда от этого не деться. Всякие страшные злые вещи. И лучше, если вы узнаете о них от меня. Страшных злых вещей несколько. Нет, их, конечно совершенно немереное количество, это я расскажу только о нескольких.

Начнём с не очень страшного. Задача – найти первый чётный элемент массива. Мы её уже решили, вот так:

```
indexEven:=0;
for i:=1 to N do begin
  if ((a[i] mod 2) = 0) and (indexEven=0)
    then indexEven:=i;
end;
```

Что бросается в глаза? Первый чётный элемент мы уже давно нашли, но продолжаем перебирать массив до конца. При этом мы вынуждены производить дополнительную проверку, чтобы не испортить уже найденный индекс. Некузяво как-то. Хуже того. Если бы мы не догадались дописать проверку на `indexEven = 0`, то получили бы не первый, а последний чётный элемент. Проблема решается волшебным словом `Break`:

```
indexEven:=0;
for i:=1 to N do begin
  if (a[i] mod 2) = 0 then begin
    indexEven:=i;
    Break;
  end;
end;
```

`Break` прекращает выполнение цикла, внутри которого он находится. То есть, если первым чётным элементом оказался третий, то ровно три раза цикл и выполнится. С одной стороны, это хорошо – даже обосновывать не надо, почему. С другой стороны, это хорошо только при правильно организованной программе – возможно, вы попутно делали с массивом что-то ещё и в своей наивности ожидали полного прохода по массиву. И повторю ещё раз, а вы прочтите внимательно - `Break` прекращает выполнение *только* цикла, внутри которого `Break` находится. *Только* этого цикла. Если циклы у нас вложенные, то прекратится выполнение только внутреннего. Если бы мы искали первый чётный элемент двумерного массива, такой приём не сработал бы. Это несколько ограничивает полезность `Break`. Зато `Break` отлично работает и с циклами вида `while` и `repeat-until`. В общем, пользуйтесь на здоровье, но помните...

Следующий экспонат кунсткамеры гораздо противнее. Называется `Continue`. Действует он чуть-чуть наоборот. Цикл выполняется ровно

столько раз, сколько предполагалось изначально. Но вот внутри цикла, всё что находится после Continue выполняться не будет. Иллюстрировать не буду, поскольку считаю эту команду безусловным и законченным злом. По сути, Continue – эмулятор Того, О Чём Нельзя Говорить (в приличном программистском обществе). Я и не буду. Приговор – забыть немедленно.

Теперь два схожих на лицо монстрика из другой экологической ниши. Первый - FillChar. Задумка совершенно безобидная. Заполнить строку одним и тем же символом. Использоваться (по-хорошему) должен вот так:

```
FillChar( s[1], Length(s), 'A');
```

Обратите внимание на s[1] – это не спроста. Функция Length тоже не случайна. Заполнить можно только уже ранее заполненную ранее чем-то строку – длина строки хранится в её нулевом символе, а он в процессе не меняется. Встроенная справка Турбо Паскаля предлагает примерно вот такой вариант заполнения строки с изменением её длины:

```
FillChar( s[1], 80, 'A');  
S[0]:=#80;
```

Щупальца за такое пообрывать! Кстати, не помню, встречалось ли нам уже использование решётки перед числом. Так вот, решётка перед числом переводит его в константу типа char, которая константа содержит символ с кодом равным этому самому числу. Трудно, непонятно? Нет, очень просто. То есть, #65 во всех отношениях абсолютно эквивалентно 'A'. Для чего этот полёт фантазии? Только для того, чтобы обойти принципы строгого типирования данных, изначально заложенные в Паскаль. В данном случае – загнать в определённый байт строки конкретное числовое значение.

Разумеется, FillChar абсолютно безразлично относится к вопросу, что и чем он заполняет. Первый параметр – просто адрес, начиная с которого производится заполнение. Границы переменных игнорируются. Третий параметр вообще может быть переменной символьного или байтового типа или константой со значением в пределах байта. На практике, никто с помощью FillChar строки не заполняет, а используют его совсем по-

другому. Например, имеется двумерный массив `array[1..N,1..M]` of integer, и надо его обнулить. По-хорошему, делается это так:

```
for i:=1 to N do
  for j:=1 to M do
    a[i,j]:=0;
```

А по-плохому вот так:

```
FillChar( a, SizeOf(a), 0);
```

Это очень-очень плохо. Я тоже так делаю, но меня хоть совесть мучает.

Последний номер нашей программы – Move. Самое убойное, что есть в Турбо Паскале. С ним можно сломать что угодно. Происхождение процедуры тянется вглубь веков, куда-то к динозаврикам. Происходит она от одноимённой ассемблерной команды. Три параметра. Первый параметр – адрес откуда. Второй параметр – адрес куда. Третий параметр сколько слать, в байтах. Процедура берёт указанное количество байтов начиная с первого адреса и копирует их по второму адресу. Адрес – имя переменной или указатель на память. Проверок никаких. Следующие четыре варианта абсолютно законны (p1,p2 объявлены как pointer).

```
Move (p1^,p2^,1024);
Move (p1^,p2, 1024);
Move (p1, p2^,1024);
Move (p1, p2, 1024);
```

Смысл имеет только первый. По крайней мере, я так думаю. В норме Move используется только совместно с указателями. Любое другое применение чревато и заставляет задуматься, всё ли в порядке в программе. А в сочетании с указателями вещь чрезвычайно полезная и даже необходимая.

Примеров не будет. Вам ещё рано.

### **Никаких новых слов**

А в этом разделе никаких новых слов не будет. Только по-другому расставим старые. Когда-то, давным-давно, рисовали мы таракана и управляли его движением. Вот так:

```

if sc = ArrowLeft then begin
  x:=x-1;
end;
if sc = ArrowRight then begin
  x:=x+1;
end;
if sc = ArrowUp then begin
  y:=y-1;
end;
if sc = ArrowDown then begin
  y:=y+1;
end;

```

Похожие конструкции встречались нам часто и есть у них одна нехорошая особенность. Нужную клавишу мы уже встретили, нужный выбор сделали, но упорно продолжаем проверять клавиши дальше. Это не очень хорошо. Но, по крайней мере, в нашем случае стрелка может быть нажата только одна. А если условия не являются взаимоисключающими? Как, например при выборе боковой клетки для хода в крестиках-ноликах. Могут быть свободны хоть все четыре. Мы в этом случае всегда выберем последнюю. Нам, конечно, всё равно – в данном конкретном случае. Но иногда - и даже чаще, чем иногда - желательно выбрать первую из допустимых альтернатив.

Как с этим справиться? Вспомнить, что есть ещё и **else**, и, с его помощью отсечь излишние проверки. В нашем случае очевидно так:

```

if sc = ArrowLeft then begin
  x:=x-1;
end
else begin
  if sc = ArrowRight then begin
    x:=x+1;
  end
  else begin
    if sc = ArrowUp then begin
      y:=y-1;
    end
    else begin
      if sc = ArrowDown then begin
        y:=y+1;
      end;
    end;
  end;
end;
end;

```

Очевидно то оно очевидно, но как-то уж очень кудряво. Заставляет задуматься, а это лишнее. И всё время уплзает куда-то вправо. Хотелось бы спрямить. Вспомним, что операторные скобки **begin-end** не являются

необходимыми, если после **else** следует только один оператор – а в нашем случае всё, следующее за **else**, можно рассматривать как один очень длинный условный оператор. Выкидываем только что вставленные **begin**'ы и **end**'ы, поднимаем **else**'ы на строчку выше и имеем:

```
if sc = ArrowLeft then begin
    x:=x-1;
end else
if sc = ArrowRight then begin
    x:=x+1;
end else
if sc = ArrowUp then begin
    y:=y-1;
end else
if sc = ArrowDown then begin
    y:=y+1;
end;
```

Так мне больше нравится.

## Том второй, пять старушек – рупь

О чём вообще речь и зачем вообще нужен том второй? Он нужен потому, что мы всё изучили и вроде как даже – с некоторой неуверенностью – всё усвоили. Но – как показывает печальная практика – никто ничего не усвоил – в лучшем случае. В худшем случае, новые знания наложились на старые знания и в голове образовался невообразимый компот.

Поэтому – надо повторять, повторять и повторять! А главное, не просто повторять, а смотреть на приобретённые знания под другим углом. С другой точки зрения. Поверните систему координат и мир заиграет для вас новыми красками.

*Искажённый микроплёнкой, ГУМ стал маленькой избёнкой,  
И уж вспомнить неприлично, чем предстал театр МХАТ © Высоцкий*

Новых знаний здесь не будет, мы возьмём те же знакомые кирпичики и сложим из них новую неведомую фигию.

### Глава 2-1

#### Ещё раз: простая программа и переменные

##### Повторение пройденного

План действий такой:

Сначала напишем просто Очень Простую Программу.

А потом напишем Очень Простую программу, но с секциями констант, типов, из нескольких модулей, с секциями инициализации и чего ещё только можно придумать. То есть модель Настоящей Большой Программы в масштабе 1:35.

*Дятел-самец, выполненный из железобетона в масштабе 32:1, является наилучшим памятником тестю © Е.Шестаков*

Далее Довольно Крупная Программа из нескольких модулей с активным и повсеместным применением указателей.

Ну а дальше как повезёт.

А теперь Очень Простая Программа. Не знаю, как теперь, а в моём чудном детстве, заполучив трамвайный или троллейбусный или автобусный билет, мы тут же проверяли, не счастливый ли он. Счастливым считался билет, у которого сумма первых трёх цифр номера совпадал с суммой последних трёх цифр номера. Такой билет полагалось сожрать немедленно после выхода из общественного транспорта.

Сейчас номера на билетах тоже шестизначные (я проверял). Напрашивается идея простенькой программки, определяющей является ли введенный номер счастливым. Но кому это надо? Так что мы эту, первую, идею отвергаем и задаемся другим вопросом – а какой процент билетов является счастливым? Это уже интереснее, потому что можно оценить интуитивно, а потом проверить глубину своей интуиции. Так что идём и пишем программу.

Обычно программа состоит из трех частей

- ввод данные
- обработка
- вывести результат, он же обработанные данные

Наш случай немного проще хотя бы потому, что входных данных у нас нет - мы хотим проверить *все* билеты на счастье. Дж.Фокс, автор хорошей книги с унылым названием *Программное обеспечение и его разработка* немного пренебрежительно называет такие программы зубочистками – один раз использовал и выбросил. Ну и ладно, пусть будет так, хотя и обидно.

Делаем заготовку программы, которая вообще ничего не делает. Единственный наш творческий вклад – имя программы

```
program Ticket;  
begin  
end.
```

Ticket – это билет по-английски.

Скунс – это американский хорёк! © Гальцев

Теперь немного думаем – совсем немного – что бы мы могли добавить в программу такого, что само бы напрашивалось. На первом этапе создания программы, главное для нас – как можно меньше думать.

Во-первых, очевидно, что в конце работы наша программа должна выдать результат. А результат её выполнения – процент счастливых билетиков. А раз процент – он, скорее всего, будет дробным – нам нужна переменная типа single. Но процент из воздуха не берётся. Как нас в школе учили, процент это одна величина, поделенная на другую и потом умноженная на сто. Возникает немедленное желание объявить ещё две переменных, но почти сразу приходит понимание, что вторая величина равна в точности одному миллиону – количеству билетов. А вот первую переменную – количество счастливых билетов – всё-таки придётся объявить.

Дальше – мы хотим перебрать все возможные шестизначные билеты – от 000000 до 999999. Нулевого билета не бывает, но такая мелочь нас не волнует. Номер билета, безусловно, целый. И он, судя по всему, будет переменной цикла. Переменные цикла традиционно называются I,J,K,L,M, N. Но мы назовём нашу переменную чуть более изысканно – number. И, ещё раз напоминаю, весь смысл вашей программы заключается в переборе всех шестизначных номеров билетов. Итого имеем:

```
program Ticket;
  var
    number           : integer;
    percent          : single;
    lucky            : integer;
begin
  percent:=0;

  for number:=000000 to 999999 do begin
    end;

    percent:=(lucky/1000000) * 100;
    Writeln( 'percent = ', percent:8:2);
end.
```

Процент мы обнулили в самом начале, на всякий случай, хуже не будет. Переменная цикла, начинающегося от 000000 – бесполезно, однако красиво.

Следующий наш шаг заключается тоже во вполне очевидном – разобрать шестизначный номер на две трехзначных половины, а для начала эти две

половины надо объявить, тоже, естественно, как целые переменные. Вопрос посерьёзнее – а как разобрать номер на две половины? Ответ – с помощью операций **mod** и **div**. Вторым операндом в обоих случаях будет 1000. Но если применить операцию **div**, то в результате мы получим результат от деления на эту самую тысячу, то есть первые три цифры, а если операцию **mod**, то остаток от деления, то есть последние три цифры. Что нам и надо. Затем мы должны получить суммы первых трех и последних трёх цифр – сразу заводим под это дело переменные. Программа всё растёт и растёт и приобретает вот такой вид:

```

program Ticket;
var
  number           : integer;
  numberLeft,
  numberRight     : integer;
  sumLeft, sumRight : integer;
  percent         : single;
  lucky           : integer;
begin
  percent:=0;
  lucky:=0;
  for number:=000000 to 999999 do begin
    {определяем первые и последние половины}
    numberLeft :=number div 1000;
    numberRight:=number mod 1000;

    {а тут мы как-то сосчитали две суммы}

    if sumLeft=sumRight
      then lucky=lucky + 1;
  end;

  percent:=(lucky/1000000) * 100;
  Writeln( 'Percent = ', percent:8:2);
end.

```

Осталась сущая ерунда – посчитать две суммы цифр. Но чтобы их посчитать, нужно сначала разобрать числа на цифры – в каждом числе – три цифры. А значит, надо объявить по три переменных для каждого числа. Поскольку чисел всего два, предлагаю объявить сразу шесть переменных. Так проще, а чисел не так уж и много. А как разобрать трехзначное число на цифры – предлагаю опять-таки самым банальным способом – через **mod** и **div**. Как получить первую и последнюю цифру, мы уже знаем:

```

var
  c1, c2, c3           : integer;

```

```

    c11, c12, c13      : integer;
begin
    c1:=numberLeft div 100;
    c3:=numberLeft mod 100;

    c11:=numberRight div 100;
    c13:=numberRight mod 100;

```

Со второй цифрой чуть сложнее. Размышляя аналогично (или рекурсивно, если вам угодно) – вот если бы получить число, состоящее из двух первых цифр нашего числа, то, применив **mod** мы бы с полпинка получили искомое. А как получить то самое число? Применить операцию **div**. На конкретном примере

```

456 div 10 = 45
45 mod 10 = 5

```

Что и требовалось. Сгребаем все наброски и эскизы в одну кучу, и получаем окончательный текст программы:

```

program Ticket;
var
    number           : integer;
    numberLeft,
    numberRight     : integer;
    sumLeft, sumRight : integer;
    c1, c2, c3      : integer;
    c11, c12, c13   : integer;
    percent         : single;
    lucky           : integer;
begin
    percent:=0;
    lucky:=0;
    for number:=000000 to 999999 do begin
        {определяем первые и последние половины}
        numberLeft :=number div 1000;
        numberRight:=number mod 1000;

        {а тут мы как-то сосчитали две суммы}
        c1:=numberLeft div 100;
        c2:=(numberLeft div 10) mod 10;
        c3:=numberLeft mod 100;
        sumLeft:=c1 + c2 + c3;

        c11:=numberRight div 100;
        c12:=(numberRight div 10) mod 10;
        c13:=numberRight mod 100;
        sumRight:=c11 + c12 + c13;
        if sumLeft=sumRight
            then lucky:=lucky + 1;
    end;
end;

```

```
percent:=(lucky/1000000) * 100;  
Writeln( 'Percent = ', percent:8:2);  
end.
```

Сразу заметим, что если бы номера билетов были не шестизначными, а, к примеру, шестнадцатизначными, то наш подход не покатил бы. Но мы ведь пишем очень простую программу!

## Разбор полётов

*По оглашению приговора мы вылетаем в три окна*

© Лопе де Вега *Собака на сене* в исполнении Джигарханяна.

Разумеется, и само собой, вы уже запустили эту программу. Правда? Не заставляйте меня в вас разочароваться. То есть, считаем, что вы её запустили. Прекрасно, я–то сам её не запускал ни разу. Вот написал её прямо здесь, в тексте книги и всё, поверил в собственную непогрешимость. А напоследок всё-таки решил таки запустить, исправив, самой собой, пару тройку отсутствующих запятых и точек с запятыми.

После чего немедленно обнаруживаются следующие мелкие недостатки

1. Главный недостаток - после запуска программа радостно докладывает, что счастливые билеты составляют 91% от их общего, билетов вообще, числа. Этот, сам по себе, безусловно приятный факт несколько противоречит моему личному жизненному опыту.
2. Второе – после завершения программа вылетает просто в никуда – не хватает Readln в конце – а я ведь я сам вас этому учил. Не хватает также ClrScr в начале – чтобы экран не мусорился результатами предыдущих исполнений. А для вызова ClrScr требуется ссылка в uses на Crt или OpCrt, кому как повезёт.
3. Далее бросается в глаза вот такая чудная строчка

```
sumRight:=c11 + c12 + 13;
```

Бросается в глаза она потому, что выполняющая совершенно аналогичные функции строка чуть выше выглядит ощутимо иначе:

```
sumLeft:=c1 + c2 + c3;
```

В одной из этих двух строк ошибка. Угадайте, в какой из двух. Желательно, с первого раза.

4. Про вас не знаю, но лично я к этому моменту был уже ни в чем не уверен. Я подправил *временно* для отладочных целей программу, вот так:

```
for number:=000000 to 999999 do begin
  number:=123456;
```

Затем поставил точку останова на второй строке и стал отслеживать дальнейшие события. События не замедлили состояться. Переменные *c1* и *c2* соответствовали моим ожиданиям. Переменная *c3* не лезла ни в какие ворота. Совсем немного приглядевшись, обнаружилась и причина:

```
c3:=numberLeft mod 100;
```

Само собой разумеется, вместо 100 должно быть 10. Вы и сами уже догадались, ведь правда, догадались?

5. После исправления всего этого безобразия запускаем окончательный вариант и получаем ответ 7.60 процентов, что, лично мне, кажется более-менее правдоподобным.

Чисто из принципа, успокоиться на этом я не могу, и должен дать хоть какие-то дальнейшие рекомендации по улучшению программы. Разумеется, программа сама по себе совершенна и идеальна, и лучше быть не может – ведь это я её написал. Но чисто из познавательных целей, я рекомендовал бы:

- расширить программу на произвольное количество цифр в номере и построить график зависимости процентов счастливых билетов от количества цифр в них.

- подумать о том, что делать в случае очень большого количества цифр, которое уже не вмещается в стандартные типы данных.

- очень сильно подумать, могут ли быть счастливыми два подряд идущих билета – здесь начинается самая настоящая высшая математика, конкретно – теория чисел.

**Глава 2-2**  
**Вспомнить всё**  
**или**  
**Не очень сложная программа – Ханойские Башни**

**О чём речь?**

Теперь напишем Не Очень Сложную Программу. Игрушку, как и следовало ожидать. Вариантов у меня было два: Ним или Ханойские Башни. Я очень долго не знал, на что решиться, а затем пошёл традиционным русским путём и подбросил монету. Монета указала на Ханойские башни. Всё без обмана, всё по-честному.

Сначала историко–археологический экскурс. Откуда вообще взялись Ханойские башни? Экскурс идёт откуда-то то ли из Тибета, то ли из Непала, какая разница? Где-то очень высоко в горном буддийском монастыре, или даосском монастыре (а это большая разница, читайте Ван Гулика) сидят на горе три монаха. А, может шесть. Если даосских, то три, а если буддийских, то шесть, читайте Ван Гулика. Согласно древнекитайской традиции, буддийских монахов считали паразитами и дармоедами, так что шесть буддийских заменяли трёх даосских.

Итак, перед монахами возвышаются три нефритовых стержня – не путать с нефритовым известно чем в китайской традиции. В самом начале процедуры на левом стержне насажено 64 - шестьдесят четыре - хрустальных диска.

Задачей трёх мудрецов-дармоедов является перенести все 64 - шестьдесят четыре - кольца на третий стержень. Список допустимых манипуляций следующий. Взять можно только верхнее кольцо на любом стержне. В начале, естественно доступны только кольца с первого стержня. Перенести кольцо можно только в двух направлениях. Или на совершенно пустой нефритовый стебель – это понятно. Или на занятый стержень, но такой, где верхнее кольцо меньше по диаметру, чем наше, перетаскиваемое кольцо.

Тут бы я конечно хотел привести отрывок из книги Я.Перельмана и картинку оттуда, но он у нас весь закопирейченный и я очень боюсь, что сразу набегит много маленьких перельманчиков из его родственников и закусают меня до полной чесотки.

Upd. Ан нет. Вышел, вышел срок.

И это все правила. Ни малейшего ума игра не требует, но требует чудовищного и невообразимого терпения.

### Всем всё понятно, программируем

А теперь приступим к программированию. Задача реализуема на любом языке программирования и в любой его версии, которая поддерживает хоть какую-то, но графику. Возможна, разумеется и реализация в текстовом режиме.

*Хорошо быть девочкой в розовом пальто,*

*Можно и в зелёньком, но уже не то* © Кто-то

В текстовом почему-то будет выглядеть менее удачно.

Начнем с самого простого – нарисуем три этих самых нефритовых стержня на каком-нибудь – не чёрном , там плохо видно – фоне.

```
program Hanoi; { 14.02.2017 }
              { 14.02.2017 }
uses
  OpCrt, Graph;
const
  { основание нижней левой части первого слева стержня }
  { ну вы поняли, да? }
  x0 = 100;
  y0 = 300;
  { расстояние между стержнями по горизонтали }
  xIncr = 200;
  { какое-то расстояние по вертикали }
  yIncr = 5;
  { ширина стержня }
  wRod = 10;
  { ширина верхнего (самого маленького) кольца }
  small = 50;
  { увеличение следующего кольца }
  incr = 20;
  { высота стержня }
  h = 200;
  { толщина кольца }
  thi = 20;
var
  driver, mode : integer;
{-----}
procedure Figovina( num : integer);
begin
  SetColor(Green);
  SetFillStyle( SolidFill, Green);
```

```

        Bar3D( x0+(num-1)*xIncr,    y0,
              x0+(num-1)*xIncr+20, y0-h,
              10, TopOn);
end;
{-----}
begin
    driver:=EGA;
    mode:=EGAHi;
    InitGraph( driver, mode, 'c:\bpascal\bgi');

    SetFillStyle( SolidFill, LightGray);
    Bar( 0,0, 639,349);

    Figovina(1);
    Figovina(2);
    Figovina(3);

    Readln;

    CloseGraph;
end.

```

Обратите внимание на процедуру Bar3D. Вместо стержня мы, конечно, имеем типичную сваю, зато получаем максимум визуального результата при минимуме программирования.

В целом получилось так себе, не эротично. Главное, у меня в памяти засело смутное воспоминание о том, что нефрит как будто зелёного цвета, и может быть при этом где-то местами даже и в крапинку. Покрасьте сами.

Далее, мне кажется, надо запрограммировать процедуру для рисования кольца. Точнее, мысль эта является совершенно бесспорной. Спорными остаются только нюансы. Что задавать на вход процедуры? Номер стержня, это понятно. А номер кольца? И откуда его, номер кольца, нумеровать? Или принять кардинальное решение? Каждый раз перерисовывать все кольца на стержне сразу?

А может быть, мы задумались не о главном? Любая наша программа, если что-то рисует на экране, состоит из двух частей – та что нарисовано, и та что внутри, не видно и на самом деле. Интеллигентные люди частенько говорят, что программа делится на интерфейс и бизнес-логику. Интерфейс у нас – те стержни с кольцами, что нарисованы на экране. Бизнес логика то, как эти кольца и стержни будут представлены внутри программы.

Очень часто в вопросах выбора внутренних структур очень полезным оказывается такой подход – не закладываться только на нашу конкретную

задачу, а попытаться заложить решение, которое будет применимо и к задаче более общей.

Я выразился длинно и непонятно? Постараюсь объяснить. Колец у нас потенциально много, даже до шестидесяти четырех. Поэтому сразу становится ясно, что кольца на каждом стержне должны быть объявлены массивом. А стержней всего три, и у многих юных программистов возникает желание объявить три отдельные переменные для этих целей.

Расширим задачу – а если бы стержней было, например, двенадцать? Пришлось бы объявлять массив, однозначно. Поэтому безо всяких сомнений объявляем массив из трех элементов, по количеству стержней. Если такой подход кажется вам неочевидным, то вы не правы, а я прав.

В итоге получаем двухмерный массив – три стержня умножить на шестьдесят четыре кольца

```
const
    maxK      = 64;
    howMany   = 5;
type
    THanoi = array[1..3,1..maxK] of integer;
```

Тройка не объявлена как константа, просто потому, что в этом случае исчезает весь сакральный смысл игры – ведь стержней всегда ровно три. Что означает значение элемента массива? Я считаю, что оно должно обозначать размер кольца – от 1 (самое маленькое) до 64 (самое большое). Присмотревшись к этому объявлению типа повнимательнее, я прихожу к окончательному выводу, что нам нужна процедура рисующая весь стержень с кольцами сразу. На вход ей будет подаваться, само собой, параметр типа – а какого типа?

THanoi – а почему? Почему мы должны перерисовывать весь экран, по сути, ради одного стержня? Честно говоря – какая разница? Видеокарта, она работает быстро и даже очень быстро. Но мы программируем не только ради того, чтобы достичь результата, но и ради того, чтобы достичь определённых педагогических целей. Короче говоря, мы (я) хотим (хочу) научить вас программировать.

Пока не задействованная константа HowMany означает фактическое (не максимальное) количество колец. Иными словами, колец у нас будет ровно пять. Это только для начала, потом можно сделать количество колец задаваемым в начале.

Далее, мы начинаем смотреть на вещи шире, и вместо нашей чудесной Figovin'ы вводим новую сущность – процедуру DrawRod. Rod – так на их смешном языке называется стержень.

```

procedure DrawRod(      num : integer);
  var
    i                      : integer;
begin
  Figovina(num);
  for i:=1 to HaSk[num] do begin
    DrawDisk( num, i, Ha[num,i]);
  end;
end;

```

Остается непояснённой и незакодированной процедура drawDisk. Параметров у неё три. Первый параметр легко узнаваем – это номер стержня, одного из трёх, всегда трёх, не устаю напоминать. Второй параметр – номер диска на стержне, начиная снизу. Почему снизу – ну, во-первых, я так решил. А во-вторых, низ вот он низ, ниже низа только подполье и крысы, а где будет этот верхний диск ещё надо угадать – или сосчитать, кому что легче и привычнее. Так что считать снизу гораздо удобнее для программирования. Третий параметр, очень хитрый с виду, это размер диска. По буквам:

num - номер стержня

i - позиция кольца на стержне, считая снизу, как и договорились

Ha - двумерный массив типа THanoi, содержащий кольца, насаженные на стержни

HaSk - одномерный массив **array[1..3] of integer** – количество дисков на стержне. Чтобы легче запомнить: Ha – Hanoi, Sk – skoka.

Ha[num,i] - размер диска с номером I на стержне с номером num.

```

procedure DrawDisk(      num      : integer;
                        posOnRod : integer; { считаем снизу }
                        numDisk  : integer);
  var
    width                : integer; { ширина рисуемого кольца }
  }
    x1,y1                : integer;
    i                    : integer;
begin
  width:=small + (numDisk-1)*incr;

```

```

x1:=x0 + (num-1)*xIncr;

y1:=y0;
for i:=1 to posOnRod do begin
    y1:=y1 - thi - yIncr;
end;

SetColor(Blue);
SetLineStyle( 0, 0, 2);
SetFillStyle( SolidFill, Yellow);
FillEllipse( x1+10,y1, width,thi);
end;

```

Основная сложность здесь оказалась банальной – попасть серединой кольца ровно в центр стержня. У меня сходу и без раздумий этого не получилось. Пришлось ввести промежуточную переменную width. Не потому, что она была нужна, а потому, что всю формулу сразу я написать правильно не смог. Пришлось поделить её на два оператора. Слона едят по частям.

Потом оказалось, что все диски почему-то рисуются только на первом стержне. Оказалось, я забыл добавить выделенное курсивом выражение (или, по научному - литерал). Тестируйте свои программы! Там есть ошибки, и много ошибок. Даже если вам кажется, что их там нет. Они там всё равно есть.

Ну и само собой, если есть процедура DrawRod, то должна быть и процедура HideRod. Вот она:

```

procedure HideRod(      num : integer);
begin
    SetColor( White);
    SetFillStyle( SolidFill, White);
    Rectangle( x0+(num-1)*xIncr      -80,      y0,
               x0+(num-1)*xIncr+wRod+80,      y0-h-30);
end;

```

С виду непонятная константа 30 в последней строке имеет очень понятный на самом деле смысл – мы использовали процедуру Var3D, а она автоматически занимает немного больше места на экране, чем мы предполагаем – в случае, если последний параметр вызова равен TopOn.

Кстати о главном – вы уже подумали, как мы будем тестировать нашу процедуру? Сколько случаев надо проверить? Вопрос не очень интересный – стержней всего три, можно тупо проверить все. А по

минимуму? По минимуму два – левый стержень и средний. Так что какая разница – проверяйте всё.

Теперь несложная процедура, которая перед началом игры размещает кольца по стержням. Точнее, насаживает все кольца на первый стержень. А можно это сделать не тупо ручками я, а по-умному в цикле? Можно. Вот вы и сделайте.

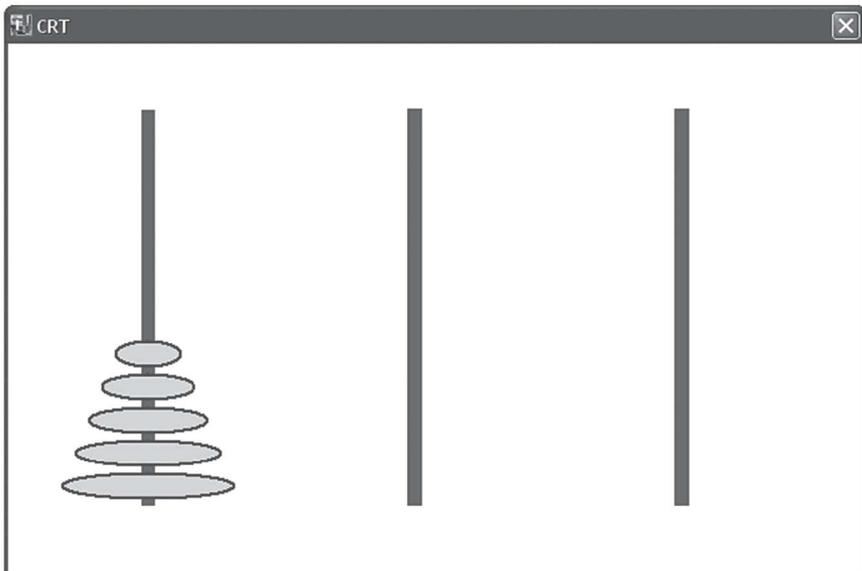
```
procedure Init;
begin
    Ha[1,1]:=HowMany;      Ha[1,2]:=HowMany-1;
    Ha[1,3]:=HowMany-2;   Ha[1,4]:=HowMany-3;
    Ha[1,5]:=HowMany-4;
    HaSk[1]:=5;
end;
```

А вот так выглядит головная программа. *Пока* так выглядит.

```
Init;

DrawRod(1);
DrawRod(2);
DrawRod(3);
```

А это восхитительный и несравненной красоты результат её (нашей (моей)) работы. Вам не нравится? Ну даже и не знаю... Мне тоже не нравится. Я хочу, что бы стержни были как будто нефритовые, а кольца как будто малахитовые, и всё в 3D. Но не выходит у Данилы каменный цветок, не выходит... Картинка страшная, из бета-версии, если вы понимаете о чём я говорю.



Мы уже в состоянии перемещать кольца с одного стержня на другой. Актуальный теперь вопрос – как выбрать стержень с какого перемещать и как выбрать стержень на какой перемещать. Мышь нам недоступна – могу предложить клавиатуру. Если мы имеем под рукой только клавиши, то нам, соответственно, доступен только курсор.

А курсор надо сначала нарисовать, а потом, чтобы переместить в другую позицию, курсор надо стереть.

Сначала необходимо объявить константу:

```
const
  { ширина курсора }
  curSize = 50;
```

Теперь объявим несложную процедуру для рисования курсора:

```
procedure DrawCursor(   num : integer);
begin
  SetColor(Blue);
  SetFillStyle( SolidFill, Blue);
  Bar( x0+(num-1)*(xIncr+3) - 15, y0+10,
      x0+(num-1)*(xIncr+3) + curSize - 15,y0+20);
end;
```

Константы получены эмпирическим путём и обсуждению не подлежат. Разумеется, вслед за этой процедурой немедленно следует и процедура для стирания курсора:

```
procedure HideCursor(      num : integer);
begin
  SetColor(backColor);
  SetFillStyle( SolidFill, backColor);
  Bar( x0+(num-1)*(xIncr+3) - 15, y0+10,
      x0+(num-1)*(xIncr+3) + curSize - 15,y0+20);
end;
```

Теперь, после всего этого безобразия, пишем *основной цикл*. Изначально основной цикл прост и незатейлив. В советские времена в таких случаях говорили – *простой, как батон за тринадцать копеек*. Вот:

```
gamover:=false;
```

```
repeat
until gamover;
```

Переменная *gamover*, разумеется, где-то раньше объявлена, как *boolean*. Если вам кажется, что программа наша рассыпается на части и становится местами необозримой, переживать не надо – в приложении вы найдёте полный последовательный текст.

Теперь запишем основной цикл нашей программы на псевдокоде, а перед этим изложим наши действия простыми человеческими словами. Стрелками вправо и влево перемещаем курсор – вправо и влево, контролируя невыход за крайние позиции. Нажимая пробел в первый раз, снимаем кольцо со стержня, под которым курсор находится. Если кольцо уже снято, то повторное нажатие пробела насаживает кольцо на стержень, под которым курсор сейчас. Если это тот же самый стержень, с которого кольцо сняли, то ничего страшного – просто вернём кольцо назад. Сделаем, как было.

*Тут Левко стал замечать, что тело ее не так светилось, как у прочих: внутри его виделось что то черное © Гоголь Майская ночь или утопленница*

Тут я стал замечать, что в теле нашего алгоритма виднеется что-то мутное. Что такое – *снимаем кольцо*? Когда-то, давно я уже писал эту программу, и тогда трактовал это так – мы просто запоминаем в уме, что с этого стержня снято кольцо, но визуально никак это не отображаем. А когда мы выбираем стержень, на который кольцо будет перемещено,

перерисовываем оба стержня – и тот, с которого снять, и тот, на который надеть.

К счастью, тот вариант программы я благополучно потерял. Да, я очень аккуратный, но случаи бывают разные. Оно и к лучшему. Сейчас я вижу картину по иному. При нажатии на пробел кольцо исчезает – визуально – со стержня, на котором оно было, и зависает где-то в небесах. Небеса мы разместим в правом верхнем углу экрана. При повторном нажатии пробела кольцо в небесах исчезает и возвращается на выбранный стержень.

Быстро пишем процедуры рисования и стирания кольца в небесах. Всё, что мы делаем, можно сделать по хорошему и по плохому, по правильному и неправильному, по сложному и по простому, продолжать можно долго. К сожалению, все эти дихотомии совсем не тождественны, скорее даже наоборот. По умному и правильному, мы должны написать процедуру рисования *одного* кольца и вызывать её из процедуры рисования *всех* колец на стержне. Но это же столько писать и переписывать... Вот если бы мы правильно спроектировали нашу программу с самого начала... Но что вылезло, то и вылезло.

Короче – пишем совершенно отдельную процедуру для рисования кольца в небесах:

```
procedure DrawDiskInOuterSpace(    numDisk : integer);
var
    width                : integer;
begin
    width:=small + (numDisk-1)*incr;

    SetColor(Blue);
    SetLineStyle( 0, 0, 2);
    SetFillStyle( SolidFill, Yellow);
    FillEllipse( 550,40, width,thi);
end;
```

Процедуру удаления кольца нафантазируйте сами. Впрочем, напоминаю, что в приложении всё есть. Мой совет – процедура удаления кольца не должна иметь параметров.

Теперь вернёмся к нашему псевдокоду:

```
gamover:=false;
```

**repeat**

```
если нажали стрелка влево
    если не в крайней левой позиции
        переместить курсор влево
если нажали стрелка вправо
    если не в крайней правой позиции
        переместить курсор вправо
если нажали пробел
    если кольца в небесах нет
        отправить его туда
    если кольцо в небесах есть
        убрать с небес
        поместить на текущий стержень
если нажали Esc
    то, собственно, всё – gamover как он есть
```

**until** gamover;

Остаются неясные моменты – они всегда остаются, верьте мне, они будут прояснены в процессе кодирования. Если вы их, эти мутные моменты, не увидели с первого взгляда, не огорчайтесь – я тоже не увидел.

### Решительно кончаем программу

Сначала поглядим внимательно на наш псевдокод. Хорош ли он? В первом приближении, хорош. Со второго взгляда возникают отдельные вопросы.

Вопрос номер один – вот эта часть псевдокода:

```
если кольцо в небесах есть
    убрать с небес
    поместить на текущий стержень
```

Мы забыли проверить, что поместить кольцо на выбранный стержень мы имеем право только в том случае, когда размер кольца в небесах меньше размера верхнего кольца на стержне. Или когда на стержне колец вообще нет.

Вопрос номер два не так бросается в глаза:

```
если кольца в небесах нет
    отправить его туда
```

Выглядит абсолютно невинно, однако есть один вопрос – что будет, если колец на стержне вообще нет? В псевдокоде ничего плохого, на то он и

псевдокод, а вот в реальной программе получим безусловный Range Check Error – в нашем конкретном переводе – выход за границы массива. Следовательно, требуется дополнительная проверка.

Как разработка любой программы, разработка нашей тоже движется одновременно во все стороны. Займёмся обработкой нажатия клавиши:

```
repeat
  if OurKeyPressed then begin
    sc:=OurReadKey;
    if sc = ArrowRight then begin
      { что-то }
    end else
    if sc = ArrowLeft then begin
      { что-то }
    end else
    if sc = SpaceBar then begin
      { что-то }
    end else
    if sc = Esc then begin
      Break;
    end;
  end;
until gamover;
```

Главное, идея понятна – хотя пока мы реализовали полностью только обработчик нажатия клавиши Esc – самый безобидный обработчик из всех обработчиков.

С тех пор, как мы создали процедуры рисования и стирания курсора в заданной позиции, вопросов по клавишам *влево* и *вправо* не осталось. Быстро заполняем. Клавиша влево:

```
if sc = ArrowLeft then begin
  if whereCurs > 1 then begin
    HideCursor(whereCurs);
    whereCurs:=whereCurs - 1;
    DrawCursor(whereCurs);
  end;
end else
```

Клавишу вправо запрограммируйте самостоятельно. Скажем дальше.

Самое сложное – обработать нажатие пробела. Пробел нажимается в двух целях – снять кольцо со стержня и надеть кольцо. Задача распадается на

две. При этом, в любом случае, мы должны выполнить проверку, имеем ли мы право проделать заказанную манипуляцию.

Начнём, само собой, со снятия кольца. А точнее, начнём со введения вспомогательной переменной по имени `skoKa`. Эта загадочная переменная соответствует количеству колец на выбранном стержне. Можно и без неё, переменной, но с ней проще. Текст программы не такой громоздкий и, главное, гораздо более понятный. В начале обработчика пробела мы должны эту величину запомнить, а в конце обработчика восстановить. Насчёт восстановить я не вполне уверен, может и не обязательно – но, в любом случае, никому не повредит.

Запомнить и восстановить это так:

```
skoKa := HaSk[whereCurs];  
.....  
HaSk[whereCurs] := skoKa;
```

// Суровая правда жизни

Так, как мы здесь и сейчас программируем, вам, скорее всего, программировать не придётся. Все современные языки программирования поддерживают обработку событий. Точнее будет сказать, все современные среды обработки поддерживают обработку событий. Ну, вы поняли – что *совой об пень, что пнём об сову*.

В чём смысл программы, ориентированной на обработку событий? Или программы, обрабатывающей прерывания? В обычной жизни вы пишете процедуру. Потом вы её вызываете совершенно явным образом – явным, это значит, что вы абсолютно уверены, в каком именно месте вашей программы вы эту процедуру вызываете и при каких условиях. А знаете вы это по той простой причине, что именно вы и только вы эту процедуру вызываете. Если над проектом работает несколько человек, то процедуру может вызывать ваш коллега, но от этого теоретически ничего не меняется.

Если стержень пустой, проигнорировать нажатие – ведь если колец на стержне нет, то и делать ничего не надо, потому что делать что-то невозможно, да и не нужно.

Запомнить, что стержень, с которого будет снято кольцо уже выбрали, и запомнить для себя, какой именно стержень.

Если стержень *с которого* сняли кольцо уже выбран раньше (и нами запомнен само собой), то проверить, первое, что сейчас выбрали не тот же самый стержень, и, второе, что размер кольца выбранного ранее на стержне с которого сняли кольцо, меньше, чем размер верхнего кольца стержня *на который* поместили кольцо.

Если всё удачно совпало, перенести диск со стержня с которого на стержень на который. Перенести сначала в памяти (массивы `Ha` и `HaSk`) а потом и на экране.

На всякий случай проверить, не закончена ли игра.

Извините, что так занудно.

И всё это будет происходить в процедуре, обрабатывающей нажатие кнопки мыши, или, как чаще принято говорить в *обработчике* нажатия мыши.

В Турбо Паскале тоже очень даже можно организовать обработку событий. Перехватываете соответствующее прерывание уровня ДОСа или БИОСа, вешаете на него свой обработчик – и, собственно, всё. Звучит загадочно, а на самом деле для этого даже ассемблера знать не надо. Я расскажу вам об этом в соответствующей главе. Если не забуду.

// конец Суровой правды жизни

Теперь, после реализации вспомогательных функций, переходим к основному. Основная функция у нас одна – обработчик нажатия клавиши *Пробел*. Текст длинен и извилист.

```
if sc = SpaceBar then begin
  skoka:=HaSk[whereCurs];
  if (outerDisk > 0) and
    (( skoka = 0) or (outerDisk < Ha[whereCurs,skoka]))
  then begin { plus }
    skoka:=skoka + 1;
    HaSk[whereCurs]:=skoka;
    Ha[whereCurs,skoka]:=outerDisk;
    HideDiskInOuterSpace;
    outerDisk:=0;
    HideRod(whereCurs);
    DrawRod(whereCurs);
  end
else begin { minus }
  if (skoka >= 1) and (outerDisk = 0) then begin
    outerDisk:=Ha[whereCurs,skoka];
    Ha[whereCurs,skoka]:=0;
```

```
    skoka:=skoka - 1;
    HaSk[whereCurs]:=skoka;
    HideRod(whereCurs);
    DrawRod(whereCurs);
    DrawDiskInOuterSpace(outerDisk);
end;
end;
HaSk[whereCurs]:=skoka;
```

Что здесь нужно прокомментировать? Достаточно сложные условия, отвечающие за то, чтобы вообще хоть что-то делать. Использование вспомогательной переменной `skoka`, заметно сокращающее текст, влечет за собой столь же заметную необходимость поддерживать её актуальное состояние. За всё приходится платить.

## **Глава 2-3**

### **Всё таки кое что новое**

#### **Как устроена большая программа**

Более или менее большая программа состоит из:

Основного текста (основного модуля) - называйте как хотите  
Нескольких внешних модулей

В приличной программе в отдельный модуль вынесены объявления типов, констант и объектов – классов. Обычно одни объявления констант ссылаются на другие объявления констант, объявления типов ссылаются на объявления констант и объявления других типов, а объявления переменных на объявления типов (иначе зачем же вы эти типы объявляли) и на константы. В каждом отдельно взятом модуле может быть секция инициализации, и свои секции объявления типов и констант, хотя это и нежелательно.

Секция инициализации в каждом модуле нежелательна так же. Лучше собрать всё их содержимое со всех модулей и объединить в один общий модуль инициализации, явно вызываемый из главной программы. Также неплохо было бы избегать присутствия в программе не вами написанного кода, который к тому же вызывается по чужому велению. Это возможно только в том случае, если всю программу я пишу сам. Если модуль чужой, то порядочный автор перенесёт всё из секции инициализации в отдельную инициализационную процедуру. Зачем это надо, поймёте с годами.

#### **Очень простая и маленькая большая программа**

Простая эта программа потому, что она действительно очень простая – как бы я ни старался выбрать что-то ещё проще, я бы не смог – ну не таблицу же умножения нам программировать! Вместе с тем, мне бы очень хотелось, чтобы программа эта, будучи очень простой по содержанию, была достаточно сложной по форме. Ну, вот такой я фантазёр – обычно хотят строго наоборот.

А я хочу того, чтобы якобы маленькая программа наша была с виду совсем как большая – содержала все особенности, секции, модули большой программы. А заодно и все проблемы большой программы.

- секция uses – не знаю как перевести © А.С.Пушкин
- секцию типов
- секцию констант - обычно наоборот - сначала секция констант – вы конечно понимаете, почему
- про секцию переменных даже не говорим
- и много процедур и функций.

И ещё я хочу, чтобы программа наша была разбита на отдельные модули:

- головной,
- объявления типов и констант (но ни в коем случае не глобальных переменных)
- и несколько модулей с процедурами и функциями, которые действительно что-то делают.

Давайте, наконец, напишем ту самую маленькую и простую программу, которую я, в бытность мою преподавателем, заставлял писать всех пионеров.

Пионер – здесь это подразумевает не сторонника Владимира Ильича Ленина с красным галстуком, который в свободное от сбора макулатуры время бьёт морду сторонникам Лейбы Давидовича Троцкого. Аналогично в царской России традиционно гимназисты (ученики гимназий) били морды реалистам (ученикам реальных училищ) и наоборот. Эти учебные заведения для удобства так рядом и размещали.

Нет, здесь у себя, пионерами я называю розовых юношей старшего школьного возраста, а то и младшего студенческого, которые с какой-то стати вдруг решили, что уже умеют программировать.

А зловещая программа эта называется *Решение квадратного уравнения*. Для тех, кто уже забыл, квадратное уравнение это

$$ax^2 + bx + c = 0;$$

А корни его определяются так

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Выражение под корнем (знаком радикала) обычно называется дискриминантом. Если дискриминант равен нулю, корней не два, а один. Если дискриминант меньше нуля, то корней нет. То есть они, конечно, есть, но не действительные, а комплексные – а зачем нам комплексные корни? Комплексные корни нам не нужны.

Сферический пионер в вакууме пишет программу примерно так:

```
program Kwa;
  uses
    OpCrt;
  var
    a,b,c           : single;
    dis             : single;
    x1,x2           : single;
begin
  ClrScr;
  write('vvedite a,b,c >> ');
  readln(a,b,c);

  dis:=b*b - 4*a*c;
  x1:=(-b+Sqrt(dis))/2*a;
  x2:=(-b-Sqrt(dis))/2*a;

  writeln('x1 = ',x1:8:2,' x2 = ',x2:8:2);
  readln;
end.
```

У этой программы есть масса недостатков. А что будет если:

Дискриминант меньше нуля? Корней нет, но мы об этом не узнаем.

Уравнение линейное (a=0)? В теории этого не может быть, потому что тогда это будет уже совсем не квадратное уравнение. Но на практике это будет! Именно так, с восклицательным знаком!

Тождество (0=0 3=3)

Корней не, но по-другому. Что-то вроде (7=3)

То, что вводимые данные не проверяются на корректность, то есть что они действительно числа, а не бессмысленный набор символов, это мелочь и мы её игнорируем. И, чисто эстетическое – если дискриминант равен нулю, то эта программа выдаст два одинаковых корня. Это, безусловно, правильно, но некоторые предпочитают думать что их, корней, в таком случае только один. Можно было бы пойти этим некоторым навстречу, нам пустяк, а им приятно.

Но вы, конечно, поняли, что это лишь шутка. Ни один пионер такой программы никогда не напишет.

Он напишет примерно такое:

```

program KwaReal;
uses OpCrt;
var a,b,i,e, koren,a21 :
single;
var j,k,
c99:integer;
begin ClrScr;
writeln('vv');
readln(a,b,i);e:=Sqr(b ) - 4* a *(i+j);
koren:=((-1)*b+ Sqrt(e))/(a+a);
a21:=
(-b - Sqrt(b*b -1*(a*i+a*i+a*i+a*i)))/2*a;
writeln('x1 = ',
koren, 'x2 = ',a21+k);
for j:=k to Round(e) do begin
end;
readln;

```

end.

Прошу обратить внимание на пустые строки перед **end**. Не знаю почему, но они там всегда есть, в количестве штук примерно до двадцати. Видимо, строки эти несут некий сакральный смысл. Бессмысленный цикл в конце также является удачным штрихом, придающим картине густой оттенок реализма.

И ещё. Программа, безусловно, работает, и работает правильно. Если задать на вход хорошие данные.

В реальной жизни, если нам понадобится программно решить квадратное уравнение, то это означает, что нам понадобится решить по меньшей мере тысячу квадратных уравнений. Разъясню. Если мне надо решить одно квадратное уравнение в год, я решу его на бумажке с помощью калькулятора. Если мне надо решить одно квадратное уравнение в месяц, я всё равно решу его на бумажке. Писать для этих жалких целей программу - себе дороже. А вот если решать квадратные уравнения придётся каждый день, то уже стоит подумать. Немедленно и сразу следует обдумать возможность применения написанного нами текста в безинтерфейсном режиме – то есть для вызова из других программ.

И программа наша будет представлять из себя процедуру, на вход которой поступают коэффициенты уравнения, а на выходе корни или некий признак, указывающий на тождественность или несовместимость уравнения. В виде отдельной программы эта штукавина не будет жить никогда. То есть будет жить, конечно – как же иначе мы будем её тестировать? Но недолго.

Но мы пока учимся. И это будет отдельная, независимо выполняемая программа. Мало того, сама по себе эта основная программа ничего решать не будет. Ее задачей станет ввести входные данные, проверить коэффициенты, вызвать на выполнение один из трёх внешних модулей – квадратное уравнение, линейное уравнение, всякая фигня, получить ответ от них и вывести результаты. Я сам человек не очень занудный, потому не стал выделять в отдельный случай приведённое квадратное уравнение, что делают в каждой порядочной школе. Кстати, вспомните, что это такое – приведённое квадратное уравнение. Со стороны подсказывают, что есть ещё вариант с чётным коэффициентом  $A$ .

Так что проектируем программу. Попутно даём имена модулям. Это очень важно и очень сложно – *Как вы лодку назовёте, так она и поплывёт.*

Program Kwa. Основная программа. Ее задача ввести входные данные, определить, что именно перед нами – квадратное уравнение, линейное

уравнение, вообще не уравнение, вызвать соответствующий модуль, получить от него ответ, вывести ответ.

Вызываемые модули и их обязанности.

Unit KwaUr      Возвращает количество корней (один, два, ноль) и сами корни.

Unit LineUr     Возвращает один корень линейного уравнения

Unit ChtoTo     Возвращает одно из двух – или у нас тождество ( $0=0$ ), или корней нет вообще ( $3=0$ ).

А что, собственно, означает *возвращает количество корней*? Скорее всего, это целое число, принимающее значения 0,1,2.

Что должен возвращать третий модуль? Вариантов завершения его работы два: либо тождество, либо корней нет. Корней нет – это уже было, означает, что количество корней равно нулю. А если тождество? Вернуть бесконечно большое число в качестве количества корней мы не можем. Программист халтурщик вернёт что-то вроде 99 – практически бесконечное число. Мы пойдём другим путём и определим свой тип, причем определим его в отдельном модуле, на который будут ссылаться остальные модули и главная программа

```
unit KwaType; { 01.06.2017 }  
              { 01.06.2017 }  
interface  
  type  
    TRes = ( r0, r1, r2, rall);  
implementation  
end.
```

Мелкое замечание – даты сверху – первая – когда модуль создан, вторая – когда последний раз вносились изменения. И вам того же желаю.

Хорошо бы в этот же модуль впихнуть и объявления констант. А какие у нас, собственно, могут быть константы? Как сказал кто-то умный – не я – с точки зрения программирования ноль и единица константами не являются и могут быть свободно употребляемыми в исходном тексте программы в первозданном виде. Тем не менее, константы здесь нам

абсолютно необходимы, но только другие, не численные, а строковые константы.

Наша программа будет выдавать сообщения – в ответ на некоторые ситуации. Тексты сообщений время от времени придётся менять – по той банальной причине, что они не понравились заказчику или пользователю – это, вообще говоря, разные люди с разным характером.

Крупное замечание – нас не должно волновать, каким образом в памяти представляется наш новый тип TRes. Даже и не думайте об этом. Ну или по крайней мере не думайте до тех пор, пока не захотите записать переменную этого типа в файл.

Обратите внимание, что после того как мы разделили нашу первоначально одну программу на четыре части, работы у нас явно прибавилось – ведь теперь нам надо организовать взаимодействие между модулями. И так всегда и везде.

Как будут связаны между собой наши модули? Первый модуль будет вызывать три остальных. Все четыре будут ссылаться на модуль с объявлениями типов и констант. И только этот модуль ни на кого ссылаться не будет. *И только хомячки никого не любят* © Шутка.

## Первый модуль

```
unit KwaA; { 01.06.2017 }
           { 09.06.2017 }

interface
  uses
    KwaType;
{-----}
procedure MakeA(   a,b,c   : single;
                  var x1,x2 : single;
                  var res   : TRes;
                  var textRes : string);
{-----}
implementation
{-----}
procedure MakeA(   a,b,c   : single;
                  var x1,x2 : single;
                  var res   : TRes;
                  var textRes : string);
var
  dis           : single;
```

```

begin
  dis:=b*b - 4*a*c;

  if dis > 0 then begin
    x1:=(-b+Sqrt(dis))/(2*a);
    x1:=(-b-Sqrt(dis))/(2*a);
    res:=r2;
    textRes:=r2Text;
  end else
  if dis = 0 then begin
    x1:=(-b)/(2*a);
    res:=r1;
    textRes:=r1Text;
  end
  else begin
    res:=r0;
    textRes:=r0Text;
  end;
end;
{-----}
end.

```

Комментарии.

Хотя мы и выделили в отдельный модуль случай полного квадратного уравнения, всё же и внутри него (модуля) мы имеем три ситуации – дискриминант больше нуля - два корня, дискриминант равен нулю - один корень, и дискриминант меньше нуля. В этом случае корней как бы и нет – они вроде бы и есть, только они комплексные, но для нас это всё равно, что корней вообще нет.

Проверки на деление на ноль здесь нет – догадайтесь сами, почему.

Обратите внимание на даты, я писал этот шедевр программисткой мысли ровно восемь дней. Это потому, что я очень аккуратный и внимательный к мелочам.

Второй модуль, как легко предсказать, будет ровно в полтора раза проще:

```

unit KwaB; { 01.06.2017 }
           { 09.06.2017 }

interface
  uses
    KwaType;
{-----}
procedure MakeB(      b,c      : single;
                   var  x1      : single;

```

```

                var res      : TRes;
                var textRes : string);
{-----}
implementation
{-----}
procedure MakeB(   b,c      : single;
                 var x1    : single;
                 var res   : TRes;
                 var textRes : string);
begin
    x1:=(-c)/b;
    res:=r1;
    textRes:=r1Text;
end;
{-----}
end.

```

Комментарии:  
Комментариев нет

Третий модуль, для вырожденного случая, что интересно, ничуть не короче второго.

```

unit KwaC; { 01.06.2017 }
           { 09.06.2017 }
interface
    uses
        KwaType;
{-----}
procedure MakeC(   c      : single;
                 var res  : TRes;
                 var textRes : string);
{-----}
implementation
{-----}
procedure MakeC(   c      : single;
                 var res  : TRes;
                 var textRes : string);
begin
    if c = 0 then begin
        res:=rAll;
        textRes:=rAllText;
    end
    else begin
        res:=r0;
        textRes:=r0Text;
    end;
end;
{-----}
end.

```

А вот наконец самый главный модуль, он же головная программа.

```
program KwaKwa;
  uses
    OpCrt,
    kwaType, kwaA, kwaB, kwaC;
  var
    a,b,c                : single;
    numOfX               : integer;
    x1,x2                : single;
    res                  : TRes;
    textRes              : string;
{.....}
  procedure InPut;
begin
  Write('a >> ');
  ReadLn(a);
  Write('b >> ');
  ReadLn(b);
  Write('c >> ');
  ReadLn(c);
end;
{.....}
  procedure OutPut;
begin
  WriteLn(textRes);

  Writeln('x1 = ', x1:8:2);
  Writeln('x2 = ', x2:8:2);

  ReadLn;
end;
{.....}
begin
  ClrScr;
  InPut;

  if a <> 0 then begin
    MakeA( a,b,c, x1,x2, res, textRes);
  end else
  if b <> 0 then begin
    MakeB( b,c, x1, res, textRes);
  end
  else begin
    MakeC( c, res, textRes);
  end;

  OutPut;
end.
```

Поглядим на результат и поразмышляем. Хорошо ли это? Довольны мы ли результатом? Что делать?

Во-первых, программу надо протестировать. Протестировать - не значит прогнать один единственный пример и убедиться, что программа на нём, этом единственном примере работает. Протестировать – значит прогнать очень много-много примеров. И, очень желательно, прогнать не самому. Любой программист очень любит свою программу, особенно ту, которую только что написал. И ему, программисту, очень не хочется этой программе сделать больно, а тем более – убить программу насмерть. Программист запускает тесты бережно и осторожно, имея в виду те ситуации, которые он предусмотрел, и в которых его программа точно будет работать.

Поэтому тестированием должен заниматься отдельный, специально обученный человек. Желательно такой человек, у которого есть личные причины вас ненавидеть. Но этого мало. Главное, внушить специальному человеку, что успешное тестирование не то, при котором все тесты прошли успешно. Успешное тестирование то, при котором программа рухнула, упала и разбилась вдребезги. Если программа с первого раза прошла все тесты - тестировщик должен быть наказан.

Теперь вручную (что значит вручную – станет ясно дальше) напишем комплект тестов для прогона по всем закоулкам нашей программы. До того я запустил программу на одном единственном примере. Пример был 1 2 3, само собой. Программа честно ответила, что корней нет – и это правильно. А теперь подойдем к вопросу скрупулёзно и методически.

Возможные варианты:

1.  $a < 0$ 
  - 1.1 дискриминант положителен  
3,10,3 два корня -3,-1/3
  - 1.2 дискриминант отрицателен  
2,4,3 корней нет
  - 1.3 дискриминант равен нулю  
5, 10, 5 один корень  $-1^{2/3}$
2.  $a = 0, b < 0$ 
  - 0, 2, -10 один корень 5

3.  $a = 0, b = 0, c < 0$   
0, 0, 10    корней нет
4.  $a = 0, b = 0, c = 0$   
0, 0, 0    тождество

Пропускаем все тесты через нашу программу, или, наоборот, нашу программу пропускаем через все тесты. Что мы видим? А почему?

А видим мы два отклонения результатов фактических от результатов ожидаемых.

В тесте 1.1 получаем фактический результат  $-3; 0$ . Приглядевшись к исходному тексту, видим чудесное:

```
if dis > 0 then begin
  x1:=(-b+Sqrt(dis))/(2*a);
  x1:=(-b-Sqrt(dis))/(2*a);
  res:=r2;
  textRes:=r2Text;
end else
```

Второе  $x1$  получено путём размножения первой строчки с неаккуратным её размножением. Комментариев два:

Первый. Только что мы наблюдали основной и самый простой источник ошибок. Ошибки размножения или, по-умному, Copy-Paste.

Второй. Хотя Delphi не является темой книги, там эта ошибка была бы отловлена на этапе трансляции – мы получили бы совет (Hint), что значение переменной  $x1$  после первого присваивания больше никогда не используется. Но вся печаль в том, что Hint и Warning большинством программистов тупо игнорируются, или, при возможности, вообще отключаются, так что, в конечном итоге результат был бы тот же.

Но это ещё не всё. В тесте 1.3 ожидаемый ответ  $-1^{2/3}$ , реально полученный  $-1$ , без дробей. Метнувшись в исходный текст, и судорожно изучив его под микроскопом, никаких видимых ошибок я не обнаружил. Тогда меня осенило и я пересчитал на бумажке корни уравнения. Оказалось, программа права, а я ошибся в расчётах. Бывает и так.

А что теперь? Успокоиться на достигнутом? Можно посадить тестировщика, привязать его за ногу к компьютеру и пусть себе тестирует до бесконечности. Хорошая мысль. Более прогрессивной считается, однако, технология автоматической генерации тестов. Идея красива и благородна – компьютер сам генерирует тесты, сам запускает нашу программу (модуль) и сам проверяет ответы на правильность и выставляет нам оценку. И никаких злых тестеров.

Вся процедура очевидным образом распадается на три этапа:

1. оформить нашу программу в виде одного вызываемого модуля
2. Написать оболочку тестирующей программы, которая, оболочка, вызывает наш тестируемый модуль, проверяет ответ и - нет, не выводит результат – в пишет все результаты в файл протокола для дальнейшего анализа.
3. Написать модуль проверки – возвращающий заведомо правильный результат.

Здесь перед нами возникает интересный вопрос – если мы можем написать проверочный модуль, дающий правильный ответ, то зачем нам писать основной, проверяемый модуль, дающий в качестве ответа непонятно что?

Первый вариант ответа – мы можем решить задачу другим способом и сравнить полученные ответы. В нашем случае, для квадратного уравнения, я могу сходу несколько ну очень альтернативных методов. Можно применить алгоритмы одномерной оптимизации. Можно использовать метод Монте-Карло. Можно провести аналитическое исследование на предмет определения количества корней. Загогулина в том, что все эти методы для разработки и тестирования несколько сложнее нашего исходного – хотя, возможно, и короче текстуально.

Второй вариант ответа – если мы умеем по квадратному уравнению найти его корни, то с тем же успехом можем по двум корням составить квадратное уравнение. Это намного проще и практически безошибочно, даже для программиста уровня сильно ниже среднего. Очередная загогулина в том, что выпадают всякие вырожденные и извращённые случаи. Впрочем, если мы в качестве двух корней квадратного уравнения будем выбирать два случайных комплексных числа, то всё как будто наладится – но вернётся проблема сложности реализации – надо сначала реализовать математику комплексных чисел.

Если бы я знал третий, самый правильный вариант ответа, я бы сразу его рассказал. Но я его не знаю, по крайней мере, для нашего специфического, не самого сложного случая. Что странно, бывают ситуации, когда выбор решения усложняет простота решаемой задачи. Зачем напрягаться, если можно посчитать на пальцах?

И ещё, я уже говорил о необходимости объявления констант с текстами потенциальных сообщений. Посмотрите на программу с точки зрения пользователя, запускающего её, программу, в ручном режиме. Какие сообщения от неё, программы, вы бы ожидали увидеть? Теперь представьте этот же модуль, но вызываемый автоматически, в контексте более крупной программы. Что должен наш модуль писать в протокол – потому что неизбежно настанет торжественный момент поиска крайнего и виноватого? Объявите приличные случаю константы.

## Глава 2-4

### По ту сторону - опустимся чуть ниже

#### Вступление

После изучения этой главы вы станете маленьким хакером. Или по крайней мере вы станете маленьким крутым хакером образца приблизительно двадцатилетней давности. Двадцатилетней – просто потому, что сейчас всё это уже никому не нужно. Но я охотно раскрою вам все свои маленькие тайны тех лет, раскрою легко, быстро и без колебаний – в основном потому, что ничего сейчас уже и не помню.

Но, в любом случае, даже если я не смогу в этой главе ничему вас научить, или если вы ничему из этой главы не научитесь, или всё это окажется для вас совершенно бесполезным – всё равно вы можете после этого считать себя самым крутым на свете хакером. Просто потому - а кто вам запретит?

Сначала я даже не хотел упоминать здесь встроенный ассемблер – потому что это очень просто. А с другой стороны – почему бы и нет? Если это очень просто, а людям приятно – то вперёд!

С сего мы начнём – начнём мы с совершенно безобидной программы. Ничего такого она не делает и никакими такими способами для этого она не пользуется. Но мы из этой программы извлечём две пользы – первое, увидим, насколько всё уныло и печально. Второе – большинство выполняемых нами операций можно выполнить и другими способами, вот мы и потренируемся на кошках.

#### Начало. Просмотр картинок

Пусть на вход нашей программы без конца поступают, к примеру, картинки. Для чего – ну не знаю, честное слово. И наша задача эти картинки хранить в памяти и при первом требовании отображать на экране.

Если вы думаете, что эта задача искусственная и вообще высосана из пальца – вы ошибаетесь. Вам непременно придётся этим заниматься. А если вы думаете, что есть какие-то уже готовые средства для управления коллекциями картинок, то вы опять конечно правы. Пусть это будут не

картинки, а песенки. И для песенок есть программа? А если у нас на вход поступают не картинки и не песенки, а фиговинки? А для фиговинки есть программа? Эта книжка пишется для будущих профессионалов – и запомните, лично у вас всё всегда будет плохо. Или очень плохо. На вход вашей программы будут поступать именно такие данные, для которых никаких средств никто ещё не придумал.

Суммируем требования – на входе есть несколько файлов заранее неизвестного, но, скорее всего, немаленького размера, наша задача сохранить их в памяти – причем для заранее не установленных целей – а что-то потом по требованию с ними сделать – например, сохранить под другим именем и в другом формате, послать на экран, написать поверх хорошее доброе слово...

Повторяю – задача эта абсолютно реальна, и, став – если – программистом, ты, мой маленький дружок, будешь заниматься этим бесконечно и безначально.

А теперь постановка задачи:

Написать набор процедур. Набор процедур будет оформлен в виде модуля. Собственно процедуры:

1. На вход поступает имя файла, который нужно загрузить.
2. Мы загружаем сам файл – в предыдущей процедуре.
3. По имени файла выдаем его на экран. Тут у нас будут серьёзные проблемы. Зато узнаем, как устроен файл с картинками изнутри.
4. Для удобства процедура выдаёт количество файлов и их имена. Куда – неважно, она может выводить их на экран или возвращать через программный интерфейс.

Итого четыре процедуры, которые нам надо написать.

Как всегда поступают настоящие программисты, пишем интерфейсы программ, при этом всячески стараясь забыть, а что у них будет внутри.

*Здесь должна была быть цитата из Дейкстры, но я её не нашёл. Может, это не Дейкстра был? Дейкстра, если что, - это такой величайший программист всех времён и народов. Смысл мысли был в том, что когда программист программирует какую-то конкретную фиговню, то он должен быть на ней, на фиговне, и сосредоточен, но не*

полностью, а только на 99%. А в уголке мозга у него, программиста, должна быть красная лампочка в погашенном состоянии. И когда программист тем текстом, который он пишет сейчас, портит что-то из написанного ранее, а, возможно, также их того, что будет написано только позже – лампочка должна ярко вспыхнуть. Ну, вы поняли. © Всё-таки Дейкстра, я же сам по себе не мог этого придумать

0. Модуль оформим в первом приближении так:

### Вариант 1

```
unit XPicture;
{-----}
interface
procedure AddPic(      fName : string);
procedure ShowPic(    fName : string);
function NumOfPic : integer;
function NameOfPic(   num   : integer;
                    var fName : string);
{-----}
implementation
{-----}
end;
```

Для чего эти нелепые длинные комментария из одних тире – а это мне так нравится, и я всех так заставляю писать. Они, то есть сотрудники, сопротивляются, а я их ловлю и заставляю. Для чего буква «X» в начале имени модуля? Это уже серьёзнее. Если ваша программа будет работать с картинками, то, почти наверняка, вы будете использовать чужие модули. Чужие – не в смысле, что они будут вылезать из монитора и смачно откусывать вашу голову как в популярном кино, а в исконном смысле – модули, написанные не вами.

И, опять-таки почти наверняка, вы встретите модуль который кто-то непредусмотрительно называл именно Picture – ведь это просто очевидное имя. И программа наша немедленно перестанет транслироваться ввиду конфликта имён. Предусмотрим хоть какую-то защиту от дурака – пусть имя нашего модуля будет XPicture. А почему не XPictures, что было бы несколько естественнее? А потому что мы в ДОСе. Имя файла здесь ограничено восемью символами, так что лучше не выпендриваться.

А теперь по существу. А по существу – всё, что мы организовали в нашем модуле чрезвычайно похоже на объект, или если кому-то нравится называть его по-другому – класс. Об объектах и классах мы поговорим позже.

Второй, немного занудный момент – а почему, собственно, процедура, а не функция. Ведь можно было написать, и очень многие пишут, а некоторые даже советуют писать другим, вот так:

```
function AddPic(      fName : string) : integer;
```

И процедура возвращала бы нам результат нашей операции. Например 0 – всё хорошо, -1 – файл не найден, -2 – файл найден, но формат его не соответствует нашим ожиданиям, и так далее. Простой ответ - мне это не нравится. Ответ чуть посложнее – есть такая формула–заклинание – *Процедура не должна иметь побочных эффектов*. Это сказано не мной и давным-давно. Смысл этой мантры в том, что ум человеческий ограничен и причём крайне ограничен. Программисту невероятно сложно уследить даже за тем, что происходит в программируемой им сейчас процедуре, даже принимая во внимание то что передаётся и получается программистом через формальные параметры. А уж понять при этом, что такое происходит через результат процедуры, таких гениев просто нет.

На самом дел автор заклинания имел в виду, что из процедуры нельзя менять глобальные переменные объявленные вне процедуры. Но я искренне надеюсь, что вы и близко даже не понимаете о чём я говорю. И даже и не пытаетесь понять.

Прошу обратить внимание, что написанное нами транслироваться (пока что) ни под каким видом не будет. Причина понятна – в секции интерфейса (**interface**) прописаны заголовки процедур, а в секции реализации (**implementation**) процедуры эти начисто отсутствуют. Следующим шагом мы это исправим.

## Вариант 2

```
unit XPicture;  
{-----}  
interface  
procedure AddPic(      fName : string);  
procedure ShowPic(    fName : string);  
function NumOfPic : integer;
```

```

function NameOfPic(      num      : integer;
                    var fname   : string);
{-----}
inplementation
{-----}
procedure AddPic(      fname : string);
begin
end;
{-----}
procedure ShowPic(      fname : string);
begin
end;
{-----}
function NumOfPic : integer;
begin
end;
{-----}
function NameOfPic(      num      : integer;
                    var fName   : string);
begin
end;
{-----}
end;

```

А вот этот вариант программы безусловно можно транслировать. Это значит, что если ваш коллега сейчас пишет Большую Программу, которая будет использовать этот модуль, то он вполне может его подключить и работать дальше. Модуль, конечно, неработоспособен, но интерфейсы процедур и функций уже определены.

Кстати, о функциях, которых здесь ровно две. При трансляции на неё будет выдано предупреждение (warning) на предмет того, что возвращаемое значение функции не определено, и при обращении к ней результаты в лучшем случае будут непредсказуемы. А худшем случае – плачевны. Прошу обдумать на досуге, почему это не относится к процедурам.

Чтобы не изменять текст заново из-за такой мелочи, подумаем о вещах более серьёзных – а где собственно будут храниться наши загадочные картинки? В целом понятно – они будут храниться в динамически выделенной памяти, на которую будут указывать наши указатели (извините за каламбур). Но вот где будут храниться сами указатели?

Если бы это была книга о программировании на и под Delphi, то ответ был бы очевиден – указатели хранились бы, само собой, в списке. Это

потому, что стандартная библиотека Delphi предоставляет программисту готовый к использованию класс TList. В Turbo Pascal этого нет. Это есть в сторонних библиотеках для Паскаля, но с ними я связываться категорически не рекомендую. Почему не рекомендую, объясню, красочно и с примерами, в книге по Delphi, если я её допишу и в ней будет потребность у потребителя.

Хотя, с другой стороны, поскольку Паскаль сейчас язык в основном, процентов на девяносто учебный, можно применить и такую стороннюю библиотеку вроде Turbo Professional / Object Professional. Вреда никакого не будет. Ведь вы же пишете программы просто для развлечения, и продавать, а тем более – поддерживать их – не собираетесь, да?

Поэтому моё предложение такое – ссылки на картинки мы будем хранить в самом обычном массиве. Поскольку в Паскале при объявлении массива обязательно надо указать тип его элемента, этим типом элемента будет указатель. А поскольку это не список, который сам в себе хранит количество своих элементов, то нам его, количество элементов, придётся хранить непосредственно, в виде отдельной переменной.

Сразу надо решить, сколько максимально картинок мы будем хранить. Вопрос серьёзный, потому что в Turbo Pascal статически можно выделять не больше 64К памяти. Но там будут храниться не картинки! Ещё раз – не картинки там будут храниться! Там будут храниться имена картинок – 12 байтов (имя 8 + расширение 3 + точка). Если кто-то скажет, что точку хранить не надо, я его заочно стану презирать, и карма его безнадежно понизится. Ещё там будет храниться самое главное – указатели на картинки – 4 байта и размер картинок – ещё 4 байта. Итого 20 байтов на всё. Для начала я предлагаю ограничиться скромным количеством в 256 хранящихся изображений. Не забывайте, что основная память, в которой будут храниться сами картинки, тоже не резиновая.

И чтобы два раз не вставать. Заранее предусмотрим случай, когда нам захочется не только добавить в список картинку, но ещё и удалить её – а нам ведь захочется, правда? Нет, конкретно нам, никогда этого не захочется – ведь мы аккуратные и дисциплинированные люди. А захочется пользователям, которые вечно мельтешат и путаются под ногами. Поэтому теперь очередной вариант нашего модуля, со всеми уже накопившимися добавлениями:

### Вариант 3

```
unit XPicture;
{-----}
interface
  const
    maxPic = 256;
  var
    numOf          : integer;
    Ps             : array[1..maxPic] of pointer;
    Names          : array[1..maxPic] of string[12];
  procedure AddPic(   fName : string);
  procedure ShowPic(   fName : string);
  function NumOfPic : integer;
  function NameOfPic(   num : integer;
                      var fName : string);
{-----}
implementation
{-----}
  procedure AddPic(   fName : string);
  begin
  end;
{-----}
  procedure DeletePic(   nomer : integer);
  begin
  end;
{-----}
  procedure ShowPic(   fName : string);
  begin
  end;
{-----}
  function NumOfPic : integer;
  begin
  end;
{-----}
  function NameOfPic(   num : integer;
                      var fName : string);
  begin
  end;
{-----}
  initialization
    numOf:=0;
end;
```

Теперь неплохо обдумать, почему добавляем мы картинки по имени, а удаляем уже по номеру, и сравнить преимущества и недостатки того и другого способа. А также появилось несколько глобальных (для нашего модуля) переменных. Если по ним есть какие-то вопросы, то они разъяснятся на следующем варианте нашей программы.

А теперь возникает мелкая техническая проблема – а в каком виде хранить картинки в памяти – как они есть, то есть в хитром формате BMP Или преобразовать их в какой-то внутренний универсальный формат? Опять-таки предлагаю об этом поразмыслить на досуге, в свободное время, которого у вас конечно предостаточно. А поскольку у меня свободного времени экстремально мало, предлагаю не думать и применить вариант номер один. Картинка грузится в память как файл, не анализируя структуру файла. И как файл в памяти и хранится. *Как файл* означает – если, безо всяких выкрутасов и преобразований, картинку скопировать из памяти на диск – получим в точности то, что мы и загружали.

Итак, пошли по списку процедур. Первым номером – загрузить картинку, сохранить в памяти и добавить в наши списки. Или подробнее:

```
procedure AddPic(      fName : string);  
begin  
end;
```

Очень скучное но очень маленькое техническое отступление. С переходом от ДОС к Windows многие вещи значительно упростились. Например, намного проще стала работа с указателями. А некоторые, как легко догадаться, совсем наоборот. Как оно было в ДОСе – запускаем программу и открываем – точнее пытаемся открыть файл. В каком каталоге программа ищет файл? Странный вопрос в том – из которого она, программа, запущена. Или нет? Вопрос, конечно, интересный и, на самом деле, непростой. Вы позже много раз столкнётесь с этой проблемой. В любом случае, общий алгоритм:

- Перейти в текущий каталог
- Проверить, существует ли файл
- Узнать его размер
- Выделить память в соответствующем количестве
- Загрузить файл в эту память

И всё – освобождать память не надо!!! Мы ведь хотим, чтобы картинка наша оставалась в доступности на всё время выполнения программы.

Получается примерно так:

```

procedure AddPic(      fname : string);
  var
    F                : TFile;
    SR               : SearchRec;
    rez              : integer;
begin
  FindFirst( fname, Archive, SR);

  if rez = 0 then begin
    numOf:=numOf + 1;
    Names[numOf]:=SR.Name);
    Sizes[numOf]:=SR.Size;
    GetMem( Ps[numOf], SR.Size);
    Assign( F, SR.Name);
    ReSet( F, 1);
    BlockRead( F, Ps[numOf]^, SR.Size);
    Close(F);
  end;
end;

```

А теперь заглянем чуть дальше, за границы Турбо Паскаля. Из какого каталога/директории/папки будет загружен наш файл? Очевидно – из того, из которого запущена наша программа. Если файла там внезапно не окажется, мы будем разочарованы. Забегая вперед, в Windows всё не так. Файл система будет искать в текущем каталоге, который совсем не обязательно является тем самым из которого наша программа запущена. Поэтому в Delphi настоятельно рекомендуется первой строкой любой программы, работающей с файлами, сделать следующую:

```
ChDir(ExtractFilePath(ParamStr(0)));
```

Мы видим вызов процедуры, которая вызывает процедуру, которая вызывает процедуру. У последней процедуры, кстати, тоже есть параметр.

Поглядим, что они собственно делают, но не слева направо, а справа налево. Это очень скучно, но это совершенно необходимо, поэтому оно будет.

ParamStr – функция возвращающая один из параметров, с которым вызвана наша программа. Имеется в виду, что программа вызвана из командной строки что-то вроде pkzip -rP mumi. В этом случае ParamStr(1) вернет строку “-rP”, а ParamStr(2) вернет строку. “mumi”. Всё это для нас имеет сугубо исторический интерес, поскольку вряд ли придётся писать нам программу, параметры которой передаются через командную строку. Но вот параметр номер ноль имеет совершенно иной смысл – почему-то.

Последний параметр возвращает полный путь к запускаемой программе – в данном случае что-то вроде “c:\pkzip.exe”. Для чего нам это нужно? В нашем случае чтобы выковырять из этого полного имени нашей программы путь, где она, наша программа, лежит. А затем, путем применения процедуры ChDir туда в этот каталог и перейти – то есть сделать его рабочим. А что такое рабочий каталог было объяснено несколько раньше.

Поскольку писать этот ужас каждый раз заново совершенно невозможно, напишем это один раз и положим в отдельный модуль, который и будем прицеплять потом ко всем нашим программам, даже не раздумывая, нужен он там и ли нет. Примерно так:

```

unit OurLib;
{-----}
interface
{-----}
procedure GoHome;
{-----}
implementation
  uses
    Dos;
{-----}
procedure GoHome;
  var
    path      : PathStr;
    dir       : DirStr;
    name      : NameStr;
    ext       : ExtStr;

begin
  path:=ParamStr(0);
  FSplit( path, dir, name, ext);
  ChDir( path + dir);
end;
{-----}
end.

```

А надо ли это вообще? Скорее всего, не надо. В предыдущих главах вам встречалась милая пионерская программка, в которой половина переменных была не инициализирована – а программа несмотря на это работала, и хорошо работала. При запуске Паскаля память зачищается нулями. Тем не менее, если вам за программирование платят деньги, рассчитывать на такое везение я бы не стал. Так и здесь. Я бы не стал рассчитывать что каталог, из которого программа запущена, является рабочим. Одна манипуляция со свойствами ярлыка...

*Один меткий выстрел...* © Женьиба Фигаро.

Теперь пишем процедуру, удаляющую из нашей коллекции картинку. Она будет заметно проще.

```
procedure DeletePic(      ind : integer);
  var
    i          : integer;
begin
  if (ind>=1) and (ind<=numOf) do begin
    FreeMem( Ps[ind], Sizes[ind]);
    for i:=numOf downto ind+1 do begin
      Ps[i-1]:=Ps[i];
      Names[i-1]:=Names[i];
      Sizes[i-1]:=Sizes[i]
    end;
    numOf:=numOf-1;
  end;
end;
```

Проверку на корректность номера удаляемого элемента добавлять надо, не раздумывая, и не надеясь на порядочность пользователя. Со мной работала чудесная девушка, в смысле – красавица просто неопиcуемая, а ко всему ещё исключительно умная. Единственный её недостаток заключался в том, что, встретив в программе диаметр чего-то, она никогда не проверяла его на равенство нулю. «Но это же диаметр!» - объясняла она окружающим идиотам – «Диаметр всегда больше нуля!». Жизнь над ней периодически грязно надругивалась.

Обратите внимание на заполнение дыры в массивах, образовавшейся после удаления элемента. Делается это от конца массива к началу, в обратном порядке. Почему и зачем, рассказано в первой книге, а сейчас просто напоминаю. Почему-то все забывают.

Далее пишем элементарнейшие функции, возвращающие количество картинок и их имена. Даже объяснять нечего.

```
{-----}
function NumOfPic : integer;
begin
  NumOfPic:=numOf;
end;
{-----}
procedure NameOfPic(      num      : integer;
```

```

                var fname : string);
begin
    if (num>=1) and (num<=numOf) then begin
        fname:=Names[num];
    end;
end;
{-----}

```

А вот теперь действительно сложная вещь, выводящая на экран изображение в формате BMP. Причем нельзя сказать, что это особо никому не нужно. С одной стороны, в Дельфи есть, само собой, специальный компонент для этого. Но жизнь наша непредсказуема и богата сюрпризами, так что возможно, именно вам, бедной маленькой овечке, придётся программировать именно в той среде и под той системой, где этого компонента не то, что нет, но и про компоненты там вообще и не слыхали.

А главное, что за последние тридцать-сорок где-то лет формат файла \*.BMP не изменился. И вряд ли уже изменится. Это действительно вечное непреходящее знание. Так что берём книгу *Том Сван Форматы файлов Windows* от 1995, а у них изданную ещё в 1993-м – и вперед.

Ненужные технические детали. Файл БМП (или, по научному, файл растровой графики, состоит из заголовка и, собственно, пикселей. Заголовок состоит из трех структур. Первая структура с лаконичным именем BITMAPFILEHEADER занимает 14 байт и содержит всякую ерунду, из которой нас интересует четырехбайтовое целое. Оно находится в конце структуры и содержит смещение до собственно картинки (пикселей). Вторая структура BITMAPINFOHEADER со смещением 8788 содержит ширину и высоту изображения в пикселях (каждое из них двухбайтовое целое). Со смещением 8787 количество битов на пиксел. Нас интересуют только изображения у которых в этом поле стоит 24. Замечательная цитата из книги: *Этот необычный формат может описывает изображение с более чем 16-ю миллионами цветовых оттенков...БМП файл этого типа может занимать огромное пространство на диске и памяти. Немногие из дисплеев персональных компьютеров могут отображать так много цветов одновременно.* Чтобы всех запутать, изображение в БМП файле хранится по строкам слева направо, а сами строки идут снизу вверх. А ещё длины строк выравниваются на четыре байта – по-русски это значит, что длины строк добиваются нулями до величины кратной четырём байтам вверх.

Так вот, мы будем выводить такие и только такие файлы, хотя бы потому, что другие сейчас найти затруднительно. Ещё напомним, что лучшее графическое разрешение, которое есть у нас в Турбо Паскале – 480,360. А если картинка, что вполне вероятно, не влезет в такой размер? А ничего, сколько влезет столько и выведем.

Первый, приблизительный, вариант:

```

{-----}
procedure ShowPic(      fname : string);
  type
    TPixel = record
      R           : byte;
      G           : byte;
      B           : byte;
      W           : byte;
    end;
    TSLine = array[1..512] of TPixel;
  var
    F           : File;
    musor       : array[1..1024] of byte;
    offsPic     : longint;
    xSize,ySize : integer;
    bitPix      : integer;
    sLine       : TSLine;
    i           : integer;
{.....}
procedure graphCreate;
  var
    driver,mode : integer;
begin
  InitGraph( driver,mode, '' );
end;
{.....}
procedure graphShowLine(      sLine : TSLine);
  var
    pixel       : TPixel;
    size        : integer;
    bw          : byte;
    k           : integer;
begin
  size:=12; {size:=SizeOf(TPixel)}
  for k:=1 to {xSize mod 4} do begin
    Move( sLine[k-1], pixel, size);
    bw:= Round(0.30*pixel.R) +
         Round(0.59*pixel.G) +
         Round(0.11*pixel.B);
  end;
end;
{.....}
procedure graphFree;
begin

```

```

        readln;
        CloseGraph;
end;
{.....}
begin
    Assign( F, fname);
    ReSet( F, 1);

    BlockRead( F, musor, 10);
    BlockRead( F, offsPic, 4);
    BlockRead( F, xSize, 2);
    BlockRead( F, ySize, 2);
    BlockRead( F, musor, 2);
    BlockRead( F, bitPix, 2);

    if (bitPix = 24) and (xSize<=512) and (ySize<=512) then begin
        Seek( F, 0);
        BlockRead( F, musor, offsPic);

        graphCreate;

        for i:=1 to ySize do begin
            BlockRead( F, sLine, xSize*4);
            graphShowLine( sLine);
        end;

        GraphFree;
    end;

    Close( F);
end;
{-----}

```

### Комментарии:

Обратите внимание на тип TRixel, внутри являющийся записью. В этом нет ничего плохого и это абсолютно нормально. И всё будет хорошо – в Паскале. А вот в Дельфи здесь будут определенные проблемы. Но пока это неважно.

Три внутренних процедуры. Внутренние процедуры имеют доступ ко всем внешним, относительно них, переменным и, само собой, к внутренним переменным собственной процедуры.

Ещё раз – три внутренние процедуры. Но теперь не по форме, а по содержанию. Первая процедура инициализирует графический модуль, третья его закрывает, а вторая в промежутке между ними делает что-то полезное. Всё в совокупности образует модель объекта в масштабе 1:1. Но об этом позже.

А теперь о главном. Картинки наши, как уже было сказано, являются полноцветными. А Turbo Pascal, как легко догадаться, ни о какой полноцветности и не слышал. Попытка отобразить полноцветную (True Color) картинку в его жалкой палитре, закончится предсказуемо жалким результатом. Единственный наш шанс – перевести картинку в чёрно-белое изображение – хоть что-то приличное гарантированно получится.

Как перевести? По ФИДО и Интернету уже второе десятилетие гуляет замечательная формула  $Y = 0.3R + 0.59G + 0.11B$ . R, G, B, как легко догадаться – красная, зелёная и синяя составляющие цвета. Смысла формулы я не понимаю, но восхищаюсь и верю. А зачем четвертый байт кодировки пикселя? У него какой-то важный смысл, но, мне кажется, все о нем давно забыли.

Что сделать потом, на досуге?

Потом добавить в процедуру добавления картинок проверку на добавление картинок только полноцветных. Если мы не можем другие показать, зачем добавлять их в список и обманывать пользователя? Пользователь обидится и разобьёт монитор.

Попробовать всё-таки вывести в цвете. Если для каждой картинке задавать собственную палитру, может получиться не очень страшно. Задача, на самом деле, нетривиальная. Хотя, когда я в молодости программировал эротический тетрис... Ну это неважно.

Ну и масштабировать картинку что бы она влезала полностью на экран. Задача нетривиальная ещё больше.

В электрическом Интернете есть ещё много интересных картинок. Перед походом не забудьте поставить антивирус.

## Глава 2-5

### Указатели. Зачем они действительно нужны

#### А чем списки лучше массивов? А чем хуже?

Обратите внимание, вопрос о том, чем списки лучше массивов по сути сводится к тому, чем указатели лучше обычных переменных.

Сначала ситуация, в которой обычные переменные – то есть массивы – гораздо лучше. К примеру, мы храним информацию за сколько-то лет о чём-то, к примеру, о погоде. Во всяких такого рода примерах очень любят хранить информацию о погоде, ну и мы в этом отношении не будем ничуть оригинальнее.

Информация хранится за несколько лет, за каждый год по каждому месяцу, за каждый месяц по каждому дню. Обратите внимание, что недели нас не интересуют, и задумайтесь, почему. Объявляем вот такую иерархию массивов:

```
const
  mDay   = 31;
  mMonth = 12;

type
  TdayRec = packed record
    temp      : single;
    press     : single;
  end;

type
  TMonth   = array[1..mDay]   of TDayRec;
  TYear    = array[1..mMonth] of TMonth;
  TCentury = array[1..100]   of TYear;
```

Что мы имеем в результате? Мы имеем массив размерностью 10x12x31, что составляет ровно 37,20 элементов массива. Каждый элемент занимает своим телом 8 байт, дальше мне даже считать лень, потому что сразу видно, что объём всего конгломерата не превышает 400 килобайт. Величина жалкая и ничтожная. Четыреста килобайт, по нынешним временам, это мягко говоря, не деньги, и считать их нечего.

Но важнее даже не это. В нашем массиве есть неиспользуемые в принципе элементы. Сразу понятно, что не все месяцы имеют в наличии

тридцать один день, бывают и короче, а мы уже зарезервировали под эти лишние дни место. Место пропадёт, это несомненно. Программисты старой школы в таких случаях начинали рассуждения от том, что важнее – экономия памяти или скорость работы программы. Мы совсем не программисты старой школы, мы теперь рассуждаем совсем по другому – что важнее – экономия памяти или экономия наших программистских умственных усилий. Потому что иметь дело с красивыми ровными прямоугольными массивами намного проще. Ради этого можно пожертвовать памятью. Да и быстродействием, честно говоря.

А какой у нас вообще был выбор? А выбор был. Мы могли бы предпочесть использование списков. В чем их преимущество? Преимущество очевидно – никаких потерь памяти, вся выделенная память использована до последнего байта. Само собой, будут кое какие накладные расходы, но без этого уже никак не обойтись.

Как бы это выглядело? На словах – вот так. Верхний список хранит ссылки на годы – тем самым, кстати, мы избавляемся от фиксации на ста годах. Теперь можно хранить сколько угодно – хоть меньше, хоть больше. Каждый элемент этого списка хранит списки месяцев, а каждый элемент списка месяцев хранит списки дней – тем самым мы экономим на неполных (меньше тридцати одного дня) месяцах.

Злые и завистливые люди могут заметить, что в каждом году ровно двенадцать месяцев и на этом можно было бы сэкономить – сэкономить не в смысле экономии памяти, ну её, а смысле экономии программистского труда. В чём-то злые люди, конечно, правы. Вместо трех уровней списков – хранящего годы, месяцы и дни – можно ограничиться всего двумя уровнями – год и день. Но список по годам в таком случае не должен хранить непосредственно дни, он должен хранить списки по дням, организованные по месяцам. Непонятно? Сейчас объясним.

### **Всё то же самое, но медленно и по шагам. Шаг первый**

Мы об указателях уже очень много говорили, но теперь ещё раз применим на практике.

Хранилище для данных за один месяц.

Массив. Всё очень и очень просто:

Объявление типа

```
type  
    TMonth      = array[1..mDay] of TDayRec;
```

Объявление переменной:

```
var  
    m           : TMonth;
```

Использование этой переменной на запись, за седьмое число неизвестного нам месяца:

```
m[7] .temp:=-51;           // ну очень холодно
```

Использование на чтение:

```
thisTemp:=m[7].temp;
```

Всё просто до безобразия, вопросов не вызывает и вызвать никаких вопросов просто не может. А теперь то же самое, но со списками. Списки будем использовать готовые, из Delphi. Что у них там внутри, у дельфийских списков, мне даже говорить стыдно. Вообще-то, во времена моей программистской юности, программирование простейшего списка считалось классической учебной задачей в процессе дрессировки юного программиста. Допускалось использовать при этом только указатели и записи. Записи – это в качестве поощрения и облегчения задачи. Так вот, внутри списка из Delphi – банальный, но огромный массив.

*Но что нам в том?* © А.С. Пушкин

Ссылка на модуль и объявление переменной:

```
uses  
    Classes;  
var  
    L           : TList;
```

Но теперь нашу переменную L просто так, сразу, использовать нельзя. Увы. Поскольку TList это класс, нам необходимо создать экземпляр класса. Вот так:

```
L:=TList.Create;
// в конце нашей программы должна присутствовать строка L.Free,
// но мы сейчас не об этом
```

Точно так же, нельзя просто так присвоить списку нашу запись. Дельфийские списки получают на вход указатели, поэтому нам придется выполнить ряд малоинтересных манипуляций, как то:

Объявить нашу запись.

Обратите внимание – запись для одного дня, а не массив записей для всего месяца

Объявить указатель

выделить для него память

отправить в эту память содержимое нашей записи:

```
var
    rec           : TDayRec;
    p             : pointer;
begin
    // первая запись
    GetMem( p, SizeOf( TDayRec));
    Rec.temp:=-52; // вот такой я добрый
    Rec.press:=700;
    Move ( rec, p^, SizeOf(rec));
    L.Add(p);

    // .....
    // последняя запись
    GetMem( p, SizeOf( TDayRec));
    Rec.temp:=50;
    Rec.press:=690;
    Move ( rec, p^, SizeOf(rec));
    L.Add(p);
```

Необходимые комментарии. Необходимость комментариев отчасти связана с желанием пояснить, как вообще работают списки. А с другой стороны отчасти с тем, что списки в Дельфи не совсем классические, правильные списки. Внутри каждого дельфийского списка на самом деле находится гигантский одномерный массив!

Вначале мы объявляем нетипированный указатель. Напоминаю, нетипированный указатель, в отличие от типированного, указывает на что угодно. Типированный указатель способен указывать только на переменную одного, конкретного типа – integer, single, boolean. Практической ценности это счастье не имеет.

Выделяем память для этого указателя. Количество памяти равно размеру нашей записи. Заполняем нашу запись особо ценными данными. Отправляем запись с этими данными по адресу памяти, связанному с указателем. И – далее – собственно работа со списком. А именно, указатель на эту память добавляем в список.

Обратите внимание на два момента. Первый – в отличие от массива я не указываю номер, под каким наш указатель (указывающий на запись) окажется в списке. Потому что я его не знаю. То есть, догадываться приблизительно конечно могу, но точно не знаю. Второй момент – об указателях и о том, куда они указывают я всё время говоря разными словами и расплывчато. Это специально – чтобы вы поняли, что все упомянутые термины - это одно и то же.

Идём дальше. Теперь очень важный момент. Память мы выделили, в список добавили, но освободить её не собираемся. Между тем, я всегда учу, что программа должна быть симметричной. Здесь же у нас память выделяется много раз, причем для одного и того же указателя, но так ни разу и не освобождается, даже в конце программы Почему?

Потому что, выделив память первый раз, мы перелазим указатель на эту память в список. С этого момента за эту память отвечает не наша основная программа (процедура), а этот самый список. Об этой памяти наша программа должна забыть. Когда придет время эту память освободить, этим должна будет заняться не наша программа, а сам список.

При втором и дальнейших распределениях памяти и добавлении в список все ещё проще. Надо сосредоточиться и понять, что это совсем другая память. Точнее, память конечно та же самая, физически и логически, но вот область памяти это уже совсем другая.

О тонкой разнице между  $p$  и  $p^{\wedge}$  мы уже говорили. Это и есть то самое, чего половина программистов не понимает. И это не потому, что я очень умный. Увы, это по совсем другой причине...

Но, раз мы данные где-то сохранили, то это ведь не просто для того, чтобы от этих данных избавиться и навсегда о них забыть. Данные эти нам

ещё понадобятся и мы должны быть способны эти данные из их хранилища извлечь и использовать. На этом этапе вы встречаемся с другой очаровательной операцией, на которой дохнет ещё половина программистов и программисток.

И зовут её, не программистку, а операцию, - разыменование указателя. Как ранее многократно я упомянул, указатель наш – нетипированный, то есть указывает непонятно на что. Точнее, когда мы в память, выделенную этим указателем, что-то отправляем, мы знаем, что именно это было. А потом все следы этой информации, увы, теряются. Давайте сначала опробуем технологию на простом примере.

```
var
    P                : pointer
    Int1             : integer;
    Int2             : integer;
begin
    GetMem( p, 4);
    Int1:=44;
    Move( int1, p^, 4);
    // .....
    // происходят какие-то ужасы
    // .....
    // а теперь нам понадобилось сохранённое значение
    // можно по простому
    Move( p^, int2, 4);
    Writeln( 'int2 = ', int2);
```

Всё хорошо. А теперь вообразим ситуацию, когда переменной int2 просто не существует в природе. Кажется невероятным? Да, если всё объявлено именно так, то действительно да. А если нет, то, возможно, и нет.

Что делать, если нам надо получить значение переменной, на которую указывает указатель, не используя переменной этого типа? А вот в этом случае и совершается разыменование указателя. Выглядит это вот так, тот же вывод на экран, но без переменной:

```
Writeln( 'int2 = ', Integer(p^));
```

Integer(p^) как бы намекает компилятору, что те четыре байта, на которые указывает наш указатель, содержат в себе целое число, по чистой случайности и занимающее в памяти эти самые четыре байта.

И, в очередной раз, повторяю: `p` – указатель, содержит некоторый адрес в памяти, непосредственному использованию для вычислений и вывода не пригоден. `p^` - область памяти, на которую этот указатель указывает, в данном случае область эта содержит в себе целое число типа `integer`, с которым уже можно делать всё, что только можно делать с целым числом.

Обратите внимание – `Integer(p^)` – слово `Integer` с большой буквы. В Паскале, как всем известно, большие и малые буквы абсолютно равнозначны. Но. В случаях разыменования указателей есть традиция имена типов писать с большой буквы. А традиции я уважаю.

// Прочтите и забудьте

Приведение типов – а по-английски `typecast`, именно так называется по научному то, чем мы сейчас и занимались. На самом деле, операция эта никакой непосредственной и обязательной связи именно с указателями не имеет, потенциальная сфера приложения этой операции гораздо шире.

Приведение типов позволяет обращаться с переменной одного типа таким образом, как если бы она, переменная принадлежала совсем к другому типу. Главное, чтобы размер совпадал, в байтах, в смысле.

*Главное, чтобы костюмчик сидел* © Чей, дело тёмное

Зачем это надо? В общем и целом, это совершенно не надо. А в тех случаях когда у нас возникают какие-то такого рода потребности, мы вполне можем использовать функции `Chr` и `Ord`.

Но иногда всё-таки надо. Пока всё.

// конец Прочтите и забудьте

Итак, мы сохранили в списке наши записи, и даже ухитрились благополучно извлечь и применить.

## **Всё то же самое, но медленно и по шагам. Шаг второй**

Хранилище для данных за один год. Типы мы уже объявили, давным-давно:

```
const
  mDay   = 31;
  mMonth = 12;

type
```

```
TMonth    = array[1..mDay]    of TDayRec;  
TYear     = array[1..mMonth] of TMonth;
```

А теперь, опять-таки всё будет очень и очень просто. Если вас огорчает, что совершенно не на чем продемонстрировать, как блещет разум ваш чудесный, не скучайте. Потом мы перейдём к тому же, но с применением списков, и разум ваш мгновенно потухнет. И погаснет.

Объявление переменной:

```
var  
  c                : TYear;
```

Использование этой переменной на запись, за седьмое число третьего месяца неизвестного и несохраняемого года:

```
c[3,7].temp:=+51; // ну вот такая температура
```

Использование на чтение:

```
thisTemp:=c[3,7].temp;
```

Как я и обещал, всё удивительно просто. Другого трудно было бы и ожидать, с массивами всегда всё просто, по крайней мере мне так кажется. Говоря между нами, я сам всегда использую только массивы, если в этом есть хоть малейшая целесообразность, само собой.

А вот теперь мы переходим к спискам. Разумеется, речь не идёт об унылом списке, указывающем на конкретный день по формуле номерВсписке = день + (месяц-1)\*12. Потом из всего этого вычесть единицу, поскольку списки в Дельфи начинаются с нулевого индекса.

Нет, так, конечно, сделать можно тоже, но мы пойдем трудным путём. Мы создадим список, элементом (Item) которого будет не указатель на конкретную переменную, а указатель на другой список. Формально я сейчас сказал банальность, или, говоря сильнее, вообще ничего сейчас не сказал. По той простой причине, что список сам тоже является переменной, и что в каком-то смысле приятно, как и все дельфийские объекты, является указателем – соответственно для его помещения и извлечения в первый, верхний список не требуются никакие

дополнительные манипуляции. Но вы, надеюсь, поняли, что я на самом деле имел в виду.

*Вы следите за моими мыслями?  
Вы следите, мне самому трудно.*

Итак, каждым элементом нашего верхнего (назовем его так) списка будет нижний (назовём его так) список, в котором уже будут храниться конкретные записи с конкретными данными.

Вам кажется, что мы занимаемся ерундой, и всё это можно сделать гораздо проще, и вы даже знаете, как именно? Вы правы.

Но, как учит нас великий кинофильм всех времён и народов – *Тренируйся на кошках!* Вот мы сейчас на них и тренируемся. В роли кошки в роли сторожа – сложный список в роли заместителя двумерного массива. Проще не бывает, уж поверьте мне.

А теперь – вечная триада. Объявить всё что надо, записать прочитать. Напоминаю, пользуемся стандартными средствами Delphi.

```
uses
    Classes;
var
    L           : TList;
    L2          : TList;
    dr, drOut   : TDayRec;
    p           : pointer;
    i, k        : integer;
```

Комментарии. В секции **uses** объявлен дельфийский модуль **Classes**. Он необходим здесь для того, чтобы далее иметь возможность использовать тип **TList**, что мы далее и делаем. **L** – наш самый верхний список, он один, один он и объявлен. Это понятно. **L2** – список нижний, второго уровня. Их много, конкретно в этом случае ровно двенадцать штук, но объявлен всё равно один. Если понятно, то очень хорошо. Если непонятно, подумайте. Если не помогло, то ситуация прояснится чуть позже.

Создаём верхний список. В цикле создаём двенадцать (по месяцам, несложно догадаться) списков нижнего уровня и добавляем их в качестве элементов (Item) в верхний список.

```
for i:=1 to 12 do begin
  L2:=TList.Create;
  L.Add(L2);
end;
```

Жизненно необходимые комментарии. Не все написанный выше текст понимают правильно. Некоторые понимают неправильно, а многие, вы даже и не поверите, не понимают вообще.

Первое, крупное, непонимание. В список L двенадцать раз добавляется переменная L2, так же, по случайному совпадению, являющаяся списком. Почему мы двенадцать раз тупо добавляем одно и то же в список? Потому что это не одно и то же. Мы двенадцать раз создаём объект L2, каждый раз совершенно новый, само собой, и двенадцать раз добавляем двенадцать разных объектов в список L.

Второе, даже не непонимание, а, скорее, нюанс. Обычно, когда мы что-то хотим добавить в список, мы, под это что-то, сначала выделяем память а потом добавляем в список указатель, на эту самую память указывающий. Это самая что ни на есть стандартная технология. Сейчас же мы добавляем нашу переменную молча, с суровым лицом. Никакой памяти при это м не выделяя. Почему?

Чисто техническая причина – список хранит указатели. Указатель занимает четыре байта. Ровно столько же занимает целое типа integer или плавающее типа single. Поэтому хранить в списках их легко и просто. Приходится выполнить только несложную манипуляцию под названием приведение типа. Где-то так:

```
var
  Int           : integer;
begin
  // туда
  L.Add(Pointer(Int));

  // обратно
  int:=Integer(L.Item[nomer]);
```

Утомительно, но несложно. Проверьте.

Но мы не делаем даже этого. Причина проста – список хранит указатели. Но и сам список является указателем, точно так же, как и все объекты в Delphi. Поэтому хранить списки в списке можно безо всяких распределений памяти и преобразований типов.

А теперь заполняем нашу хитрую конструкцию значениями. Для простоты дальнейшей проверки алгоритм заполнения несложен – поле temp равно значению месяца, а поле press – дню.

```
for i:=1 to 12 do begin
  for k:=1 to 30 do begin
    dr.temp:=i;
    dr.press:=k;
    GetMem(p, SizeOf(dr));
    Move( dr, p^, SizeOf(dr));
    TList( L[i-1] ).Add(p);
  end;
end;
```

Вот здесь уже добавление идёт каноническим образом – сначала выделяем память, привязанную к указателю P, затем отправляем туда значение нашей записи DR, и добавляем указатель в список.

Провокационный вопрос – мы добавляем в список одну переменную - указатель. А почему не добавлять в список, просто и без затей, DR, переменную-запись? Подумайте сами.

Итак, хитрый двухэтажный список мы создали. Данные в него записали. Теперь осталось эти данные из него извлечь, и, напоследок, всё это уничтожить, чтобы и следа не осталось – в оперативной памяти, я имею в виду.

Я хочу выковырять из списка данные за седьмой день восьмого месяца. Как нетрудно догадаться, на выходе мы должны иметь циферки 7 и 8 соответственно. Всё очень просто – первая строка, короткая, чтобы извлечь данные из списка, вторая строка, подлиннее, выводит их на экран.

```
drOut:=TDayRec( TList( L[6] ).Items[7]^ );
ShowMessage( 'month = ' + FloatToStr(drOut.temp) +
```

```
' day = ' + FloatToStr(drOut.press));
```

Почему, желая получить 7 и 8, мы запрашиваем 6 и 7 соответственно? А потому, что списки традиционно имеют нумерацию элементов, начиная с нуля, то есть, желая получить седьмой по порядку элемент, мы должны обращаться к шестому элементу списка. Это мелочи, не стоящие нашего внимания.

А теперь давайте приглядимся внимательно к этой строке и попытаемся понять, как я её написал. Нам нужны данные за седьмой месяц и восьмой день. Это означает, что мы должны взять седьмой по счету (но шестой по индексу) нижний список, хранящийся в верхнем списке. А из нижнего списка взять восьмой (по индексу седьмой) элемент.

Как записать, что нам нужен шестой элемент верхнего списка? Просто – `L[6]`. А как разъяснить компилятору, что это не абстрактный указатель на область памяти, а указатель на переменную определенного типа, в нашем конкретном случае – опять же список? Ведь вместе со списком эта информация не хранится. Хранится она только в голове программиста. Тоже почти просто – явным образом выполнить приведение типа, вот так – `TList(L[6])`. Вот это вот выражение и представляет собой живо нас интересующий список нижнего уровня. Теперь мы должны получить из него элемент с индексом 7. Это несложно – `TList(L[6]).Items[7]`. Получили.

Есть только небольшой нюанс – мы получили опять-таки нетипированный указатель, указывающий непонятно на что. Теперь, чтобы получить не просто указатель, а область памяти, на которую он ссылается, мы должны выполнить следующую нехитрую манипуляцию – добавить в наше выражение всего одну закорючку, вот так – `TList(L[6]).Items[7]^`. Эта штучка с крышечкой и называется операцией разъименования указателя. Говоря точнее, без крышечки мы имели в качестве значения выражения адрес, по которому указатель находится. А теперь мы имеем значение, хранящееся в указателе, то есть адрес памяти, на который указатель ссылается.

А указывает наш указатель на запись, которая содержит данные о погоде за один день. Но, опять-таки, как и сказано раньше, указатель ни малейшего понятия не имеет, на что именно он указывает. И нам надо объяснить транслятору, что указывает указатель именно что на запись

типа TDayRec. Получается вот так - TDayRec( TList( L[6] ).Items[7]^ ). Собственно, именно с этого мы и начали, а сейчас просто пришли к этому медленно и осторожными шагами.

Запрограммируйте, проверьте и убедитесь, что оно вроде как работает *Вроде как* - потому что никогда до конца уверенным быть нельзя. В программировании точно нельзя.

А вот так мы это всё уничтожим:

```
for i:=1 to L.Count do begin
  for k:=1 to TList( L[i-1] ).Count do
    FreeMem( TList( L[i-1] ).Items[k-1], SizeOf(dr) );
  TList( L[i-1] ).Clear;
  TList( L[i-1] ).Free;
end;

L.Clear;
L.Free;
```

Две последние строчки понятны – мы очищаем и уничтожаем список. Перед этим на всякий случай его очищаем. Это нетрудно, всего одна строка. Вроде бы этого можно и не делать, справка Delphi уверяет, что всё будет хорошо и так – то есть, все элементы списка будут из него удалены перед ликвидацией этого самого списка. Так что мы просто перестраховываемся.

Но, в той же справке, многократно и черным по-английски написано, что метод Clear, а также дружественные ему Delete и кто-то ещё, указатели из списка удаляют – это святое, а иначе, зачем эти методы вообще нужны. А вот память, на которую эти указатели указывают, никто за программиста освобождать не будет. Если об этом не позаботиться – вот этими вот самыми руками – то распределенная нами ранее память перейдёт в категорию *мусор*. И упрекать за это транслятор никак не возможно – список не помнит, указатели на что именно в нём хранятся, и сколько именно памяти связано с конкретным указателем. Список помнит только, где эта память начинается.

Потому мы должны и обязаны освобождать ранее выделенную память вручную, при помощи FreeMem. Получается громоздко, но деваться некуда.

```
FreeMem( TList( L[i-1]).Items[k-1], SizeOf(dr) );
```

Разумеется, очень ленивые люди пишут свои версии списков, которые хранят информацию о размере данных внутри себя и сами освобождают память при ликвидации. А также сами её, память, и выделяют. Но для этого надо иметь хотя бы самое элементарное представление об Объектно Ориентированном Программировании.

А пока зададим себе вопрос и сами себе дадим на него ответ. Что в данном случае проще, удобнее, и вообще лучше? Если выбирать между двумерным массивом и двухуровневым списком?

*При всём богатстве выбора, другой альтернативы нет!* (С)  
Черномырдин.

Да, массив лучше. Проще, надёжнее и, главное, понятнее. Не вижу у списка ни малейшего преимущества. Кроме грошовой экономии памяти. Так зачем же мы сейчас занимались тем, чем мы сейчас занимались? Затем, чтобы на простом, и элементарном, и насквозь понятном примере понять, как же оно работает. А то, что список не даёт ни малейших преимуществ, так это ведь относится только к нашей конкретной задаче. В этой ситуации не даёт, а в другой очень даже и даёт.

Сейчас рассмотрим задачу сложнее.

### Усложняем. Шаг третий

А теперь рассмотрим случай сложнее. Пусть мы оформляем информацию по городским адресам. До этого мы имели в качестве уровней, сверху вниз – год, месяц, день. А теперь у нас будет – улица, дом, квартира. Сразу видим некоторую разницу. Если количество дней в месяце почти одинаково, и так же почти одинаково количество, не то что месяцев, а даже дней в году, то с городской хаотической застройкой ситуация сильно другая.

В нашем Городе, имеющем принципиально нерегулярный и романтически непредсказуемый характер, на одной улице запросто может находиться и панельная девятиэтажка на двести шестнадцать квартир и старый деревянный памятник архитектуры конца восемнадцатого века ровно на одного хозяина.

Ну и что же мы будем иметь в качестве размеров массива? Массив трехмерный, первое измерение – улицы, второе – дома на улице, третье – квартиры в доме. Начнём с конца.

Сколько может быть квартир в доме – не в среднем, а максимально? Двести шестнадцать – это не предел, в доме всего шесть подъездов. Ходят смутные предания о стоящих в тумане на краю города домах с десятью, двенадцатью и даже шестнадцатью подъездами. Хватит ли тысячи квартир на дом? Не уверен. Надо закладываться сразу на две. Или, по-нашему, по-программистки – максимальное число квартир будет 2048.

Теперь о количестве домов на улице. Количество начинается – не от единицы – а от нуля. В нашем Городе есть ненулевое количество улиц, на которых ровно ноль домов. Одна из них даже названа в мою честь – *Коллин тупик*. Источник информации – *К.В.Литвицкий Энциклопедия тверских улиц*. Максимальное количество – ну, допустим двести пядесяят шесть. Тут мы сразу имеем две мелкие проблемы.

Двести с лишним домов на улице – это в нашем гордом, но скромном городе. В других городах, распухших и разжиревших на наших деньгах, городах домов может быть и сильно больше. А вдруг мы напишем гениальную программу? А вдруг её все захотят купить? И даже покупатели из лучшего города на земле? А сколько у них там домов на улицах, даже страшно и подумать.

Второй нюанс – не знаю, как у вас, а нас, на некоторых и даже многих улицах больше половины номеров не используется. Итого, если мы зарезервируем номера домов по максимуму, да ещё часть их использоваться не будет, то процент затребованной, но не используемой, памяти становится весьма значительным.

А сколько у нас в городе улиц? А я не знаю, наверное, много. А в столицах нашей родины улиц гораздо больше. А в посёлке Выползово гораздо, гораздо меньше. А где-то улица всего одна. Ещё более где-то их, улиц, нет вообще, но мы это игнорируем, поскольку это портит нашу стройную картину и нам пока совершенно не интересно. И не забывайте, мы с самого начала планируем создать гениальную программу, которую купят все, от городов побольше, до поселков поменьше, то есть с самым

разнообразным количество улиц. Иными словами, если раньше в определениях максимально допустимых значений для индексов нашего массива был некоторый смысл, сейчас он исчез и растаял как дым над водой.

Обратим внимание вот ещё на какой занимательный момент. Нумерация квартир идёт более-менее подряд. Это означает, что номер квартиры может с удобством служить и индексом массива, ну и наоборот, понятное дело. С номерами домов уже сложнее, потому что, чем новее улица, тем большая часть номеров не задействована – ну нет таких домов пока. А может и никогда не будет. То есть использовать номера в качестве индекса можно, но как-то не очень экономно, хотя и не критично.

А улицами всё совсем печально. Номеров у улиц нет, есть имена, а их использовать в качестве индексов теоретически можно, но практически крайне затруднительно. Найти улицу по названию можно только тупым построчным поиском и проверкой имён на соответствие.

Это я плавно подвожу вас к мысли, что нет у нас другого выхода, кроме как использовать списки. Задача наша усложняется тем, что в отличии от хранения в списках данных по месяцам и дням, нельзя просто хранить в верхнем списке сами списки нижнего уровня.. Точнее для домов и квартир это можно, с незначительными потерями памяти, а вот с улицами так уже не получится, ведь где-то должны храниться и названия улиц.

Как быть? Первый вариант – верхний список имеет своими элементами (Item) записи. Запись содержит всю информацию об улице, которую нам надо хранить, плюс, в конце указатель на привязанный к этой улице нижний список с номерами домов. Мне это как-то не очень нравится, даже не могу сразу сформулировать почему. Может быть, говоря по-умному, потому, что хранимая информация беспощадно смешивается с метainформацией о структурах хранимых данных?

Я предлагаю другой вариант, с одной стороны несколько более громоздкий, с другой стороны, данные у нас, будут отдельно, а организация данных – совсем отдельно. То есть списков верхнего уровня, который один, у нас на самом деле будет два. Перевожу с русского на русский. Создаём список, в нём будут храниться просто записи, содержащие информация по улицам. Создаём еще один список, который

будет хранить указатели на другие списки. А те, другие, списки уже и будут хранить номера домов.

Кстати, попутно, а какого типа будет переменная с номером дома? Целое не предлагать. Не отходя от моего дома и на двести метров, имеем следующее – «44», «44/2», «44 корпус 1». Увы, номер дома обречён быть строкой. Возникает мысль – пусть верхний список будет просто список, а нижний список будет то, что в Delphi называется TStringList.

Напоминаю, если вы случайно не знали, TStringList – это такой список, который специально предназначен для хранения строк. В обычном списке строки хранить нельзя. Точнее, хранить-то их можно, но вот сохранить в файл уже точно нельзя. Причина – строки в Delphi на самом деле являются указателями и где-то почти псевдоклассами. А если сегодня сохранить указатель в файл, а завтра из файла прочитать - то куда, собственно, этот указатель будет указывать? Ну да, именно туда и будет.

И вот именно для таких случаев и создан TStringList – он содержит строки, сохраняет их в файл и восстанавливает оттуда. Кроме того, он имеет ряд полезных для работы со строками возможностей. И вроде бы он нам подходит. Но! Но что будет, если в процессе применения программы потребности пользователя постепенно повысятся? Чего они могут захотеть? Да мало ли чего – хранить вместе с номером дома количество подъездов, этажей, цвет фасада и кое-что ещё. Предлагаю на всякий случай хранить в нижнем списке записи с информацией по дому. Номер дома в этой записи будет самой важной информацией и самой первой, но отнюдь не единственной.

Такая структура кажется вам громоздкой? И мне тоже. Но, к сожалению, она лучшая из всех возможных.

- *Лёлик, но это же не эстетично...*
- *Зато дешевле, надёжно и практично.*

(С) Бриллиантовая рука

А теперь быстро всё то, о чём я только что говорил, но в очень сокращённом виде. Но в виде программного кода.

Объявляем всё необходимое:

```

var
  Lstr          : TList;
  LstrBui       : TList;
  Lbui          : TList;
  strRec       : TStreetRec;
  buiRec       : TBuildingRec;
  p            : pointer;

```

А что мы такое наобъявляли, будет сейчас понятно.

```

Lstr:=TList.Create;
LstrBui:=TList.Create;

```

Lstr – список, который хранит информацию об улицах – потенциально любую информацию, кроме информации о дамах, которые на этой улице находятся. Информация о домах будет храниться в списке LstrBui – точнее, в этом списке будут храниться списки со списками домов по каждой улице. Что важно, индексация списков Lstr и Lstrbui должна быть синхронной. То есть, Lstr[3] указывает на информацию по улице номер четыре, в частности название улицы. Почему улица четвёртая, а не третья, вы, конечно, отлично понимаете. А LstrBui хранит список домов по этой же самой улице.

Списки созданы, но пока что они пусты. Теперь добавляем в первый список общую информацию по улице:

```

strRec.name:='Пушкина';
strRec.numOfChurches:=0;
strRec.numOfSynagogues:=0;
GetMem( p, SizeOf(strRec));
Move( strRec, p^, SizeOf(strRec));
Lstr.Add(p);

```

Это просто. А теперь во второй список добавляем список второго уровня, который будет содержать сведения о домах по этой улице:

```

Lbui:=TList.Create;
LstrBui.Add(Lbui);

```

А теперь добавляем дом по первой улице. Добавляем мы его в список домов по первой улице. Обращая внимание в бесконечно какой раз, что список домов по первой улице является элементом списка списков домов

по всем улицам. Причём имеет в этом списке индекс ноль – потому что нумерация в списках вообще начинается с нуля. А кому легко?

```
buiRec.number:='5/25';
buiRec.color:=clYellow;
GetMem( p, SizeOf(buiRec));
Move( buiRec, p^, SizeOf(buiRec));
TList(LstrBui[0]).Add(p);
```

Добавьте от себя ещё несколько домов на ту же улицу, совершенно аналогичным способом.

А теперь самостоятельное задание сложнее – добавьте ещё одну улицу и дома по ней.

Теперь проверочное задание – попробуем прочитать что-нибудь из того, что мы записали. Узнаём количество улиц. Это очень просто, потому что количество улиц равно количеству элементов в самом первом списке. И во втором списке тоже, между прочим.

```
numStr:=Lstr.Count;
ShowMessage('numOfStr = ' + IntToStr(numStr));
```

Теперь узнаем и выведем на экран название первой улицы:

```
strRec:=TStreetRec(Lstr[0]^);
ShowMessage( strRec.name);
```

Чуть сложнее, но, всё-таки, ничего особенного сложного. Берём нулевой элемент списка – это указатель. Производим разыменованье, это птичка сверху. Теперь вместо указателя мы имеем область памяти, на которую этот указатель указывает. Но эта область памяти пока что является серой массой унылых байтов, и, чтобы её структурировать, производим то, что по-английски называется *typecasting*, а по-русски – приведение типа. Эта операция накладывает на бесформенную массу шаблон – структуру записи `TStreetRec`, после чего, мы имеем полное право обращаться к полям этой записи. При желании можно было написать короче, но непонятнее:

```
ShowMessage(TStreetRec(Lstr[0]^).name);
```

Заодно сэкономили на объявлении переменной. Хотя, лично мне больше нравится когда длиннее, но понятнее, как и было в начале.

Теперь извлекаем количество домов по второй улице. Для этого мы используем список `Lstrbui`, который содержит списки нижнего уровня, каждый из которых хранит информацию по конкретному дому, относящемуся к этой улице.

Изложение у меня получается унылым, тоскливым и нудным, но это неизбежно. Программист, по определению, обязан быть унылым, тоскливым и нудным. А самой главное – программист не имеет права быть нервным. Нервные программисты долго не живут, в буквальном смысле слова. И ещё, чуть не забыл – программист при этом при всём должен быть общительным, коммуникабельным и всем улыбаться. Забудьте бред о программистах-аутистах-шизофрениках. Таких экземпляров не бывает. Мы, программисты, на самом деле белые и пушистые, но очень, очень спокойные. Как удав. И не вздумайте называть нас земляными червяками.

Извините, отвлекся. Вот что у нас получается в итоге.

```
ShowMessage( 'Домов на второй улице ' + IntToStr(
TList(LstrBui[1]).Count ));
```

А теперь, чтобы домучить пример до конца, получим и выведем сведения по второму дому второй улицы:

```
buiRec:=TBuildingRec( TList(LstrBui[1]).Items[1]^ );
ShowMessage('Второй дом на второй улице ' + buiRec.number);
```

## Дополнение Всякие важные вещи

### Как установить Турбо Паскаль

Прежде, чем установить Турбо Паскаль, надо его где-нибудь взять. Где взять? Помните великого ирландско-английского драматурга Оскара Уайльда (Oscar Wilde) и его пьесу *Как важно быть серьёзным* (*The Importance of Being Earnest*)? Джентльмен посылает слугу купить огурцов, тот возвращается без них, и отвечает:

*- Сэр, сегодня на рынке не было огурцов. Даже за деньги!*

Некоторые вещи купить нельзя даже за деньги, например DOS 5.0 или Турбо Паскаль. Как быть? А если очень хочется? Как в анекдоте – *Где взял, где взял?! Нашёл!* Настоятельно рекомендуется найти Turbo Pascal 7.0. Если это будет Borland Pascal 7.0 – всё равно. В нудные разъяснения об их отличиях вдаваться не будем. Если версия окажется 7.01 – не страшно, а даже лучше. На самом деле, такой версии нет, это творение местных умельцев.

Короче, нашли вы где-то Турбо Паскаль. Теперь надо установить. Нашли вы или дистрибутив, или готовый образ. Готовый образ – это значит - у приятеля уже установлен и даже работает. Тут, конечно, даже думать нечего – хватаем и тащим. Если дистрибутив – установить его традиционным образом, со времён DOS'а тут мало что изменилось.

Если образ – скопировать его к себе на жёсткий диск. В любом случае для установки рекомендуется каталог с именем типа "c:\bpascal". Всяческие Program Files не рекомендуются категорически – установить, конечно, можно, и, даже, работать будет, но в дальнейшем возможно возникновение мелких и не очень проблем. Дальше я буду предполагать, что Турбо Паскаль установлен именно в каталог c:\bpascal\.

Причина – Турбо Паскаль разработан для DOS, а не для Windows, соответственно родные имена каталогов и файлов для него состоят не более чем из восьми символов без пробелов с возможным расширением до трех символов.

Если всё хорошо, то в каталоге `c:\bpascal` найдутся подкаталоги с именами

```
c:\bpascal\bin
c:\bpascal\units
c:\bpascal\bgi
```

Это как минимум, вполне возможно, что подкаталогов будет больше. Наш самый главный файл `turbo.exe` находится в каталоге `c:\bpascal\bin`. Если его запустить – получим готовую к работе среду Турбо Паскаля. А как запустить? Для повседневного употребления – создать иконку. Или, что лучше, создать пункт пользовательского меню в, например, FAR'е или добавить вызов в панель инструментов в Total Commander.

На скорую руку обычно просто вызывают тот самый `turbo.exe` из его родного каталога `c:\bpascal\bin\`. Но это не совсем хороший вариант – файлы с исходными текстами программ, если не предпринимать дополнительных усилий, валяются в тот же каталог, откуда Паскаль вызвали, туда же отправляются полученные в результате исполняемые файлы и, даже, о ужас! – объектные файлы. Хуже всего то, что всё валится в один каталог, где образуется гигантская помойка.

Возможно, не всем всё понятно, что я говорю. Это, в общем-то, нормально - не написав ни одной программы, нам приходится обсуждать, какие этапы она проходит на своем жизненном пути. Попробуем, однако, немного разобраться.

Мы запустили интегрированную среду разработки Turbo Pascal – тот самый исполняемый файл `turbo.exe`. Набрали исходный текст нашей программы и сохранили его. Если специально не озаботиться местом его сохранения, то сохранится он в каталоге, откуда этот `turbo.exe` запущен – то есть `c:\bpascal\bin`. Например, исходный текст сохранили в файле `song.pas` – ну не программа у нас, а просто песня! Затем программу мы транслируем и получаем исполняемый файл `song.exe` – в том же каталоге. Попутно, в зависимости от настроек, могут образоваться и другие файлы – `song.bak`, `song.map` и ещё кто-нибудь. Если наша программа состоит из нескольких модулей – то есть содержит несколько файлов исходных текстов – то образуются файлы с именами типа `song.tpu`. У всех образовавшихся файлов одно и то же имя, но разные расширения. Ещё

обязательно появится файл с именем turbo.tp – обратите внимание, что его имя не связано с именем нашей программы.

Всё это вместе называется проект. На самом деле оно так не называется, вернее, именно так оно и называется, но только в Borland Delphi и других современных средах программирования. Поскольку термин удачный и востребованный, будем применять его и по отношению к Турбо Паскалю. Проект – совокупность всех файлов, относящихся к одной программе. Из одного проекта получаем один исполняемый файл. Как всегда, всё не совсем так, но в первом приближении сойдёт

А как всё сделать правильно? По-хорошему, все файлы, относящиеся к одному проекту (программе) должны находиться в одном отдельном каталоге. Они и только они. Один проект – один каталог.

То есть – собрались писать новую программу – создаёте новый каталог для неё. Заходите в этот каталог и уже оттуда вызываете Турбо Паскаль, чтобы в дальнейшем все ваши файлы сохранялись в этом самом каталоге. Проиллюстрируем, как это сделать с помощью FAR'а. В нём надо создать пользовательское меню (user menu). F9, <Commands>Edit user menu>, Main, Insert command – а кто говорил, что легко будет? Хотя чего трудного, собственно? Hot key и Label по вкусу, в Command пишем наше уже знакомое c:\bpascal\bin\turbo.exe.

Теперь, создав новый каталог для проекта, или зайдя в каталог с проектом уже созданным, нажимаем F2 и выбираем из меню наш Паскаль. Все созданные нами и возникшие самопроизвольно файлы будут сыпаться именно в этот каталог.

Теперь об очень-очень печальном. Турбо Паскаль работать на современных быстрых процессорах не может. Ну, вообще. Ну, почти. То есть, он делает вид, что работает. Пока мы не впишем в секцию uses модуль Crt – а мы его очень скоро туда впишем. После этого при запуске *любой* программы получаем совершенно неуместное сообщение о делении на ноль.

Как бороться? Если у Вас уже упомянутая версия 7.01, то повезло. Borland, ясное дело, такой версии не выпускал, это дело рук отечественных Кулибиных. Если не повезло, то будете сам себе Кулибин.

Идёте в Яндекс и набираете “*turbo.tpl исправленный*”. Скачиваете файл и заменяете. И будет вам счастье!

## Как настроить Турбо Паскаль, чтобы было приятно и удобно

В принципе, у нас уже всё хорошо – то есть Турбо Паскаль запускается и как-то даже трепыхается. Но, естественно, хочется ещё лучше. А, чтобы было ещё лучше, надо слегка подкрутить настройки Паскаля – те, что идут по умолчанию, как всегда, в целом неплохо, но не совсем соответствуют. В принципе, конечно, можно ничего не трогать. Но лучше всё-таки потрогать

Как до настроек добраться? Нажимаем **F10** – активизируется главное меню. Шлёпаем стрелочкой направо, пока не доберёмся до пункта главного меню под названием <Options> и нажимаем <Enter>. Разворачивается меню, точнее - подменю). Мышкой всё это сделать тоже можно, но неспортивно. Мышка – оружие дельфиста, паскалисты шлёпают по клавишам.

Теперь наша задача – чтобы внесённые нами изменения в настройки действовали всегда и везде. Отправляемся в пункт меню <Save as...> Можно побороться с не очень хитрой системой навигации по каталогам, но мне так кажется, что оно нам в дальнейшем не понадобится, поскольку задача наша одноразовая. Так что трудолюбиво в верхней строке (Options file name) набираем `c:\bpascal\bin\turbo.tp` и нажимаем Enter. На всякий случай прогуляйтесь в этот самый каталог – там должен присутствовать файл `turbo.tp` с текущей датой и свеженьким временем. Снова войдите в меню <Options>. В пункте меню <Save> (не <Save as...>!) появилось имя и путь сохранённого файла, возможно в сокращённом виде – что-то типа `c:turbo.tp`. Это нормально.

Теперь мы будем менять настройки, как нам больше нравится, а для сохранения нажимать на <Options\Save>. И в путь по закоулкам меню. Движемся сверху вниз.

<Compiler options>. Ставим птички.

Runtime Errors – все.

Debugging – все.

Numeric Processing – все.

Syntax Options – помечаем Strict var Strings и Extended Syntax

Больше птичек нам не надо. Скорее всего, в этом разделе и так всё было хорошо. Извините, что не объясняю, какая птичка зачем – все птички переустанавливать не придётся. Поставить и забыть.

<Memory sizes> - игнорируем. Всё пока хорошо.

<Linker> - Всё равно.

<Debugger>. Пропускаем, всё должно быть уже хорошо. На всякий случай – Integrated, Smart.

<Directories>. Тут важнее. В пункте Unit directories вписываем c:\bpascal\units. Вы, конечно, понимаете, что все эти каталоги, возможно, придётся скорректировать с учётом того, как они называются конкретно у вас.

<Tools>. Безусловно, пропускаем.

<Environment>. Тут работы будет побольше.

<Environment\Preferences>. Screen sizes у устанавливаем в 25 Lines. Desktop file – Current directory. В разделе Auto Save помечаем всё.

<Environment\Editor>. Расставьте галочки по вкусу, большого вреда не будет. Только обязательно отметьте Syntax highlight – чтобы было красиво. И подумайте, нужны ли Вам мусорные файлы. Если нет, выключите Create Backup Files.

<Environment\Mouse>. Здесь вроде бы изначально всё хорошо.

<Environment\Startup>. Аналогично.

<Environment\Colors>. Обратите внимание на раздел Syntax. Покрасьте по настроению. Лично я всегда крашу числа в голубой, а строки в зелёный, но своё мнение никому не навязываю.

Очень важно. При запуске программы из среды Турбо Паскаля у Вас всё будет хорошо. Ну, или всё будет нехорошо, но хоть как-то будет. Когда Вы вернётесь во внешний мир, то обнаружите, что от Вашей бурной деятельности не осталось никаких следов. Нет исполняемого файла! А почему?

F10\Compile\Destination. Видим текст Destination Memory. То есть программа наша компилируется и компоуется как положено, но результат сохраняется только в оперативной памяти. А после выхода из Турбо Паскаля – всё пропало. Выберите этот пункт – и поменяйте назначение. Когда вы посмотрите на него следующий раз, там будет стоять Destination Disk. То есть, программа изготавливается на диске, и у вас остаётся весомый, грубый, зримый exe-файл.

Вот, собственно и всё. Не забудьте сохранить настройки - <Options\Save>.

### **И ещё кое-что**

Если нажать Alt/F1, то мы попадём в оглавление справочной системы. Более того, если встать курсором на какое-нибудь умное слово, например integer и нажать Ctrl/F1 то мы получим справку по существу вопроса – в данном случае – справку по типу integer, точнее, справку по всем целым типам.

И всё бы хорошо, но справка на английском. Лично моё мнение, что ничего плохого в этом нет, а есть только полезное. Программист без знания технического английского языка в режиме чтения, подлежит немедленной утилизации в целях прекращения бесполезных страданий и освобождения занимаемых квадратных метров. Но некоторые стонут и плачут. Становится жалко. Но есть, есть справка и по-русски! Необходима для этого подмена файла turbo.tph. Где искать? В Интернете, вестимо. Но я советую оставить по-английски. И ещё кое-что. Не забывайте про волшебное сочетание клавиш Alt/Enter. Оно сделает Ваше маленькое ДОСовское окошко большим и симпатичным.

### **Имейте свой стиль**

Своего стиля у Вас пока что нет. Так что за неимением своего, будете иметь мой. Я понимаю, что это диктатура, задушенная свобода слова и растоптанный нераспустившийся хрупкий цветок юного программистского гения. Свободолюбие и своеобразие из юного программиста мы будем выбивать линейкой по пальцам.

*Тебя посодют, а ты не воруй* © к/ф Берегись автомобиля.

Можете дальнейшее пропустить – если знаете, как лучше.

1. Большие и маленькие буквы (прописные и строчные, тьфу).

- a. Все слова Паскаля пишутся с маленькой буквы. Имеются в виду слова как зарезервированные - **var, begin, end, string, div**, так и не зарезервированные - `integer, single`. Если сказано, что слово пишется с маленькой буквы, подразумевается, кроме особо оговоренных случаев, что и все остальные буквы в слове тоже маленькие
- b. Все идентификаторы пишутся с маленькой буквы, кроме особо оговоренных случаев
- c. Если имя состоит из нескольких слов, второе и последующие слова допустимо начинать с большой буквы

`numoflines`  
`numOfLines` – так даже лучше

Также в этом случае допустимо разделять слова символом подчёркивания и начинать идентификатор с него

`num_of_lines`  
`num_of_Lines`  
`_numOfLines`

- d. Имена констант могут начинаться как с большой, так и с маленькой буквы
- e. Имя программы может писаться с большой или маленькой буквы
- f. Имена процедур и функций - при объявлении и при вызове - пишутся с большой буквы
- g. Имена модулей в `uses` только с большой буквы
- h. Имена массивов могут писаться с большой или маленькой буквы
- i. В имени типа большими обязательно являются первая буква T и следующая за ней буква

`TBigArray = array[1..60000] of byte;`

## 2. Объявления

- a. Большому кораблю – большое плавание, маленькому – соответственно. Имена переменных индивидуальных (правильно - скалярных) пишутся исключительно с

маленькой буквы, имена агрегатов – массивы, классы – допустимо, но не обязательно, с большой

- b. i,j,k используются только для цикла **for**, пишутся с маленькой
- c. Не мешайте в именах русский с английским. EtoIsVulgar.
- d. Имя типа записи имеет первые две буквы большие - на правах типа. В конце имеет суффикс **Rec**

```
TBigRec = record
    bigArray          : TBigArray;
end;
```

- e. Имя типа должно начинаться с буквы **T**
- f. Двоеточия в секции **var** располагаются строго одно под другим
- g. Каждая переменная объявляется на отдельной строке. Переменные аналогичного назначения – beginX, beginY - допускается объявлять совместно на одной строке

### 3. Отступы и позиции

- a. Пишутся с левой границы: **program**, первый - в программе, процедуре, функции - **begin, end., procedure, function, unit, interface, implementation.**
- b. Пишутся с 4-й позиции: **uses, const, type, var.**
- c. Невложенные операторы пишутся с 7-й позиции.
- d. Каждый следующий уровень вложенности выделяется тремя отступами – плюс три позиции.
- e. Имена объявляемых переменных пишутся с 7-й позиции. Поля записи объявляются с тремя отступами относительно слова **record**

```
TBadGirl = record
    name           : string;
    address        : string;
    reallyBad      : boolean;
end;
```

- f. При объявлении записи - как типа или непосредственно - **record** записывается где придётся, соответствующий **end** строго под ним.
- g. При объявлении переменных двоеточие ставится в 32-й позиции.

#### 4. Оформление процедур

- a. Начало и конец процедуры выделяются строкой комментариев { 68 символов “-“ }.
- b. Начало и конец вложенной процедуры выделяются строкой комментариев { 58 символов “. ” }.
- c. Между двумя процедурами ставится только одна строка комментариев.
- d. Каждый параметр объявляется на отдельной строке. Сходные по назначению параметры допускается объявлять на одной строке.
- e. Параметры с **var** объявляются по следующей схеме:  
Пробел – **var** – пробел – имя – пробел - двоеточие – пробел – тип.

```
procedure MS( var a : TArray);
```

- f. Параметры без **var** объявляются  
Пять пробелов – имя – пробел – двоеточие – пробел – тип.

```
procedure MS(      a : TArray);
```

#### 5. Использование пробелов

- a. Оператор присваивания пишется без пробелов с обеих сторон  
`x:=y;`
- b. В выражении + и – выделяются пробелами. \* и / не выделяются  
`x:=a*b + a/b;`
- c. При объявлении констант знак равенства выделяется пробелами с обеих сторон  
`maxNumber = 50;`
- d. При объявлении тип переменной отделяется от двоеточия одним пробелом.
- e. При вызове процедуры фактические параметры отделяются пробелом.
- f. В секции **uses** модули перечисляются без пробела перед запятой, с пробелом после запятой

```
uses  
  Crt, Dos, Graph;
```

- g. Двоеточие перед типом функции выделяется пробелами с обеих сторон

```
function Something : boolean;
```

- h. Знак отношения в условном операторе выделяется пробелами

```
if x > y then SomethingReallyBad;
```

## 6. Операторы

- a. Каждый оператор пишется на отдельной строке.  
b. В условном операторе **begin** пишется в той же строке

```
if x > y then begin
```

- c. Условный оператор может быть написан на одной строке, или **then** начинается на следующей строке с тремя отступами

```
if x > y then z:=100;  
if x > y  
  then z:=100;
```

- d. Вложенная конструкция **if – then – else** записывается следующим образом

```
if x < 10 then begin  
end else  
if x < 100 then begin  
end else  
if x < 1000 then begin  
end  
else begin  
end;
```

- e. В операторе цикла **begin** пишется в той же строке

```
for i:=1 to N do begin
```

## 7. Прочее

- a. Порядок следования секций: **uses, const, type, var.**

Если в процессе чтения этой книги вам в каком-то месте встретился текст программы, не соответствующий этим Высоким Требованиям, будьте уверены – это опечатка. И вообще, я хочу, чтобы все ходило строем. Ну, может, не все. Но программисты – точно.

## Все полезные клавиши на одной странице

F2 – сохранить

F3 – загрузить

F9 – скомпилировать

Ctrl/F9 – скомпилировать и выполнить

Alt/F5 – посмотреть, что там, за экраном

Ctrl/Break – прекратить выполнение программы

F10 – главное меню

Alt/X – выйти из Турбо Паскаля

F5 – увеличить окно редактирования

F6 – перейти к следующему окну

Alt/F3 – закрыть текущее окно

F7 – выполнить

F8 – выполнить, но не входить в процедуру

F4 – выполнить до этого места

Ctrl/F2 – прекратить работу в режиме отладки

Ctrl/F7 – посмотреть переменную

Ctrl/F8 – включить/выключить точку останова

Ctrl/F1 - справка по слову, на котором стоит курсор

Ctrl/F1, находясь в справке - оглавление справочной системы

Alt/F1 – вернуться к предыдущему справочному экрану

Правый Shift – переключить русский/английский язык

Alt/Enter – большой-большой экран

**Все типы данных на одной странице (ну, на двух...)  
Даже те, которые от вас скрывали**

Целые:

shortint	1 байт	знаковое	-128..127
byte	1 байт	беззнаковое	0..255
integer	2 байта	знаковое	-32768..32767
word	2 байта	беззнаковое	0..65535
longint	4 байта	знаковое	-2147483648..2147483647

Плавающие типы:

single	4 байта	7-8	1.5e-45..3.4e38
real	6 байт	11-12	2.9e-39..1.7e38
double	8 байт	15-16	5.0e-324..1.7e308
extended	10 байт	19-20	3.4e-4932..1.1e4932
comp	8 байт	19-20	-9.2e18..9.2e18

Третья колонка – точность в знаках, четвёртая – диапазон.

boolean	8 бит
WordBool	16 бит
LongBool	24 бита
ByteBool	8 бит

Последние три интереса не представляют и художественной ценности не имеют.

char	1 байт
pointer	4 байта
string	256 байт
string[константа]	константа+1 байт
pchar	немедленно забудьте

Структурные типы:

array  
file  
set

record  
object

Все структурные типы, кроме объектного, вам знакомы. Про объектный тип здесь не будем, это отдельная длинная песня. Рекомендую сам себя:

*Комлев Н.Ю. Объектно Ориентированное Программирование. Хорошая книга для хороших людей, М. Солон-Пресс, 2014*

Ещё есть процедурный тип. Это песня не очень длинная, но всё равно, здесь не будем.

## Чем заняться на досуге

1. Запрограммировать Civilization I
2. Запрограммировать крестики нолики на бесконечной доске
3. Доделайте пятнашки. Ведь ерунда осталась
4. Почитайте о фракталах. Дёшево и сердито. В смысле, запрограммировать всего ничего, а результат – о-го-го!
5. Хочу простенький музыкальный редактор. Нотный стан, под ним, или по нему, ездит курсор, клавишами задаём какую в этом месте поставить ноту - высота и длительность. Возможность задания общего темпа. Сохранение/восстановление мелодий – обязательно.
6. Запишите нашей программой для музыки *Девятую симфонию* Бетховена. Или *Пятую*? Какая разница! А лучше *Маленькую Ночную Серенаду* Моцарта. Та, что из *Женитьбы Фигаро*:

*Некий муж за святость брака,  
Чтоб с женой не вышла зла,  
Им огромная собака  
Приобретена была...*

Я что, впал в маразм? Отнюдь. Просто я точно знаю, что обычный унылый спикер компьютера является, по сути своей, трёхголосым. Если вы вообще понимаете о чём я говорю. Так что, держайте!

## Модуль для работы с клавиатурой

```
unit Scan;
{-----}
interface
{----- Scan Codes -----}
  const
    ArrowLeft = 75;   ArrowRight = 77;
    ArrowDown = 80;   ArrowUp    = 72;
    PgUp       = 73;   PgDn      = 81;
    Enter      = 28;
    Esc        = 1;
    Bksp       = 14;
    SpaceBar   = 57;
    Del        = 83;   Ins         = 82;
    Ctrl       = 29;   Alt         = 56;
    Tab        = 15;
    Home       = $47;  kEnd        = $4F;
    LShift     = $2A;  RShift     = $36;
    GrayMinus  = $4A;  GrayPlus   = $4E;

    F1 = 59;  F2 = 60;  F3 = 61;  F4 = 62;  F5 = 63;
    F6 = 64;  F7 = 65;  F8 = 66;  F9 = 67;  F10 = 68;
    F11 = $85; F12 = $86;

    F1Shift = $54;  F2Shift = $55;  F3Shift = $56;  F4Shift =
$57;
    F5Shift = $58;  F6Shift = $59;  F7Shift = $5A;  F8Shift =
$5B;
    F9Shift = $5C;  F10Shift = $5D;  F11Shift = $87;  F12Shift =
$88;

    ch0 = $0B;  ch1 = $02;  ch2 = $03;  ch3 = $04;  ch4 = $05;
    ch5 = $06;  ch6 = $07;  ch7 = $08;  ch8 = $09;  ch9 = $0A;

    chA = $1E;  chS = $1F;  chD = $20;  chF = $21;  chQ = $10;
    chW = $11;  chE = $12;  chRr = $13;  chT = $14;  chY = $15;
    chU = $16;  chI = $17;  chO = $18;  chP = $19;  chZ = $2C;
    chX = $2D;  chC = $2E;  chV = $2F;  chG = $22;  chH = $23;
    chJ = $24;  chK = $25;  chL = $26;  chB = $30;  chN = $31;
    chM = $32;

    CtrlF1 = $5E;  CtrlF2 = $5F;  CtrlF3 = $60;  CtrlF4 = $61;
    CtrlF5 = $62;  CtrlF6 = $63;  CtrlF7 = $64;  CtrlF8 = $65;
    CtrlF9 = $66;  CtrlF10 = $67;  CtrlF11 = $89;  CtrlF12 = $8A;

    CtrlTab = $94;
    CtrlArrowLeft = 115;  CtrlArrowRight = 116;
    CtrlArrowUp = 141;  CtrlArrowDown = 145;
    CtrlHome = $77;  CtrlEnd = $75;
    CtrlPgDn = $76;  CtrlPgUp = $84;

    AltF1 = $68;  AltF2 = $69;  AltF3 = $6A;  AltF4 = $6B;
```

```

    AltF5 = $6C; AltF6 = $6D; AltF7 = $6E; AltF8 = $6F;
    AltF9 = $70; AltF10 = $71; AltF11 = $8B; AltF12 = $8C;

    Alt0 = $81; Alt1 = $78; Alt2 = $79; Alt3 = $7A; Alt4 =
$7B;
    Alt5 = $7C; Alt6 = $7D; Alt7 = $7E; Alt8 = $7F; Alt9 =
$80;

    AltLeft = $9B; AltRight = $9D;
    AltDown = $A0; AltUp = $98;
    AltPgUp = $99; AltPgDn = $A1;
    AltEnter = $1C;
    AltDel = $A3; AltIns = $A2;
    AltTab = $A5;
    AltHome = $97; AltEnd = $9F;
    AltX = $2D;
}
-----}
function OurKeyPressed : boolean;
function OurReadKey : byte;
}
-----}
implementation
    uses
        Dos;
}
-----}
function OurKeyPressed : boolean;
    var
        R : Registers;
begin
    R.ah:=1;
    Intr( $16, R);
    if (R.flags and fZero) <> 0
        then OurKeyPressed:=false
        else OurKeyPressed:=true;
end;
}
-----}
function OurReadKey : byte;
    var
        R : Registers;
        ch : char;
        sc : byte;
begin
    R.ah:=0;
    Intr( $16, R);
    ch:=Chr(R.al); sc:=R.ah;
    OurReadKey:=sc;
end;
}
-----}
end.

```

## Модуль для работы с нотами

Предусмотрены процедуры для нот первой, второй и третьей октав, а также процедура для паузы. Глобальная переменная `one` означает длительность целой ноты. Глобальная переменная `leg` отвечает за исполнение нот легато. Если написать

```
leg:=2; a1(8); a1(8); a1(8);
```

то первые две ноты будут исполнены связно, а третья – нет. Приношу извинения за запись процедур в одной строчке. Так, исключительно в данном случае, короче и нагляднее. Обратите внимание что перед последней строкой модуля **end.** появился **begin.** То, что между ними, называется секцией инициализации. Она выполняется однократно при запуске программы. В нашем случае секция инициализации понадобилась для инициализации переменной `one`. Не могли мы оставить её на совести пользователя.

*Потому что нельзя © песня*

```
unit Notes;
{-----}
interface
    var
        one           : integer;
        leg           : integer;

procedure c1 (      t : single);
procedure c1d(     t : single);
procedure d1 (      t : single);
procedure d1d(     t : single);
procedure e1 (      t : single);
procedure f1 (      t : single);
procedure f1d(     t : single);
procedure g1 (      t : single);
procedure g1d(     t : single);
procedure a1 (      t : single);
procedure b1 (      t : single);
procedure h1 (      t : single);

procedure c2 (      t : single);
procedure c2d(     t : single);
procedure d2 (      t : single);
procedure d2d(     t : single);
procedure e2 (      t : single);
procedure f2 (      t : single);
```

```

procedure f2d(      t : single);
procedure g2 (      t : single);
procedure g2d(      t : single);
procedure a2 (      t : single);
procedure b2 (      t : single);
procedure h2 (      t : single);

procedure c3 (      t : single);
procedure c3d(      t : single);
procedure d3 (      t : single);
procedure d3d(      t : single);
procedure e3 (      t : single);
procedure f3 (      t : single);
procedure f3d(      t : single);
procedure g3 (      t : single);
procedure g3d(      t : single);
procedure a3 (      t : single);
procedure b3 (      t : single);
procedure h3 (      t : single);

procedure pa(      t : single);
{-----}
implementation
  uses
    OpCrt;
{-----}
procedure OurDelay(  howmany : single);
  var
    i,j                : longint;
begin
  for i:=1 to Round(howmany) do
    for j:=1 to 100000 do;
end;
{-----}
procedure MS(      hz : integer;
               t : single);
  var
    ti,ti1,ti2        : single;
begin
  ti:=one/t;
  ti2:=ti/8;
  ti1:=ti - ti2;

  if leg>1 then begin
    leg:=leg-1;
    ti1:=ti;
    ti2:=0;
  end;

  Sound(hz);
  OurDelay(ti1);
  NoSound;
  OurDelay(ti2);

```

```

end;
{-----}
procedure c1 (      t : single); begin MS(262 div 2,t); end;
procedure c1d(     t : single); begin MS(278 div 2,t); end;
procedure d1 (      t : single); begin MS(294 div 2,t); end;
procedure d1d(     t : single); begin MS(311 div 2,t); end;
procedure e1 (      t : single); begin MS(330 div 2,t); end;
procedure f1 (      t : single); begin MS(349 div 2,t); end;
procedure f1d(     t : single); begin MS(368 div 2,t); end;
procedure g1 (      t : single); begin MS(392 div 2,t); end;
procedure g1d(     t : single); begin MS(415 div 2,t); end;
procedure a1 (      t : single); begin MS(440 div 2,t); end;
procedure b1 (      t : single); begin MS(465 div 2,t); end;
procedure h1 (      t : single); begin MS(494 div 2,t); end;

procedure c2 (      t : single); begin MS(262,t); end;
procedure c2d(     t : single); begin MS(278,t); end;
procedure d2 (      t : single); begin MS(294,t); end;
procedure d2d(     t : single); begin MS(311,t); end;
procedure e2 (      t : single); begin MS(330,t); end;
procedure f2 (      t : single); begin MS(349,t); end;
procedure f2d(     t : single); begin MS(368,t); end;
procedure g2 (      t : single); begin MS(392,t); end;
procedure g2d(     t : single); begin MS(415,t); end;
procedure a2 (      t : single); begin MS(440,t); end;
procedure b2 (      t : single); begin MS(465,t); end;
procedure h2 (      t : single); begin MS(494,t); end;

procedure c3 (      t : single); begin MS(262*2,t); end;
procedure c3d(     t : single); begin MS(278*2,t); end;
procedure d3 (      t : single); begin MS(294*2,t); end;
procedure d3d(     t : single); begin MS(311*2,t); end;
procedure e3 (      t : single); begin MS(330*2,t); end;
procedure f3 (      t : single); begin MS(349*2,t); end;
procedure f3d(     t : single); begin MS(368*2,t); end;
procedure g3 (      t : single); begin MS(392*2,t); end;
procedure g3d(     t : single); begin MS(415*2,t); end;
procedure a3 (      t : single); begin MS(440*2,t); end;
procedure b3 (      t : single); begin MS(465*2,t); end;
procedure h3 (      t : single); begin MS(494*2,t); end;
{-----}
procedure pa(      t : single);
begin
    OurDelay( one/t);
end;
{-----}
begin
    one:=1000;
end.

```

## Полный и аккуратный текст программы про Ханойские Башни

```
program Hanoi; { 14.02.2017 }
              { 21.02.2017 }

uses
  OpCrt, Graph, Scan2;
const
  { основание нижней левой части первого слева стержня - ну вы
  поняли, да? }
  x0 = 100;
  y0 = 300;
  { расстояние между стержнями по горизонтали }
  xIncr = 200;
  { не знаю, это именно это такое, но какое-то расстояние по
  вертикали }
  yIncr = 5;
  { ширина стержня }
  wRod = 10;
  { ширина верхнего (самого маленького) кольца }
  small = 30;
  { увеличение следующего кольца }
  incr = 10;
  { высота стержня }
  h = 200;
  { толщина кольца }
  thi = 20;
  backColor = LightGray;
  { }
  curSize = 50;
const
  maxK = 64;
  howMany = 5;
type
  THanoi = array[1..3,1..maxK] of integer;
var
  driver,mode           : integer;
  Ha                   : THanoi;
  HaSk                 : array[1..3] of integer;
  whereCurs            : integer;
  gamover              : boolean;
  outerDisk            : integer;
  skoka                : integer;
  sc                   : byte;
{-----}
procedure Init;
begin
  Ha[1,1]:=HowMany;   Ha[1,2]:=HowMany-1;
  Ha[1,3]:=HowMany-2; Ha[1,4]:=HowMany-3;
  Ha[1,5]:=HowMany-4;
  HaSk[1]:=5;
end;
{-----}
procedure Figovina(   num : integer);
```

```

begin
  SetColor(Green);
  SetFillStyle( SolidFill, Green);

  Bar3D( x0+(num-1)*xIncr,    y0,
         x0+(num-1)*xIncr+20, y0-h,
         10, TopOn);
end;
{-----}
procedure DrawDiskInOuterSpace(    numDisk : integer);
  var
    width          : integer;
begin
  width:=small + (numDisk-1)*incr;

  SetColor(Blue);
  SetLineStyle( 0, 0, 2);
  SetFillStyle( SolidFill, Yellow);
  FillEllipse( 550,40, width,thi);
end;
{-----}
procedure HideDiskInOuterSpace;
begin
  SetColor(backColor);
  SetLineStyle( 0, 0, 2);
  SetFillStyle( SolidFill, backColor);
  FillEllipse( 550,40, 100,thi);
end;
{-----}
procedure DrawDisk(    num      : integer;
                      posOnRod : integer; { считаем снизу }
                      numDisk  : integer);
  var
    width          : integer; { ширина рисуемого кольца }
    x1,y1          : integer;
    i              : integer;
begin
  width:=small + (numDisk-1)*incr;

  x1:=x0 + (num-1)*xIncr;

  y1:=y0;
  for i:=1 to posOnRod do begin
    y1:=y1 - thi - yIncr;
  end;

  SetColor(Blue);
  SetLineStyle( 0, 0, 2);
  SetFillStyle( SolidFill, Yellow);
  FillEllipse( x1+10,y1, width,thi);
end;
{-----}
procedure DrawRod(    num  : integer);

```

```

var
    i                : integer;
begin
    Figovina(num);
    for i:=1 to HaSk[num] do begin
        DrawDisk( num, i, Ha[num,i]);
    end;
end;
{-----}
procedure HideRod(    num : integer);
begin
    SetColor( backColor);
    SetFillStyle( SolidFill, backColor);
    Bar( x0+(num-1)*xIncr    -80,    y0,
        x0+(num-1)*xIncr+wRod+80,    y0-h-30);
end;
{-----}
procedure DrawCursor(    num : integer);
begin
    SetColor(Blue);
    SetFillStyle( SolidFill, Blue);
    Bar( x0+(num-1)*(xIncr+3) - 15, y0+10,
        x0+(num-1)*(xIncr+3) + curSize - 15,y0+20);
end;
{-----}
procedure HideCursor(    num : integer);
begin
    SetColor(backColor);
    SetFillStyle( SolidFill, backColor);
    Bar( x0+(num-1)*(xIncr+3) - 15, y0+10,
        x0+(num-1)*(xIncr+3) + curSize - 15,y0+20);
end;
{-----}
begin
    driver:=EGA;
    mode:=EGAHi;
    InitGraph( driver, mode, 'c:\bpascal\bgi');

    Init;
    whereCurs:=1;
    gamover:=false;
    outerDisk:=0;

    SetFillStyle( SolidFill, backColor);
    Bar( 0,0, 639, 349);

    DrawRod(1);
    DrawRod(2);
    DrawRod(3);

    DrawCursor( whereCurs);

repeat

```

```

if OurKeyPressed then begin
  sc:=OurReadKey;
  if sc = ArrowRight then begin
    if whereCurs < 3 then begin
      HideCursor(whereCurs);
      whereCurs:=whereCurs + 1;
      DrawCursor(whereCurs);
    end;
  end else
  if sc = ArrowLeft then begin
    if whereCurs > 1 then begin
      HideCursor(whereCurs);
      whereCurs:=whereCurs - 1;
      DrawCursor(whereCurs);
    end;
  end else
  if sc = SpaceBar then begin
    skoka:=HaSk[whereCurs];
    if (outerDisk > 0) and
      (( skoka = 0) or (outerDisk < Ha[whereCurs,skoka]))
    then begin { plus }
      skoka:=skoka + 1;
      HaSk[whereCurs]:=skoka;
      Ha[whereCurs,skoka]:=outerDisk;
      HideDiskInOuterSpace;
      outerDisk:=0;
      HideRod(whereCurs);
      DrawRod(whereCurs);
    end
    else begin { minus }
      if (skoka >= 1) and (outerDisk = 0)
      then begin
        outerDisk:=Ha[whereCurs,skoka];
        Ha[whereCurs,skoka]:=0;
        skoka:=skoka - 1;
        HaSk[whereCurs]:=skoka;
        HideRod(whereCurs);
        DrawRod(whereCurs);
        DrawDiskInOuterSpace(outerDisk);
      end;
    end;
    HaSk[whereCurs]:=skoka;
  end else
  if sc = Esc then begin
    Break;
  end;
end;
until gamover;

CloseGraph;

end.

```

На странице осталось ещё свободное место, так что добавим немного умных слов.

Нужна ли и уместна ли здесь рекурсия? А почему?

Масштабируема ли наша/ваша программа? То сеть – можно ли увеличить число стержней до 32, а число колец до  $2^{32}$ ? Умрёт ли программа, и как именно она умрёт?

Сделайте всё красиво, в конце концов.

Всё. Книга кончилась.

---

**Рик Гаско**

## **Простой учебник программирования**

*Серия «Программирование»*

Ответственный за выпуск: **В. Митин**

Под редакцией: **Н. Комлева**

Обложка: **СОЛОН-Пресс**

По вопросам приобретения обращаться:

*ООО «СОЛОН-Пресс»*

*123001, г. Москва, а/я 82*

*Телефоны: (495) 617-39-64, (495) 617-39-65*

*E-mail: [kniga@solon-press.ru](mailto:kniga@solon-press.ru), [www.solon-press.ru](http://www.solon-press.ru)*

**ООО «СОЛОН-Пресс»**

115487, г. Москва,

пр-кт Андропова, дом 38, помещение № 8, комната № 2.

Формат 60×88/16. Объем 20 п. л. Тираж 100 экз.