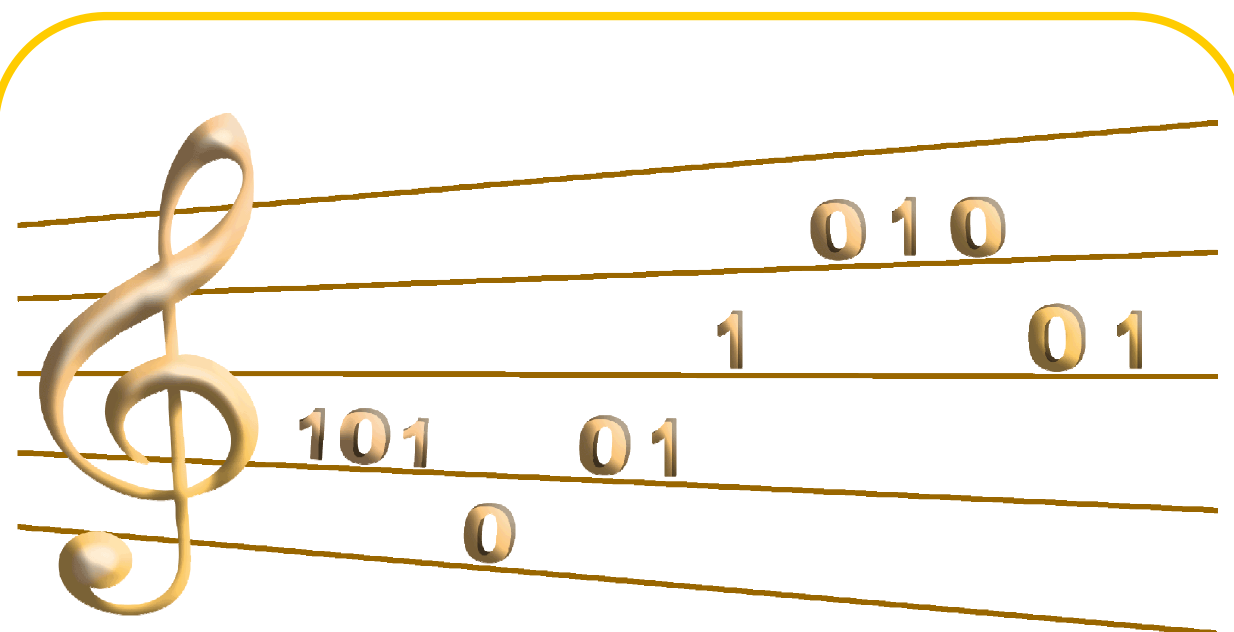


Песни о Паскале

Редакция 12.8

от 2014-07-21

© Деревенец Олег Виленович, 2010-2014,
все права защищены

http://oleg-derevenets.narod.ru

Аннотация

Изложены основы программирования на языке Паскаль. По ходу обучения решаются десятки задач (использован проектный подход). От читателя не требуется начальных познаний в программировании, но круг затронутых тем ориентирует его в профессиональную область. Книга адресована школьникам средних и старших классов, желающим испытать себя в «олимпийских схватках». Будет полезна студентам-первокурсникам и преподавателям информатики.

Условия использования

Произведение «Песни о Паскале» созданное автором по имени Деревенец Олег Виленович, публикуется на условиях лицензии Creative Commons Attribution-NonCommercial-NoDerivs (*Атрибуция — Некоммерческое использование — Без производных произведений*) 3.0 Непортированная.

Разрешения, выходящие за рамки данной лицензии, могут быть доступны на странице http://oleg-derevenets.narod.ru.

Оглавление

Только для взрослых ... 15
Детям до 16-ти ... 19
Глава 1 Путь далек у нас с тобою… ... 20
Глава 2 Вместо теории ... 23
Глава 3 Консольный интерфейс ... 28
Глава 4 Оружие – к бою! ... 32
Глава 5 Программа номер один ... 39
Глава 6 Подготовка к следующему штурму ... 46
Глава 7 Развиваем успех ... 50
Глава 8 Постоянные и переменные ... 56
Глава 9 Переменные: продолжение знакомства ... 63
Глава 10 Условный оператор ... 69
Глава 11 Операторный блок ... 75
Глава 12 Цикл с проверкой в конце ... 79
Глава 13 Правда и кривда ... 86
Глава 14 Дважды два – четыре ... 96
Глава 15 Айда в Монте-Карло! ... 101
Глава 16 Делу время, а потехе час ... 106
Глава 17 И вновь за парту ... 111
Глава 18 Аз, Буки ... 116
Глава 19 Процедуры и функции: разделяй и властвуй ... 121
Глава 20 Процедуры: первый опыт ... 127
Глава 21 Отладка ... 136
Глава 22 О передаче параметров ... 144
Глава 23 Функции ... 148
Глава 24 Криптография ... 154
Глава 25 Текстовые файлы ... 163
Глава 26 Я не читатель, — я писатель! ... 171
Глава 27 Дайте кораблю минутный отдых! ... 179
Глава 28 Редактор и справочная система ... 186
Глава 29 Читайте по-новому ... 191
Глава 30 Журнальная история ... 197
Глава 31 Финал журнальной истории ... 210
Глава 32 Порядковые типы данных ... 217
Глава 33 Вещественные числа ... 230
Глава 34 Структура программы ... 241
Глава 35 Множества ... 248
Глава 36 Множества в Паскале ... 256
Глава 37 Ввод и вывод множеств ... 262
Глава 38 Множества «в бою» ... 268
Глава 39 Командная игра (массивы) ... 278
Глава 40 Пристрелка на знакомых мишенях ... 288
Глава 41 По порядку, становись! ... 294
Глава 42 Кто ищет, тот всегда найдет ... 302
Глава 43 Сортировка по-взрослому ... 316
Глава 44 Строки ... 331
Глава 45 Очереди и стеки ... 340
Глава 46 Огромные числа ... 350

Глава 47 Системы счисления 358
Глава 48 Железная логика 371
Глава 49 Сложные массивы 377
Глава 50 Неспортивные рекорды (записи) 390
Глава 51 Указатели в море памяти 397
Глава 52 Динамические переменные 407
Глава 53 Массив указателей 413
Глава 54 Односвязные списки 421
Глава 55 Слова, слова, слова… 435
Глава 56 И снова очереди, и снова стеки… 441
Глава 57 Графомания 450
Глава 58 По графу шагом марш! 463
Глава 59 Крупные проекты 480
Глава 60 Мелкие хитрости 496
Глава 61 «Кубики» программиста (ООП) 501
Глава 62 Всё только начинается! 520
Приложение А Установка и настройка IDE Borland Pascal 529
Приложение Б Консольная программа в среде Delphi 547
Приложение В Особенности IDE Pascal ABCNet 554
Приложение Г Зарезервированные слова 557
Приложение Д Ошибки компиляции 560
Приложение Е Ошибки исполнения 569
Приложение Ж Директивы управления компиляцией 571
Приложение З Назначение пунктов меню 573
Приложение И Стандартная кодировка символов MS-DOS 578
Приложение К Некоторые встроенные процедуры и функции 581
Приложение Л Перечень программ 583
Приложение М Пример олимпиадной задачи 587
Библиография 589

Содержание

Только для взрослых 15
Десять лет спустя 15
Чему нас учат семья и школа? 15
Крошка сын к отцу пришел 16
Азбучные истины 16
Что я могу ещё сказать? 17
Благодарности 18
Детям до 16-ти 19
Глава 1 Путь далек у нас с тобою… 20
Компьютер 20
Компилятор 20
Личный багаж 20
Компьютерная литература 21
В здоровом теле – здоровый дух 21
Вместе весело шагать по просторам! 21
Повторение – мать учения 22
Соглашения 22
Итоги 22
Глава 2 Вместо теории 23
Миф о думающих машинах 23
Загадочные коды 23
Языки программирования и компиляторы 24
Следующий шаг – IDE 26
Итоги 26
Глава 3 Консольный интерфейс 28
Что такое интерфейс? 28
Консольный интерфейс 28
Прикосновение к консольному интерфейсу 29
А почему не «окна»? 31
Итоги 31
Глава 4 Оружие – к бою! 32
Оружейный прилавок 32
Установка IDE Free Pascal 33
Настройка ярлыка 34
Первый запуск и настройка IDE Free Pascal 35
Установка справочной системы 37
Итоги 38
Глава 5 Программа номер один 39
Постановка задачи 39
Создание файла 39
Наполнение файла 40
Компиляция 41
Процедура вывода (печати) 42
Запуск программы 44
Итоги 44
А слабо? 45
Глава 6 Подготовка к следующему штурму 46
Ещё об исходных файлах 46
Управление окном редактора 47

Борьба с ошибками ... 48
Итоги ... 49
Глава 7 Развиваем успех ... 50
Операторы и разделители ... 50
Программа, стой! ... 52
Алгоритмы ... 52
Блок-схемы ... 53
Итоги ... 54
А слабо? ... 55
Глава 8 Постоянные и переменные ... 56
Константы ... 56
Идентификаторы ... 58
Переменные ... 58
Ввод и вывод данных ... 60
Итоги ... 61
А слабо? ... 62
Глава 9 Переменные: продолжение знакомства ... 63
Представьтесь, пожалуйста! ... 63
Из пустого в порожнее ... 64
Сцепление строк ... 65
Инициализация переменных ... 66
Типизированные константы ... 67
Итоги ... 67
А слабо? ... 68
Глава 10 Условный оператор ... 69
Стой! Кто идет? ... 69
Вопрос ребром ... 69
Пост номер один ... 70
Неполный условный оператор ... 71
Пост номер два ... 72
Итоги ... 73
А слабо? ... 74
Глава 11 Операторный блок ... 75
Операторные скобки ... 75
Красиво жить не запретишь ... 76
Комментарии ... 77
Итоги ... 78
А слабо? ... 78
Глава 12 Цикл с проверкой в конце ... 79
Подтянем дисциплину ... 79
Нанимаем репетитора ... 80
Вежливый часовой ... 81
Досрочный выход из цикла ... 83
Итоги ... 85
А слабо? ... 85
Глава 13 Правда и кривда ... 86
Есть ли жизнь на Марсе? ... 86
Информация и её мерило ... 86
Булевы переменные ... 87
Ввод и вывод булевых данных ... 88
Логические выражения ... 88
С высоты птичьего полета ... 89

Парад логических операций ... 93
Итоги ... 94
А слабо? ... 95
Глава 14 Дважды два – четыре ... 96
Поможем братьям нашим меньшим ... 96
Числа и действия с ними ... 96
Алгоритм экзаменатора ... 97
Экзаменатор, первый вариант ... 98
Итоги ... 99
А слабо? ... 100
Глава 15 Айда в Монте-Карло! ... 101
Куда ни глянь – то процедура, то функция! ... 101
Госпожа удача ... 102
Итоги ... 104
А слабо? ... 105
Глава 16 Делу время, а потехе час ... 106
Потемкинская лестница ... 106
Итоги ... 109
А слабо? ... 110
Глава 17 И вновь за парту ... 111
Цикл со счетчиком ... 111
Итоги ... 114
А слабо? ... 114
Глава 18 Аз, Буки ... 116
Символьный тип данных ... 116
Индексация ... 117
Длина строки ... 118
Распечатка строки ... 118
Итоги ... 120
А слабо? ... 120
Глава 19 Процедуры и функции: разделяй и властвуй ... 121
Снежный ком ... 121
Описание процедур ... 121
Процедуры с параметрами ... 124
Итоги ... 125
А слабо? ... 125
Глава 20 Процедуры: первый опыт ... 127
Мухи – налево, котлеты – направо! ... 127
Сверху вниз ... 129
Первые раны ... 131
Глобальные и локальные ... 132
Локально – это разумно! ... 134
Неподдающаяся строка ... 134
Итоги ... 134
А слабо? ... 135
Глава 21 Отладка ... 136
Отладчик ... 136
Жучки, вылезайте! ... 138
Ссылка на переменную ... 141
Итоги ... 143
А слабо? ... 143
Глава 22 О передаче параметров ... 144

Процедура обмена ... 144
Замена символов в строке ... 145
О передаче строк ... 146
Итоги ... 147
А слабо? ... 147
Глава 23 Функции ... 148
Объявление функции ... 148
Пример функции ... 148
Подсчет символов в строке ... 149
Возврат строк ... 151
Когда результат не важен ... 151
Неявная переменная Result ... 152
Итоги ... 153
А слабо? ... 153
Глава 24 Криптография ... 154
Секреты Юлия Цезаря ... 154
Суть проблемы ... 155
О кодировании символов ... 155
Чудесные превращения ... 157
Шифрование символа ... 158
Расшифровка символа ... 159
Итоги ... 161
А слабо? ... 162
Глава 25 Текстовые файлы ... 163
Файлы хорошие и разные ... 163
Формат текстовых файлов ... 163
Доступ к текстовым файлам ... 164
Чтение из файла ... 165
Последовательный доступ к файлу ... 167
Самореклама ... 167
Цикл с проверкой в начале ... 168
Итоги ... 169
А слабо? ... 170
Глава 26 Я не читатель, — я писатель! ... 171
Запись в текстовый файл ... 171
Пример записи в файл ... 172
Завершение шпионского проекта ... 172
Итоги ... 177
А слабо? ... 178
Глава 27 Дайте кораблю минутный отдых! ... 179
Ошибка ошибке рознь ... 179
Фатальные ошибки ... 179
«Простительные» ошибки ... 180
Опции компилятора ... 180
Обработка ошибок ввода-вывода ... 181
Директивы компилятора ... 182
Директиву – в студию! ... 182
Парад директив ... 183
Итоги ... 184
А слабо? ... 185
Глава 28 Редактор и справочная система ... 186
Небьющиеся окна ... 186

Буфер обмена ... 188
Справочная система ... 188
Итоги ... 190
Глава 29 Читайте по-новому ... 191
Полицейская база данных, версия 1 ... 191
Полицейская база данных, версия 2 ... 194
Итоги ... 196
А слабо? ... 196
Глава 30 Журнальная история ... 197
Статистика знает все? ... 197
Строим планы ... 198
Барабаним по клавишам ... 201
Первый блин ... 203
Блин второй ... 205
Спецификатор ширины поля ... 205
«Развесные» числа ... 207
Итоги ... 208
А слабо? ... 209
Глава 31 Финал журнальной истории ... 210
Буква за буквой ... 210
Нелишняя предосторожность ... 211
Достройка программы ... 212
Испытание ... 213
Итоги ... 216
А слабо? ... 216
Глава 32 Порядковые типы данных ... 217
Типы данных: простые и сложные ... 217
Целое братство ... 219
Капля, переполняющая чашу ... 220
Инкремент и декремент ... 223
Диапазоны ... 223
Перечисления ... 224
Порядковые типы ... 225
Разумный контроль ... 228
Итоги ... 228
А слабо? ... 229
Глава 33 Вещественные числа ... 230
Изображение вещественных чисел ... 230
Печать вещественных чисел ... 231
Типы вещественных чисел ... 231
Сравнение вещественных чисел ... 233
Типы данных пользователя ... 233
Совместимость и преобразование типов ... 235
Размеры переменных и типов данных ... 237
Итоги ... 239
А слабо? ... 239
Глава 34 Структура программы ... 241
Управляющие структуры ... 241
Структура программы ... 242
Структура процедур и функций ... 243
Обмен данными с подпрограммами ... 244
Встроенные процедуры и функции ... 246

Что дальше? 246
Итоги 246
А слабо? 247
Глава 35 Множества 248
В директорском кабинете 248
Первым делом, первым делом – оцифровка 248
Множества глазами математика 249
Числовые множества 252
Мощность множества, полные и неполные множества 254
Итоги 254
А слабо? 255
Глава 36 Множества в Паскале 256
Объявление множеств 256
Присвоение значений множествам 256
Операции с множествами 257
Сравнение множеств 258
Проверка на вхождение элемента в множество (операция IN) 259
Решение директорской задачи 259
Итоги 260
А слабо? 261
Глава 37 Ввод и вывод множеств 262
Вывод множества в текстовый файл 262
Ввод множества из текстового файла. 263
Директорская задача, первый вариант 264
Директорская задача, второй вариант 266
Итоги 267
А слабо? 267
Глава 38 Множества «в бою» 268
Активисты, шаг вперед! 268
Подвиг контрразведчика 268
В тридевятом царстве 270
Решето Эратосфена 273
Мелочь, а приятно 275
Итоги 276
А слабо? 276
Глава 39 Командная игра (массивы) 278
Снежная лавина 278
А где же волшебная палочка? 280
Массивы вокруг нас 280
Объявление массивов 281
Доступ к элементам (индексация) 284
Ввод и вывод массивов 285
Ошибки индексации 285
Итоги 287
А слабо? 287
Глава 40 Пристрелка на знакомых мишенях 288
Вопрос-ответ – добиваемся гибкости 288
Полицейская база данных – ускоряем поиск 289
Ещё раз о статистике 291
Итоги 293
А слабо? 293
Глава 41 По порядку, становись! 294

Пиратская справедливость ... 294
Пузырьковая сортировка ... 295
Электронная делёжка ... 297
Возвращение на футбольное поле ... 299
Итоги ... 301
А слабо? ... 301
Глава 42 Кто ищет, тот всегда найдет ... 302
Где эта улица, где этот дом? ... 302
Последовательный поиск ... 302
Двоичный поиск ... 303
Исследование двоичного поиска ... 306
Ах, время, время! ... 312
Логарифмы? Это просто! ... 313
Итоги ... 315
А слабо? ... 315
Глава 43 Сортировка по-взрослому ... 316
Сортировка выбором ... 316
Быстрая сортировка ... 318
Процедура быстрой сортировки ... 320
О рекурсии и стеке ... 323
Алгоритмы, на старт! ... 324
Итоги ... 330
А слабо? ... 330
Глава 44 Строки ... 331
Строка – особый род массива ... 331
Укороченные строки ... 332
Операции со строками ... 333
Подсчет слов в строке ... 337
Контекстная замена ... 337
Итоги ... 338
А слабо? ... 338
Глава 45 Очереди и стеки ... 340
Вовочка в потоке событий ... 340
Танцевальный кружок ... 341
Скитания товарного вагона ... 345
Сортировочная горка ... 346
Итоги ... 349
А слабо? ... 349
Глава 46 Огромные числа ... 350
Сколько звезд на небе? ... 350
Сложение «в столбик» никто не отменял ... 350
Великая стройка ... 351
Длинная арифметика ... 353
Итоги ... 356
А слабо? ... 357
Глава 47 Системы счисления ... 358
Из тьмы веков ... 358
Число и его изображение ... 358
Десятичная система ... 360
Двоичная система ... 363
Шестнадцатеричная система ... 364
Другие системы счисления ... 366

Изображение числа в заданной системе счисления 366
Обратное преобразование 368
Итоги 369
А слабо? 369
Глава 48 Железная логика 371
Два взгляда на компьютерные «кирпичики» 371
Логические операции в регистрах 372
Сдвиги влево и вправо 375
Итоги 376
А слабо? 376
Глава 49 Сложные массивы 377
На поклон к Науке 377
Имперское строительство 377
Крестики-нолики 382
Итоги 388
А слабо? 389
Глава 50 Неспортивные рекорды (записи) 390
Кушать подано! 390
Записи 390
Второй тайм 391
Дополнительное время 394
Итоги 396
А слабо? 396
Глава 51 Указатели в море памяти 397
Погружение в оперативную память 397
«Планировка» памяти 398
Указатели, первое знакомство 399
Объявление указателей 400
Копирование указателей, пустой указатель 401
Сравнение и проверка указателей 402
Разыменование указателей 403
Нетипичный указатель 403
Примеры с указателями 404
Итоги 406
А слабо? 406
Глава 52 Динамические переменные 407
Аппетит является к обеду 407
Одолжите памяти немножко! 407
Выделение памяти 408
Освобождение памяти 408
Предупреждён – значит, вооружен 409
Итоги 411
А слабо? 411
Глава 53 Массив указателей 413
Базу данных – в кучу 413
Сортировка массива указателей 418
Итоги 419
А слабо? 420
Глава 54 Односвязные списки 421
Чудесное сочетание 421
Проблема курицы и яйца 422
Вяжем список 422

Распечатка списка 426
Поиск в несортированном списке 427
Сортированные списки 428
Поиск в сортированном списке 432
Итоги 433
А слабо? 434
Глава 55 Слова, слова, слова… 435
Частотный анализ текста 435
Слово за слово 435
Структура записи 435
Алгоритм 436
А слабо? 440
Глава 56 И снова очереди, и снова стеки… 441
Шутить изволите? 441
Танцуют все! 444
Итоги 449
А слабо? 449
Глава 57 Графомания 450
Видимое представление графа 452
Внутреннее представление графа 453
Ввод и вывод графа 456
Итоги 462
А слабо? 462
Глава 58 По графу шагом марш! 463
Империя номер два 463
Структура узла 468
В рассыпную! 469
Аты-баты 475
Итоги 478
А слабо? 478
Глава 59 Крупные проекты 480
О модулях и разделении труда 480
Модули 481
Дробление модуля – «смертельный» номер 481
Компиляция проекта 486
Инициализация модуля 487
Структура модуля 490
О совпадении имен 490
Сборочный цех 492
Фирменные библиотеки 493
Динамически загружаемые библиотеки (DLL) 494
Итоги 495
А слабо? 495
Глава 60 Мелкие хитрости 496
Включаемые файлы 496
Условная компиляция 497
Итоги 500
А слабо? 500
Глава 61 «Кубики» программиста (ООП) 501
Фокус-покус 501
Вместо паяльника 502
На трех китах 502

Инкапсуляция ... 503
Наследование ... 505
Приборостроение ... 506
Гражданское строительство ... 507
Динамические объекты ... 512
Полиморфизм ... 513
Сокрытие полей и методов ... 516
Итоги ... 518
А слабо? ... 518
Глава 62 Самое интересное только начинается! ... 520
Крупицы мастерства ... 520
Программисты, на старт! ... 523
Приложение А Установка и настройка IDE Borland Pascal ... 529
История IDE Borland Pascal, состав дистрибутива ... 529
Установка IDE Borland Pascal ... 530
Организация рабочей папки ... 536
Создание и настройка ярлыка ... 536
Пробный запуск IDE ... 538
Предварительная настройка IDE ... 540
Русификация консольного окна ... 542
Turbo Pascal School Pak ... 546
Приложение Б Консольная программа в среде Delphi ... 547
Создание пустого консольного приложения ... 547
Настройка и сохранение консольного приложения ... 548
Русификация консольного приложения ... 551
Приложение В Особенности IDE Pascal ABCNet ... 554
Приложение Г Зарезервированные слова ... 557
Приложение Д Ошибки компиляции ... 560
Приложение Е Ошибки исполнения ... 569
Приложение Ж Директивы управления компиляцией ... 571
Приложение З Назначение пунктов меню ... 573
Приложение И Стандартная кодировка символов MS-DOS ... 578
Приложение К Некоторые встроенные процедуры и функции ... 581
Приложение Л Перечень программ ... 583
Приложение М Пример олимпиадной задачи ... 587
Библиография ... 589

Только для взрослых

> Конечной целью образования должно быть искусство конструктивного мышления.
>
> Никлаус Вирт

> Ученик — это не сосуд, который надо наполнить, а факел, который надо зажечь.
>
> Плутарх

Десять лет спустя

«Не верю!» — отмахнулся бы я, если б не видел это своими глазами. Меня можно понять, как и тех, кто, барахтаясь в мутных волнах 90-х, не помышлял о дальних планах. Но оптимисты неистребимы! Они устроили тогда в нашем городе конкурс юных программистов — KidSoft. Зрители тех состязаний терялись в догадках: где «желторотики» нахватались компьютерных премудростей? Сотворенных ими программ не постыдились бы и профессионалы! А ведь найти приличный компьютер тогда было не проще, чем хороший учебник программирования. «Что же будет лет эдак через 10-15, — спрашивал я себя, — когда компьютер войдет в каждый дом?». И мнились мне колонны юных гениев, бодро шагающие на свой конкурс.

Через годы судьба вновь свела меня с «компьютерной» молодежью. Наблюдая участников олимпиад, я невольно поверял свой прогноз. Во многом он оправдался: компьютер стал предметом быта, книжные полки ломятся от компьютерной литературы, а информатикой пичкают едва ли не с детского сада. Но где колонны юных гениев? Я их не вижу! Да, конечно, «кое-кто, кое-где у нас порой…». И всё же мне видится, что интерес молодежи к программированию несколько увял. Логика, ау! Где ты? Привыкнув следовать твоим законам, я поклялся раскрыть эту тайну.

Чему нас учат семья и школа?

Интернет и другие источники привели меня к парадоксальному выводу: интерес подростков к программированию угасал с развитием компьютерных технологий! Судите сами: чем мог заняться способный мальчишка в компании с каким-нибудь примитивным «синклером» начала 90-х? Наскучив двумя или тремя простенькими игрушками, он, в конце концов, брался за программирование и лепил ещё одну. А сейчас? Об «игрунах» молчу, поскольку даже творческий человек найдет в компьютере уйму интересного!

А школа, чему она учит? Поспешая за техническим прогрессом, школа пытается втиснуть в детскую голову едва ли не все достижения информационных технологий. Сомневаясь в разумности этой попытки, согласен всё же с тем, что компьютерная грамотность стала ныне грамотностью номер два. Но теперь она не

связана с программированием: в массе востребованы офисные приложения, электронная почта, Интернет.

Спорить с этим трудно, и я бы не стал. Но как быть юным программистам? Или эта профессия отмирает, и технологии будут развивать без них? Смешной вопрос, но иным не до смеха. Ведь в школе с программистами заниматься некому и некогда, — эстафету передали в ВУЗ. А там, на профильных факультетах, давно уже бьют в набат: познания новобранцев в программировании ничтожны, и обучать их приходится с азов.

Что в сухом остатке моих изысканий? Нужны ли нам хорошие программисты? — разумеется. Есть желающие ими стать? — конечно! Но нет школы, которая их научит. Так пусть вундеркинды учатся сами, почему нет? Вот компьютер, вот полка с учебниками, — полный вперед! Так ли это? Присев на корточки, я вошёл в положение юного нахала, дерзнувшего двинуться этой тропой.

Крошка сын к отцу пришел

Итак, я стал мальчишкой лет двенадцати. В доме есть компьютер, за который изредка садятся и родители. Но, главным образом, — это мой инструмент. Мне многое по плечу: скопировать файлы или напечатать что-то? — запросто! Признаюсь, однако, что игрушки надоели, и хочется освоить программирование. Увы! Родители в этом не разбираются, программистов среди друзей нет, а учителю информатики возиться со мной недосуг. Ладно, попробую сам. Раздобыв пару книжек и ободрившись примером Ломоносова, отважно берусь за дело.

Прикусив от старания язык, я терпеливо «сверлю» страницу за страницей. Вот алфавит языка, идентификаторы, константы, выражения... Кое-что понятно, но... Мамочка! когда же я напишу хоть простенькую программку? Открыв другую книгу, нахожу то же самое — подробное описание языка программирования, или так называемую теорию. Убойная доза теории свалит с копыт даже крепкую казачью лошадь, — так устоит ли мой нежный организм? Энтузиазм вянет. «Нет, — думаю, — новый Ломоносов подождёт, может, в школе когда-нибудь научат». Что будет в школе, вы уже знаете.

Так может, мальчишке попались плохие книги? Не думаю, хорошие учебники встречаются, некоторые написаны основательно. «И всё же, всё же, всё же...». Всё же книги эти адресованы другому читателю; по сути это технические руководства, рассчитанные на закаленных, зрелых бойцов. Как же обучать юнцов?

Азбучные истины

Тогда я мысленно приложил типовой учебник программиста к преподаванию грамоты в первом классе. По замыслу такого учебника, прежде, чем нацарапать «мама мыла раму», первоклашка обязан не только выучить все буквы, но и познать премудрости орфографии, синтаксиса, склонения, спряжения и так далее. Абсурд, не так ли? Ведь я отлично помню, что слово «мама» я вывел, постигнув лишь две буквы. Полагаю, что русский язык не проще языка программирования. И если для первого удалось создать азбуку — чудную вещь! — то нельзя ли чем-то подобным

снабдить начинающих программистов? Явилась мысль сделать обратную проекцию и создать «букварь» для программиста. На мой взгляд, такой «букварь» должен строиться на следующих принципах.

Постепенность. Излагать материал следует мелкими, легко постигаемыми порциями. Высота одолеваемых учеником ступенек не должна вызывать ощущения тупика, — маленький успех окрыляет, вселяя уверенность.

Практичность. Программирование — инженерная наука. Теория и практика здесь неразделимы, пропитывают друг друга, — ученик не должен ощущать границ между ними. Букварь программиста должен сочетать в себе учебник и хрестоматию. Примеры программ должны быть либо простыми, либо очень простыми (по крайней мере, на первых порах). Необходимо показать полные решения задач с разъяснениями, — «учились бы, на старших глядя».

Поправка на возраст. Сюжеты задач в «букваре» должны учитывать психологию подростка, а он склонен учиться играючи. Хорошо, если примеры похожи на настоящие «взрослые» проекты (разумеется, упрощенные). Непоседу не увлечешь задачей в роде «посчитать по формуле такой то» или «найти сумму элементов массива». А вот полицейская база данных или экзаменующая программа — это серьезно! Разжечь аппетит юного инженера — едва ли не главная цель обучения.

Маловажное — за борт! От изложения некоторых второстепенных деталей языка лучше воздержаться. Например, можно «забыть» о записях с вариантами и не вспоминать о типизированных файлах. Не отвлекая внимания на эти детали, сосредоточиться на главном. Усвоив это главное, ученик доберёт остальное из «взрослых» учебников.

Вот, пожалуй, и всё. В идеале такой букварь будет и самоучителем для подростка, и конспектом для преподавателя компьютерного кружка.

Что я могу ещё сказать?

Итак, цель поставлена, но достигнута ли? — судить читателям. Вкратце содержание книги таково.

В главах с 1-й по 4-ю после краткой обзорной информации даны практические рекомендации для подготовки рабочего места. Далее все подносимые порции теории немедленно воплощаются на практике.

В главах с 5-й по 31-ю рассматриваются простые типы данных и базовые алгоритмические структуры. Здесь же рассказано о текстовых файлах и даны основные сведения об организации среды программирования.

В главах с 32 по 35 подводится теоретическая черта под пройденным материалом и закладывается фундамент для перехода к сложным типам данных.

Главы с 36 по 58 повествуют о сложных типах данных и связанных с ними алгоритмах. Здесь рассмотрены множества, массивы, записи и динамические структуры.

В главах 59 и 60 раскрыт секрет разработки многофайловых проектов, а глава 61 знакомит с принципами объектно-ориентированного программирования.

Последняя 62-я глава — это попытка заглянуть в будущее и указать читателю дальнейшие цели и пути их достижения.

Полагая, что книга будет, на какое-то время, единственной помощницей новичка, автор счел нелишним включить несколько справочных приложений.

Итак, отойдя от общепринятого порядка изложения теории, я стремился вовлечь читателя в активное осмысление конструкций языка, приглашая его к соавторству с Никлаусом Виртом. «Почему в языке сделано именно так, а не иначе?» — этот вопрос то и дело встает перед учащимся. Решая задачи, он видит, что элементы языка не с потолка свалились, а придуманы для решения типовых проблем. Отсюда следует порядок изложения: 1) проблема, 2) размышление, 3) решение. Сначала ставится задача. Затем обсуждается, как её решить уже известными средствами языка, или почему её нельзя решить этими средствами. После этого даётся надлежащая порция теории, и приводится решение либо с новым применением уже известных конструкций языка, либо с привлечением новой конструкции. Напоследок подводится теоретический итог очередной главы. Так теория с практикой следуют рука об руку.

Несладко быть в шкуре новичка: там и сям натыкаешься на проблемы! Автор снабдил почти все решения полными листингами работающих программ с подробными пояснениями. Кому то они покажутся избыточными, назойливыми. Но согласится ли с этим паренёк, корпящий над книгой где-нибудь в глухом поселке? Ведь для него, одинокого бойца, любая «непонятка» порой вырастает в неодолимую преграду!

Благодарности

Я признателен всем, кто высказался о «Песнях» на форумах и в личной переписке, — мы работали над книгой вместе! Особо благодарю форум freepascal.ru, а также Артёма Проскурнёва и Владислава Джавадова, подаривших мне массу полезных советов.

Пишите, мой адрес всё тот же: oleg.derevenets@gmail.com

Свежую редакцию книги и сопутствующие файлы можно скачать здесь:

http://oleg-derevenets.narod.ru

Детям до 16-ти

«У меня есть мечта!», — признался один известный человек. А у кого её нет? Вы тоже мечтаете, и я знаю, о чем. В детстве мне хотелось поскорее вырасти, и я завидовал взрослым: никто им не указ, делай, что хочешь! Вы мечтаете о том же? Но как стать большим раньше назначенного природой срока? Наклеить усы и бороду? Пробовал, — не помогает. Или прибегнуть к «сильным» средствам: крепким напиткам и табаку? Даже не пытайтесь, — вы постареете, не повзрослев!

Со временем дошло до меня, что взрослый — это тот, кто владеет профессией и занят полезным делом. Стало быть, став профессионалом, можно повзрослеть? Хороших профессий полно, выбирайте любую. А не стать ли вам программистом? «В моем возрасте? Возможно ли?» — усомнятся некоторые. Так вот вам зеркало, смотрите, кто там? Сметливый человек с цепкой памятью и страстным желанием поскорее созреть! Кому, как не вам, взяться за это дело?

Согласны? Тогда уточним, кто такой программист. Некоторые склонны считать программистом любого, кто работает с компьютером. Питая глубокое уважение ко всем мастерам разных сфер: системным администраторам, дизайнерам сайтов и многим другим, мы не станем величать их программистами. Нет, программист — это волшебник, оживляющий бездыханные железки. Порой их называют хакерами или кодировщиками. Слово «хакер» мне не по душе, поскольку пристало к взломщикам программ и паролей. А кодирование? Это всего лишь часть работы программиста, состоящая в написании программы по готовому алгоритму. Нет, настоящий программист — не презренный «кодировщик», он видит шире и копает глубже.

Итак, я зову вас в программисты-профессионалы, а это значит, что с «чайниками» нам не по пути. Чем довольствуется «чайник»? — верхушками знаний, а мы устремимся к вершинам. И, пускай, эти вершины пока далеки, мы не помчимся за быстрым результатом, прыгая через три ступеньки. Нет, наши шаги будут основательными, а обретенные знания глубокими, — «небоскреб» вашего будущего должен опираться на прочный фундамент!

Со мной вы изучите язык программирования Паскаль – один из самых красивых и полезных. Он и его потомки – Модула, Ада, Оберон – неспроста слывут самыми надежными, – оборонка, космос и авиация предпочитают эти языки. Идеи, из которых они сотканы, благотворно повлияли на новейшее программирование. Прошедшего школу Паскаля отличает ясный, экономный и надежный стиль письма. Манера эта проявляется и тогда, когда программист пишет на иных языках.

Овладев Паскалем, вы откроете себе много дорог. Одна из них – основанная на Паскале мощная визуальная среда программирования Delphi. Другое направление – школьные олимпиады по программированию, где большинство участников тоже пишут на этом языке. В конце концов, Паскаль облегчит вам изучение и других языков. Так, освоение Паскаля станет первым и самым важным шагом к вашей мечте!

Глава 1
Путь далек у нас с тобою…

Итак, вы из тех смельчаков, что готовы карабкаться со мной на вершину по имени Паскаль? Я помогу вам, с чего начнем? Соберем «рюкзачок» на дорогу, — сложим в него то, без чего не обойтись в этом путешествии.

Компьютер

Без чего нам не обойтись? «Без компьютера!» — никто не сомневался в столь разумном ответе. Программист без компьютера — всё равно, что всадник без коня или мушкетер без шпаги! Однако ж, какой компьютер нам сгодится? Ведь мощь этих машин стремительно растет, удваиваясь каждые два года. Каковы наши требования? К счастью, они скромны, — нам подойдет любой IBM-подобный компьютер. Найти «станок», выпущенный в прошлом веке теперь можно разве что на пыльном чердаке. Но даже такой «старичок» нам бы вполне сгодился. Поскольку большинство компьютеров оснащены одной из версий операционной системы Windows, я учту это в ходе дальнейших пояснений.

Компилятор

Хорошо, компьютером обзавелись, что ещё? Нужна специальная программа — компилятор, переводящая программу из текстового вида в исполняемый файл. Существуют несколько компиляторов с языка Паскаль, их можно взять в школе, либо скачать в Интернете. В 4-й главе я расскажу о том, как установить и настроить компиляторы в операционной системе Windows.

Личный багаж

Бросив в «рюкзак» компьютер с компилятором, осмотрим теперь ваш «личный багаж», — намекаю на ваши познания, конечно. Ведь компьютер — хитрая штука, насколько вы владеете им? Что вам известно о файловой системе? Умеете ли искать, копировать и переименовывать файлы? А создавать каталоги (папки) и набирать несложный текст вам по силам? Хотя бы в таком простом редакторе как Notepad (блокнот). Другими словами, от вас требуются навыки начинающего пользователя. Я не буду тратить бумагу на разъяснение этих премудростей. Если же вы слабо владеете компьютером, обратитесь к старшим.

Некоторые ставят программистов в один ряд с математиками, подозревая у тех и других математический склад ума. Отчасти это так, и в своей работе программисты нередко используют сведения из математики. А что требуется в этой части от вас? Пока ничего, что выходит за рамки школьной программы, — знаний 3-го класса вполне достаточно. Если же вам знакомы основы алгебры, то есть вы понимаете, что любое число можно обозначить буквой, тогда... тогда считайте себя профессором!

В освоении языка Паскаль вам помог бы другой язык — английский. Нет, он не лучше других. Но так уж вышло, что компьютеры и программирование зародились в англоязычных странах, и с тех пор английский стал языком тех, кто по роду занятий связан с компьютерами. Я допускаю, что вы пока не сильны в английском, или изучаете другой иностранный язык. Тогда следует знать хотя бы буквы латинского алфавита. По ходу изложения я буду переводить попадающиеся там и сям английские слова, и пояснять их. Но и сами не сидите, сложа руки! Положите под руку англо-русский словарь (или установите словарь на компьютере) и переводите все непонятные слова. Тогда через несколько месяцев вам будут доступны статьи на компьютерные темы. Короче — налегайте на английский!

Что ещё? Программисту (не кодеру!) необходимо широкое образование, — этого требует специальность. Не пренебрегайте школьными предметами, в жизни всё пригодится!

Компьютерная литература

В этой книге достаточно сведений для усвоения азов программирования. Однако, сосредоточившись на разъяснении простых вещей, я сознательно промолчал о некоторых средствах Паскаля. Со временем вас заинтересуют и другие возможности языка, и тогда вы откроете книги для подготовленных читателей. Названия некоторых из них найдете в списке литературы (библиография в конце книги). Среди ресурсов Интернета новичкам я рекомендую эти:

http://freepascal.ru — форум и много полезной информации;

http://ptaskbook.com/ru/tasks/index.php — подборка простых задач для начинающих.

В 62-й главе приведен ещё ряд ссылок для тех, кто заинтересуется олимпиадным программированием и углубленным изучением алгоритмов.

В здоровом теле – здоровый дух

Некоторые фантасты рисовали людей будущего с огромной головой и ниточками вместо рук и ног. Кажется, что компьютерные фанаты, сутками молящиеся на своих «идолов», подтверждают это предсказание. Неужели фантасты правы? Вы согласны стать колобком с висящими ниточками? Нет? Так соблюдайте меру, — свежий воздух и спорт сохранят силу серых клеточек вашего мозга. «В здоровом теле — здоровый дух» — это ещё древние греки знали.

Вместе весело шагать по просторам!

Хорошо заниматься программированием с друзьями! Товарищей можно найти где угодно: в школе, во дворе, в Интернете. А если кому-то из них потребовалась помощь? Неужели откажете? Ведь лучший способ научиться самому — это учить других! Помогайте друзьям, и тогда точно станете взрослыми!

Повторение – мать учения

В сложных вопросах не разобраться сходу, - так уж устроен наш мозг, что требует времени на усвоение нового. Но, порой случается и наоборот: кажется, что всё ясно, однако при повторном чтении проявляются новые детали, и знакомые предметы видятся с иной стороны. Не забывайте, что повторение — мать учения!

Соглашения

Некоторые слова этой книги будут выделены особыми шрифтами, вот примеры таких выделений:

Borland — особо выделенный текст, а также названия фирм, программных продуктов и т.п.;

«File Name» — имена файлов и каталогов;

Begin — служебные слова языка программирования (идентификаторы).

F9 — название пунктов меню и горячих клавиш

Место хранения дисковых файлов я буду называть каталогом или папкой. В некоторых книгах встречается термин директория, который означает то же самое.

Итоги

В конце каждой главы подводятся краткие итоги сказанному. Сейчас подведем первые итоги и двинемся дальше.

- Изучать программирование мы будем на практике, – для этого вам нужен любой персональный компьютер с операционной системой Windows.
- Для формирования программ потребуется компилятор – программа, преобразующая текст в исполняемый файл.
- От вас требуются, по крайней мере, знания и навыки начинающего пользователя компьютера, а также знание букв латинского алфавита.
- По мере профессионального роста вам не обойтись без дополнительной литературы по программированию.
- Крепите свое здоровье и помогайте друзьям!

Глава 2
Вместо теории

Жаль, что вы не застали компьютеров первых поколений! Тогда они назывались ЭВМ — электронные вычислительные машины, — слово «компьютер» ещё не было в ходу. ЭВМ помещались в залах солидных размеров, — уж машины так машины, было на что посмотреть! Вход в эти дворцы охраняла угрюмая стража, отсекавшая тех, кто не сдал экзамен по так называемой теории. Иначе говоря, чтобы сесть за ЭВМ, надо было сначала изучить язык программирования и сдать экзамен по нему. Это суровое требование объяснялось тем, что на одну машину рвались сотни пользователей. Потому машинное время ценилось на вес золота, — уж если добрался до ЭВМ, так занимайся делом, а не барабань без ума по клавишам!

Теперь не то, и нравы смягчились: вас пускают за компьютер, не экзаменуя по теории. Но значит ли это, что теория не нужна? — нет. Однако изучать теорию приятней и полезней на практике: мы будем создавать работающие программы. Много программ. Вижу, как вы прыгнули за компьютер и в нетерпении потираете ладошки. Ой, как я вас понимаю! Но ради будущих успехов потерпите до следующей главы. Сейчас я приоткрою завесу тайны, о которой так загадочно молчат взрослые. Не осознав некоторых вещей, невозможно двигаться дальше.

Миф о думающих машинах

Миф — это красивая выдумка, сказка. Компьютеры породили миф о думающих машинах. Ну как отказать в интеллекте этим чудесным созданиям? Восхищение не искушенного в компьютерах человека понятно, но порой приводит к недоразумениям. Вот слышу в новостях: в таком то аэропорту остановлены полеты из-за сбоя управляющих компьютеров. Вероятно, сами компьютеры здесь ни при чем, — современные машины очень надежны, а в особо важных применениях дублируются. И, хотя сбой компьютерной «железки» не исключен, неполадки в системе управления скорей всего на совести программистов. Как ни сложен компьютер, программы, которые он выполняет, в тысячи раз сложнее, а значит и возможность ошибок в них выше. Уверяю вас: компьютер — это всего лишь примитивный автомат, выполняющий команды, заложенные в него программистами.

К чему я клоню? Замыслив стать профессионалом, отбросьте миф о думающих машинах. Компьютер ни о чем не думает, не мечтает, и не спотыкается. Не пеняйте на него, когда ваши программы «захромают», — ищите ошибки у себя.

Загадочные коды

Вам известно, конечно, что исполняемые программы — это файлы с расширением EXE. Заглянем внутрь такого файла, как он устроен? С этой целью я воспользовался программой, подобной Total Commander. Выбрав один из

исполняемых файлов, я нажал клавишу *F3* — просмотр файла — и увидел следующую картину (рис. 1).

```
00026FB0: D8 8B C6 8B F7 8B F8 80|FB 16 75 7D 80 7D FF 14
00026FC0: 75 77 6A 2A 5E 56 E8 BD|68 FF FF E8 AE 6C FF FF
00026FD0: 6A 57 E8 8A 6C FF FF 56|E8 AB 68 FF FF E8 9C 6C
00026FE0: FF FF 6A 55 E8 78 6C FF|FF 56 E8 99 68 FF FF E8
00026FF0: 8A 6C FF FF 6A 57 E8 66|6C FF FF 56 E8 87 68 FF
00027000: FF E8 78 6C FF FF 6A 56|E8 54 6C FF FF 56 E8 75
00027010: 68 FF FF E8 66 6C FF FF|6A 54 E8 42 6C FF FF 56
00027020: E8 63 68 FF FF E8 54 6C|FF FF 6A 56 E8 30 6C FF
00027030: FF 83 C4 30 E9 ED 00 00|00 80 7D FF 16 74 4B 80
00027040: FB 16 74 46 80 3D 6D 7F|4B 00 00 74 16 E8 64 69
00027050: FF FF 6A 43 E8 08 6C FF|FF 56 E8 A0 A1 FD FF 59
00027060: 59 EB 27 E8 4E 69 FF FF|6A 57 E8 F2 6B FF FF 56
00027070: E8 19 7A FF FF E8 3C 69|FF FF 6A 56 E8 E0 6B FF
00027080: FF 56 E8 8E 75 FF FF 83|C4 10 80 FB 14 74 46 80
00027090: 3D 6D 7F 4B 00 00 74 16|E8 19 69 FF FF 6A 42 E8
000270A0: BD 6B FF FF 57 E8 55 A1|FD FF 59 59 EB 27 E8 03
```

Рис. 1 – «Внутренность» исполняемого файла

Что бы это значило? Я, к примеру, здесь ничего не понимаю! Мы видим код программы, который понимает только процессор компьютера. Вероятно, наши человеческие представления о здравом смысле очень далеки от компьютерных! Откуда взялся этот код? Надо ли программистам разбираться в этой тарабарщине? К счастью, большинству из них этого не требуется, — на выручку приходят языки программирования.

Языки программирования и компиляторы

Разумеется, вы слышали об этих языках, к настоящему времени их насчитывают тысячи. Зачем так много? Причины разные. С одной стороны, это объясняется разнообразием решаемых задач, а с другой — течением времени. Многие ранние языки устарели и отмирают, им на смену приходят новые. Однако все их объединяет одно — языки создавались, чтобы избавить человека от программирования на «тарабарском» языке процессора.

Кстати, знаете ли вы китайский язык? А японский или арабский? Теперь представьте себя президентом, принимающим послов этих стран, как вы будете с ними общаться? Очевидно, пригласите переводчиков. В таком же положении находится и процессор компьютера, знающий только свой «тарабарский» язык, а все прочие понимающий через переводчиков. Переводчики — это специальные программы — компиляторы. Правда, в отличие от людей, способных переводить в обе стороны, компиляторы переводят лишь с человеческого языка программирования на «тарабарский» язык процессора.

Рассмотрим рис. 2, где представлена упрощенная схема перевода с трех языков программирования: Паскаля, Си и Фортрана.

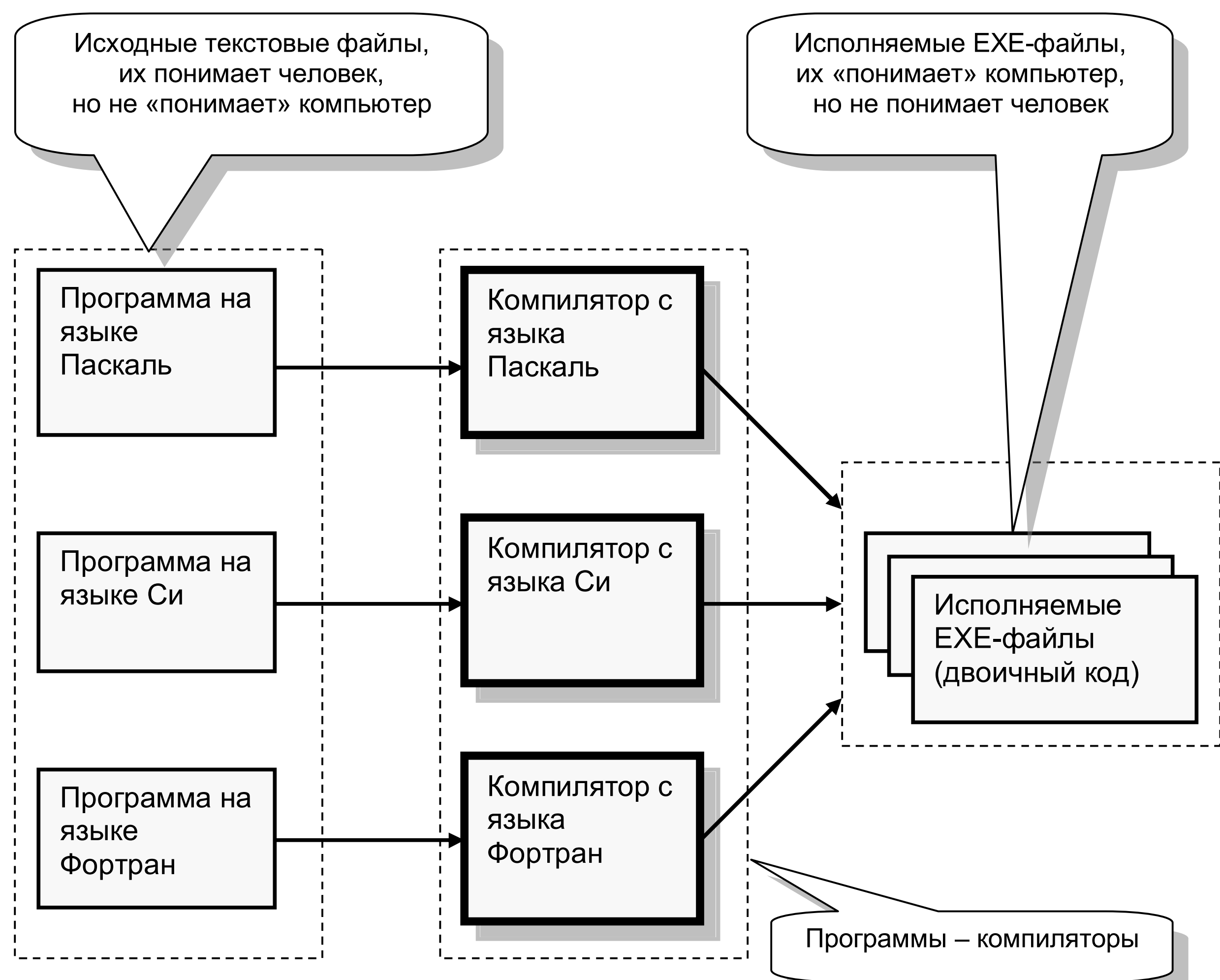


Рис. 2 – Схема применения компиляторов

Что мы видим? Работу над программой начинают с подготовки текстового файла, где на выбранном языке записывают порядок действий для решения поставленной задачи. В текстовом файле можно напечатать что угодно — стихи, роман, или программу. Сохранив файл на диске, вы можете в любой момент вновь открыть его, полюбоваться, отредактировать и снова сохранить. Это ещё не программа, а лишь её текст, заготовка. Такой файл называют исходным текстом или, на жаргоне программистов, — «исходником», «сырцом» (русское слово «сырьё» отчасти созвучно английскому Source — «источник»). Исходные файлы показаны на рисунке слева. Вот пример небольшой программки на языке Паскаль.

```
Var a, b : integer;
Begin
      Readln(a,b);
      Writeln('a*b = ', a*b);
End.
```

Конечно, вам она ещё не понятна. Но, согласитесь, в отличие от загадочного машинного кода, здесь чувствуется возможность что-то понять.

Итак, исходный текст иногда понятен автору программы, но неясен процессору. Потому после подготовки текста программист вызывает компилятор, переводящий текст в код процессора. Для каждого языка существуют свои правила и свой компилятор, вот его-то и надо запустить. Полученный в результате компиляции исполняемый EXE-файл далее «живет своей жизнью»: его можно запускать на выполнение, копировать, проверять на вирусы и заражать ими, — с исходным файлом он уже не связан. А если захочется что-то изменить в программе? Тогда без исходника не обойтись. Надо вернуться к нему, исправить редактором текста и вновь вызвать компилятор для перевода на «тарабарский» язык. Поэтому исходные тексты берегут, как зеницу ока, а то и секретят, если программа имеет коммерческое или военное значение.

Следующий шаг – IDE

Итак, для создания программы нужны, по меньшей мере, два инструмента: редактор текста и компилятор. Но на практике их требуется больше, — ведь без отладчика и справочной системы трудно обойтись. Нужда в нескольких инструментах доставляла когда-то программистам массу неудобств. Приходилось многократно «бегать по кругу», запуская эти программы одну за другой, пока результат не приближался к задуманному.

Но с появлением персональных компьютеров всё изменилось: была создана интегрированная среда разработки, или сокращенно ИСР. В компьютерной литературе чаще применяют англоязычное сокращение — IDE (Integrated Development Environment), мы тоже примем его.

Так что же такое IDE? Слово «интегрированная» значит «объединяющая». IDE — это мощная программа, объединяющая в себе и редактор, и компилятор, и отладчик, и справочную систему по языку. С появлением IDE программисты будто пересели с дребезжащей телеги в роскошный автомобиль, оплатив покупку быстрой и качественной работой. В скором времени мы «оседлаем» одну из таких IDE к языку Паскаль.

Итоги

- Отбросьте миф о думающих машинах, – действия компьютера определяются только вами, его ошибки – это ошибки программиста.
- Человек и компьютер «говорят» на разных языках. Процессор компьютера «понимает» лишь язык своих кодов, в котором трудно разобраться человеку.
- Для программирования изобретено много языков. На этих языках человек излагает порядок решения задачи в понятной для него форме.
- Для перевода текстового файла с программой в исполняемый EXE-файл используют программы-компиляторы.

- Современная интегрированная среда разработки (IDE) объединяет в одной программе редактор текста, компилятор, отладчик и справочную систему.

Глава 3
Консольный интерфейс

Пришло время засучить рукава и сесть за компьютер, по которому вы так истосковались! Хотите ли взглянуть одним глазком в столь желанное завтра? — я покажу вам ваши будущие программы. Не удивляйтесь, мы познакомимся с программами, которых ещё нет. Вернее, ознакомимся не с программами, а с их интерфейсом. Что такое интерфейс? Знакомое слово, не так ли?

Что такое интерфейс?

Интерфейс — это механизм слаженного взаимодействия систем. Например, компьютеров в сети, либо человека и компьютера. В ходе этих отношений человек отправляет компьютеру команды или снабжает его данными, а тот возвращает ему результаты своей работы.

Кому не знаком удобный и красивый оконный интерфейс? Здесь к услугам пользователя даны меню, кнопочки и прочие удобные штучки. А чем отвечает компьютер? Да чем угодно! Ответом могут быть и текст, и картинки, и даже подвижные изображения: фильмы, мультики.

Но сейчас, до сотворения наших первых программ, я познакомлю вас с другим интерфейсом — консольным. Что это за интерфейс, откуда он взялся и чем хорош?

Консольный интерфейс

Первые компьютеры появились не сегодня и не вчера. Тогда не было дисплеев, и пользователь общался с компьютером посредством электрической пишущей машинки — консоли. Отсюда и название интерфейса — консольный. Инженер печатал команды и вводил строчки с данными, а компьютер печатал на бумаге результаты вычислений. Бывшие тогда операционные системы поддерживали лишь консольный интерфейс. Давно минули те времена, но консольный интерфейс, как самый простой и надежный, сохранился и в новейших операционных системах. Именно этим интерфейсом будут обладать наши первые программы.

На первый взгляд, в сравнении с привычными для нас окнами, консольный интерфейс кажется примитивным и неудобным. Вместо щелчков мышью здесь надо вводить команды, набирая на клавиатуре загадочные сочетания из английских букв. Но у каждой медали две стороны, и у консольного интерфейса есть свое достоинство. В чем оно? В том же, в чем и недостаток — в примитивности, а точнее — в простоте. Консольные программы требуют меньше ресурсов компьютера, да и пишутся проще.

Впрочем, консольные операционные системы не исключают развитых оконных интерфейсов. Напомню, что оконные «коммандеры» и «навигаторы»

появились в консольной MS-DOS. Со временем и вы научитесь создавать оконные программы.

Прикосновение к консольному интерфейсу

Теперь испытаем консольный интерфейс «на ощупь», обратившись к консольному интерфейсу вашей операционной системы. Однако ж, где найти его среди многочисленных окон? Воспользуйтесь пунктом главного меню, который в ранних версиях Windows назывался «Сеанс MS-DOS», а в более поздних — «Командная строка». Итак, для вызова окна консоли обратитесь в главное меню Windows:

Пуск → Программы → Стандартные → Командная строка

или

Пуск → Программы → Стандартные → Сеанс MS-DOS

Щелчок на этом пункте вызовет окно, похожее на это (рис. 3).

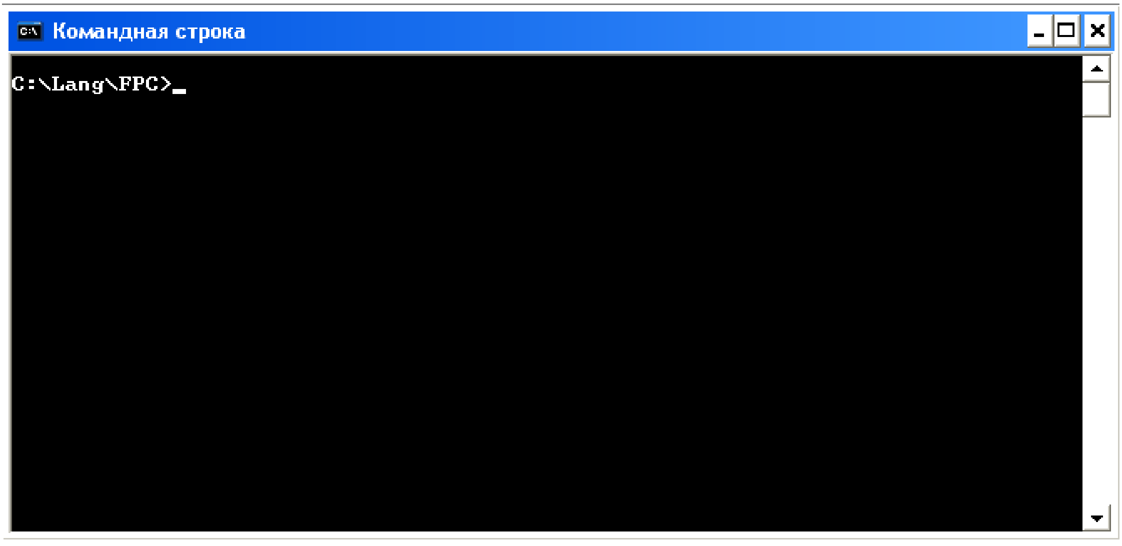


Рис. 3 – Окно командной строки (консольное окно)

Здесь выведено название текущей папки с угловой скобкой в конце. Эта строка с «уголком» называется строкой приглашения. Курсор, мигающий после угловой скобки, предлагает вам ввести какую либо из команд операционной системы. Таких команд насчитывается несколько десятков, их полное описание можно найти в справке по Windows. Сейчас испытаем три из них: **DIR** — распечатка каталога, **CLS** — очистка экрана, и **EXIT** — выход из окна консоли.

Напечатайте с позиции курсора слово **DIR** (большими или маленькими буквами — не важно) и нажмите клавишу *Enter*. Эта команда заставит систему распечатать информацию о файлах текущей папки. На моем компьютере я увидел вот что (рис. 4).

```
Командная строка
C:\Lang\FPC>dir
 Том в устройстве C имеет метку System
 Серийный номер тома: 94FA-76A1

 Содержимое папки C:\Lang\FPC

12.04.2010  10:56    <DIR>          .
12.04.2010  10:56    <DIR>          ..
12.04.2010  10:54    <DIR>          2.4.0
12.04.2010  10:57    <DIR>          doc
               0 файлов              0 байт
               4 папок  35 298 353 152 байт свободно

C:\Lang\FPC>
```

Рис. 4 – Распечатка содержимого текущей папки командой DIR

Выполнив команду, операционная система снова выводит «уголок», приглашая напечатать следующую команду. При желании повторите команду **DIR** ещё пару раз. А теперь введите команду **CLS** (очистка экрана), — в результате окно консоли очистится, и будет видна лишь строка приглашения. Наконец подача команды **EXIT** (выход) закроет консольное окно, и на этом сеанс завершится.

В прежних системах консольное окно можно было переключать в полноэкранный режим, и тогда оно занимало весь экран, а рабочий стол Windows исчезал. Это колдовство срабатывало при нажатии комбинации клавиш *Alt+Enter*, эта же комбинация возвращала экран в привычный оконный вид Windows. Но, в новейших на этот момент системах (Vista, Windows-7) полноэкранный режим уже не предусмотрен.

Командами консольного интерфейса можно выполнить всё то, что мы делаем через окна: создавать, копировать, удалять и переименовывать файлы, создавать каталоги и т.д. Вот ещё несколько команд, испытайте их:

VER — вывод версии операционной системы;

MEM — распечатка характеристик памяти;

TREE — распечатка дерева каталогов;

HELP — вывод списка всех команд операционной системы.

Здесь на время прервём знакомство с консольным интерфейсом, и вернемся к нему в главе 5, где напишем свою первую программу.

А почему не «окна»?

Читатели, наслышанные о таких мощных визуальных средах программирования как Delphi и Lazarus, обязательно спросят: почему бы нам не воспользоваться этими инструментами? Ведь создавать красивые оконные приложения в том же Delphi очень интересно и не так уж сложно!

Да, творить окошки с кнопочками в IDE Delphi на первый взгляд просто. Но эта простота скрывает непостижимые для новичка механизмы событийного и объектного программирования. А мы ведь договорились не прыгать по верхушкам, — оставим это развлечение «чайникам». Это первое.

Открою и второй умысел. Рассмотренный нами консольный интерфейс применяется для ввода и вывода данных не только на экран, но и в текстовые файлы. Профессионал обязательно соприкоснётся с такими файлами, а вы ведь будущий профессионал, не так ли? К тому же, с текстовыми файлами имеют дело и участники школьных олимпиад, в которых вы наверняка пожелаете сразиться.

Итоги

- Интерфейс – это механизм слаженного взаимодействия технических систем, например, двух компьютеров, либо человека и компьютера.
- Консольный интерфейс или интерфейс командной строки – это простой и надежный механизм, используемый для общения человека с компьютером. Он применялся в ранних поколениях ЭВМ, и жив по сей день.

Глава 4
Оружие – к бою!

Скоро вы напишете свою первую программу, и для этого заточим наше оружие, наш рабочий инструмент — интегрированную среду разработки (IDE). Тогда, подобно воину, вы сможете вмиг обнажать меч двойным щелчком мыши.

Оружейный прилавок

Напомню, что IDE — это мощная программа, объединяющая редактор текста, компилятор, отладчик и справочную систему. Для программирования на Паскале создано несколько таких IDE, рассмотрим кратко наиболее известные из них, а именно:

- Borland Pascal 7.0 (7.1);
- Free Pascal;
- Delphi;
- Pascal ABCNet.

Годятся ли эти IDE для обучения новичка? Каковы их особенности? Не нарушаем ли мы авторских прав? И что выбрать? Рассмотрим их в исторической последовательности.

Borland Pascal — первая среда такого рода, разработанная фирмой Borland ещё в эпоху консольной операционной системы MS-DOS. Последним версиям IDE присвоены номера 7.0 и 7.1. Изделие вышло удобным и надежным, и задало фактический стандарт в данной области (признаюсь, и поныне это мой любимый компилятор).

Delphi — новое детище фирмы Borland, сменившее покинутый ею Borland Pascal. Визуальное программирование — вот главная изюминка этого продукта. Даже не слишком умудренный программист соорудит с помощью Delphi вполне приличные оконные программы.

Free Pascal. Как ни хороши изделия фирмы Borland, их применение ограничено в смысле авторских прав. Потому энтузиасты Паскаля создали свободно распространяемую IDE, очень похожую на Borland Pascal. Проект Free Pascal развивается, и во многом он ушел дальше своего предшественника, конкурируя с Delphi. К настоящему времени выпущены версии для нескольких платформ и операционных систем.

Pascal ABCNet — эта остроумная, удобная и бесплатная IDE создана энтузиастами Южного федерального университета на основе технологии «точка Net», продвигаемой фирмой Microsoft. Буквочки «АВС» намекают на «азбучное», то есть образовательное направление проекта.

Подытожим наш краткий обзор и сделаем выбор. Новичкам будет удобно в среде Pascal ABCNet. Однако её основа — технология «.Net» — не отвечает требованиям школьных олимпиад по информатике (а многие захотят поучаствовать в них). Этим требованиям удовлетворяют другие IDE, которая из них лучше? Мощная визуальная среда Delphi избыточна и сложна для начинающих. Borland Pascal подкупает своей надежностью, но эта IDE слегка устарела. К тому же упомянутые продукты фирмы Borland не бесплатны. Что нам остаётся? Free Pascal? — выберем его своим основным «оружием».

Но и приверженцев других инструментов я не брошу на произвол судьбы. Примеры из этой книги не привязаны к какой-либо версии языка и сработают в любой из рассмотренных IDE (кроме разве что ABCNet, где часть примеров требует переработки). С установкой, настройкой и особенностями работы в этих средах программирования можно ознакомиться в следующих приложениях:

- Приложение А – Borland Pascal;
- Приложение Б – Delphi;
- Приложение В – Pascal ABCNet.

Установка IDE Free Pascal

Приступим к установке и настройке приглянувшейся нам IDE Free Pascal. Прежде всего, раздобудем её дистрибутив. Если вам доступен Интернет, войдите на сайт www.freepascal.org. Переключившись на страницу «Downloads», можно скачать дистрибутивы под любую аппаратно-программную платформу. Установочный файл для компьютеров на базе процессоров Intel с операционной системой Windows доступен по ссылке

www.freepascal.org/down/i386/win32.var

Здесь для закачки предложено несколько равноценных зеркал на случай, если другие окажутся неработающими (не качайте с SourceForge, где содержатся исходные файлы). На момент написания этих строк с зеркал скачивался файл

fpc-2.6.0.i386-win32.exe

Запустив его, и ответив на несколько несложных вопросов установщика, вы получите желаемый результат. Затрудняясь с выбором ответов, оставляйте то, что предложено по умолчанию. Тогда полный комплект IDE установится в папку «C:\FPC\2.6.0» (для версии 2.6.0), а на рабочем столе появится ярлык для запуска IDE. Если ярлык не появится, создайте его вручную на файл

c:\fpc\2.6.0\bin\i386-win32\fp.exe

Настройка ярлыка

Прежде, чем «дергать» за ярлык, настроим его, указав рабочую папку и параметры шрифта. Что такое рабочая папка? Это папка, где мы будем хранить свои программы, а их будет немало. Негоже сорить файлами по всему диску: ведь найти их будет трудно, а удалить по неосторожности — легко. Создайте для своих программ папку в подходящем месте. Я, к примеру, создал её со следующим путем:

C:\User\Pascal

Теперь в ярлыке, запускающем IDE, укажем путь к этой рабочей папке. Щелкните по ярлыку правой кнопкой мыши и в контекстном меню выберите пункт «Свойства». В открывшемся окне свойств щелкните на вкладке «Ярлык», и в поле «Рабочая папка» вместо пути, указанного по умолчанию, впечатайте путь к своей рабочей папке (рис. 5 слева).

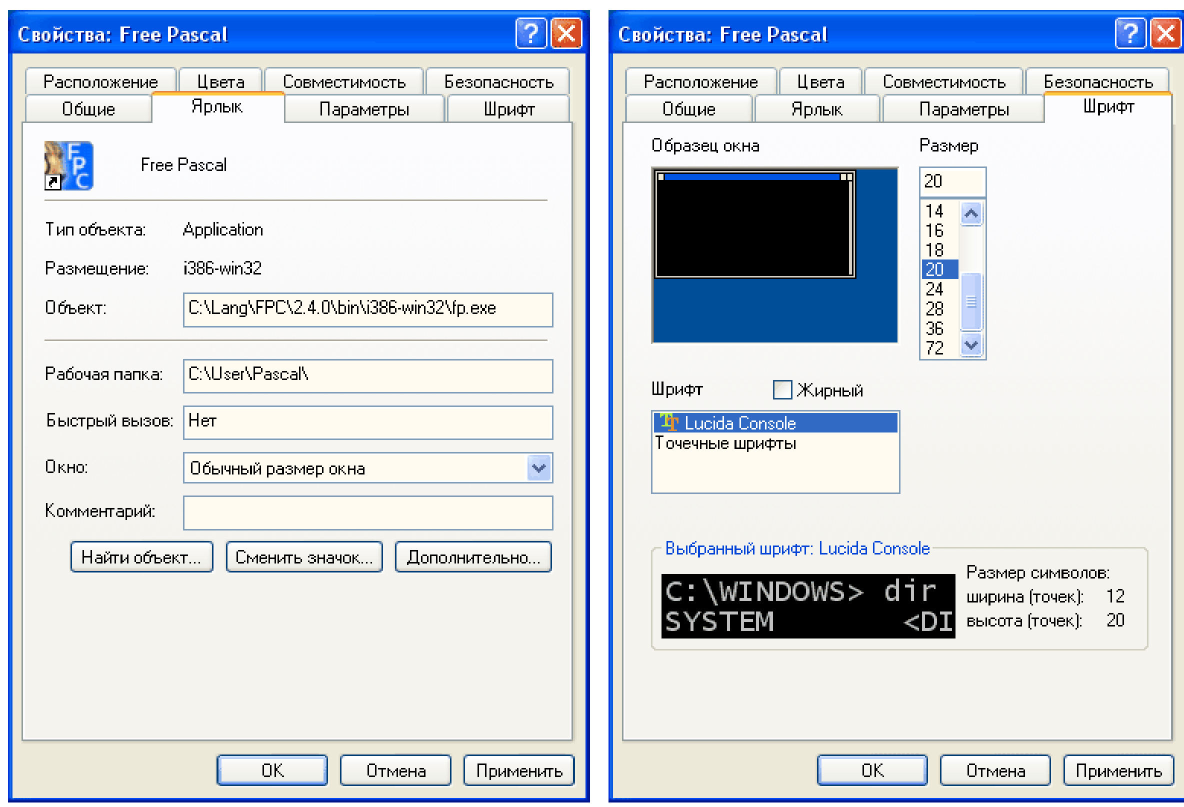


Рис. 5 - Окно свойств ярлыка

Затем переключитесь на вкладку «Шрифт» (рис. 5 справа) и задайте достаточно крупный размер шрифта, — пощадите свои глаза! В завершение нажмите кнопку *OK*.

Первый запуск и настройка IDE Free Pascal

Теперь смело «дергайте» за ярлык. При каждом своем запуске IDE ищет в рабочей папке файл с текущей конфигурацией. Поскольку при первом запуске такого файла ещё нет, вам будет задан вопрос о его создании (рис. 6).

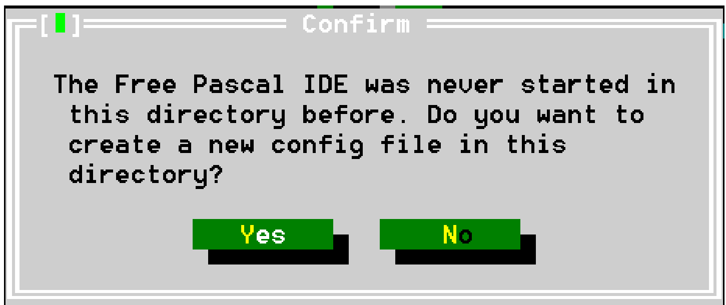


Рис. 6 - Вопрос с предложением создать новый конфигурационный файл

Дайте утвердительный ответ *Yes*, после чего IDE создаст в вашей рабочей папке файл «FP.CFG». В дальнейшем, когда вы будете менять настройки IDE, они будут автоматически сохраняться в этом файле.

Сейчас, например, мы укажем размеры окна IDE, выбрав один из предлагаемых вариантов. В окне IDE, обратитесь к пункту меню *Options* → *Environment* → *Preferences* (рис. 7)

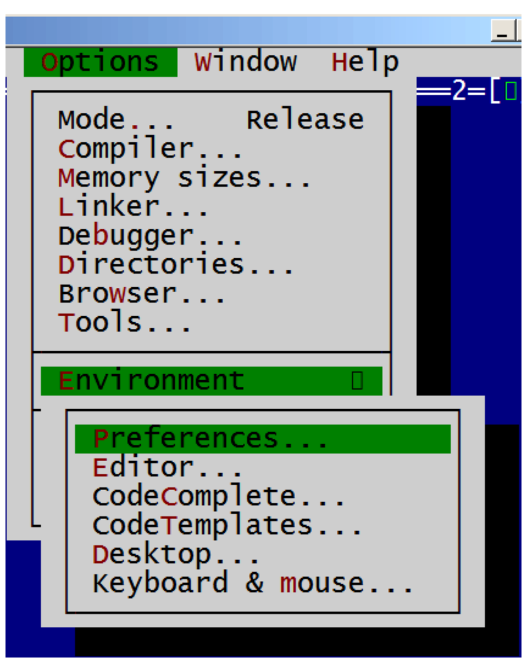


Рис. 7 - Пункт меню для настройки предпочтений

Появится окно для настройки предпочтений пользователя (рис. 8). Здесь в поле «Video mode» предлагается один из трех видеорежимов. Так, режим «80x25 color» соответствует стандартному дисплею в текстовом режиме, но для работы удобней будет задать 30 строк или более. Остальные опции оставьте

такими, как показано на рис. 8. После нажатия кнопки *OK* ваши предпочтения будут сохранены в конфигурационном файле.

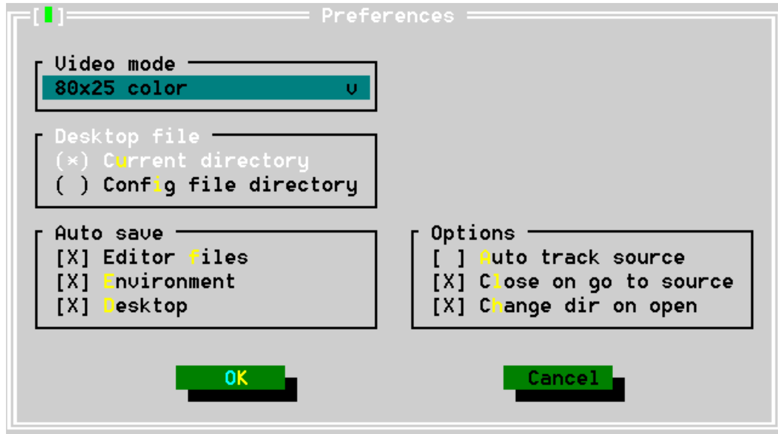


Рис. 8 - Окно для указания предпочтений пользователя

Пустое окно IDE Free Pascal примет вид, показанный на рис. 9. Буквы «FPC» на заставке означают «Free Pascal Compiler» — свободный компилятор с языка Паскаль.

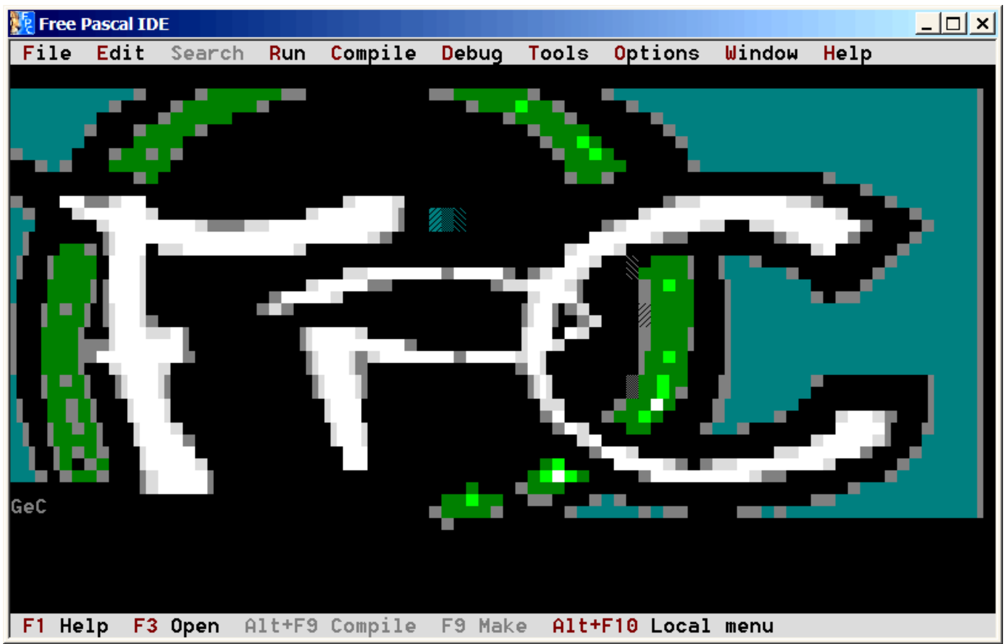


Рис. 9 - Пустое окно IDE Free Pascal

На этом установка IDE завершена, и можно приступать к первой программе. Правда, для полного счастья не хватает справочной системы. Впрочем, на первых порах она не нужна, а когда вас одолеет желание поупражняться в чтении английских статей, вернитесь к установке справочной системы.

Установка справочной системы

Справочная система скачивается отдельно от установщика, на момент написания этих строк работают следующие ссылки:

http://freepascal.org/docs.var — файл в формате PDF;

http://freepascal.org/down/docs/docs.var — файлы в форматах HTML и CHM.

Примечание. Со временем эти ссылки могут устареть, в таком случае начинайте поиск документации с корневой ссылки http://freepascal.org.

Скачав файл «doc-html.zip», распакуйте его в любое удобное место. Рекомендую создать для этого папку с именем «HELP» в той директории, где установлена IDE Free Pascal, — распакуйте zip-архив туда. Стартовый файл для открытия справки называется «fpctoc.html», он открывается любым браузером Интернета.

Эту же справочную систему можно встроить и внутрь IDE Free Pascal, выполнив следующие шаги.

Запустите IDE и активизируйте пункт меню *Help → Files…* (рис. 10).

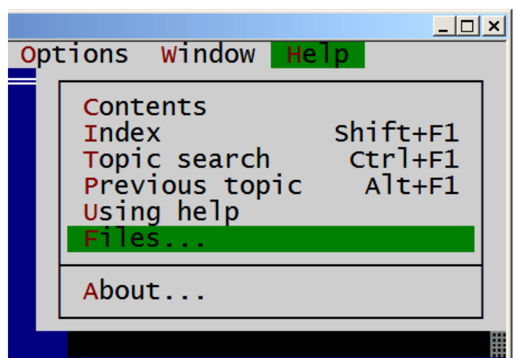


Рис. 10 - Пункт меню для настройки справочной системы

Появится окно для добавления файла справочной системы (рис. 11).

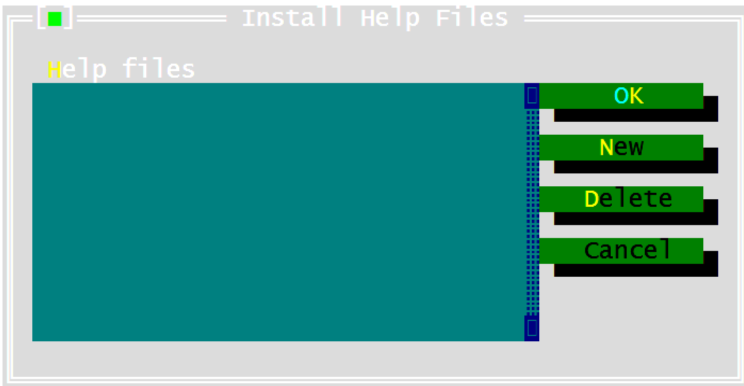


Рис. 11 - Окно для вставки файла справочной системы

Щелкните по кнопке *New* и откройте файл «fpctoc.html», — здесь IDE запросит подтверждение на создание индексного файла. Дайте положительный ответ и подождите несколько минут, пока буден создан индексный файл «fpctoc.htx». Если же файл «fpctoc.htx» уже был создан ранее, откройте сразу его (рис. 12).

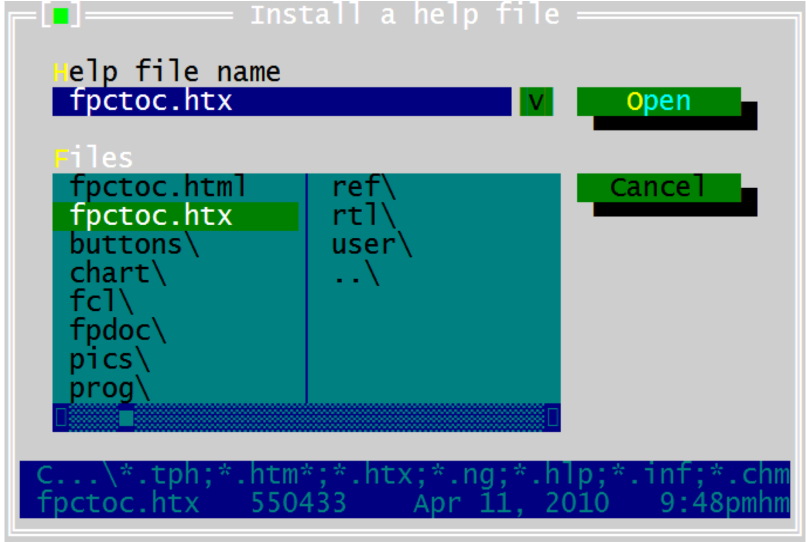


Рис. 12 - Указание файла справочной системы

Так или иначе, справочный файл появится в списке установленных, и вам останется лишь нажать кнопку *OK*. С этого момента вы сможете открывать справочную систему как отдельно (посредством файла «fpctoc.html»), так и внутри IDE Free Pascal клавишами *F1* и *Ctrl+F1*.

Обновление версий Free Pascal

В случае скачки и установки очередной версии Free Pascal вам придётся заново настроить ярлык, а также удалить из рабочей директории старые версии файлов «fp.cfg» и «fp.dsk».

Итоги

- Существует несколько сред разработки (IDE), пригодных как для обучения, так и для профессионального программирования на Паскале.
- С учетом ряда соображений, основной средой программирования мы выбрали IDE Free Pascal. Однако примеры из данной книги годятся почти для любой из упомянутых IDE.
- Для установки IDE Free Pascal нужен установочный файл (дистрибутив) и архив справочной системы.
- Создаваемые программы разумно хранить в отдельной рабочей папке, которую надо указать в свойствах ярлыка, запускающего IDE Free Pascal.

Глава 5
Программа номер один

Постановка задачи

Отныне мы будем повелевать компьютером, а он — исполнять наши капризы. Чем бы таким озадачить его? Ответ на подобный вопрос программисты называют постановкой задачи. Никто из них и пальцем не шевельнет, не прояснив суть предстоящей работы. Пусть наша первая программа выведет на экран слово «Привет!», — славно, когда тебя приветствует собственный компьютер!

Создание файла

Запустите IDE Free Pascal, — воспользуйтесь для этого ярлычком, который мы настроили в предыдущей главе. Затем создайте новый файл, выбрав пункт меню *File* → *New* (рис. 13). В области редактора появится пустое окно с заголовком «NONAME00.PAS», — это так называемый безымянный файл; две цифры в конце имени (00, 01, 02 и т.д.) помогают различать такие файлы.

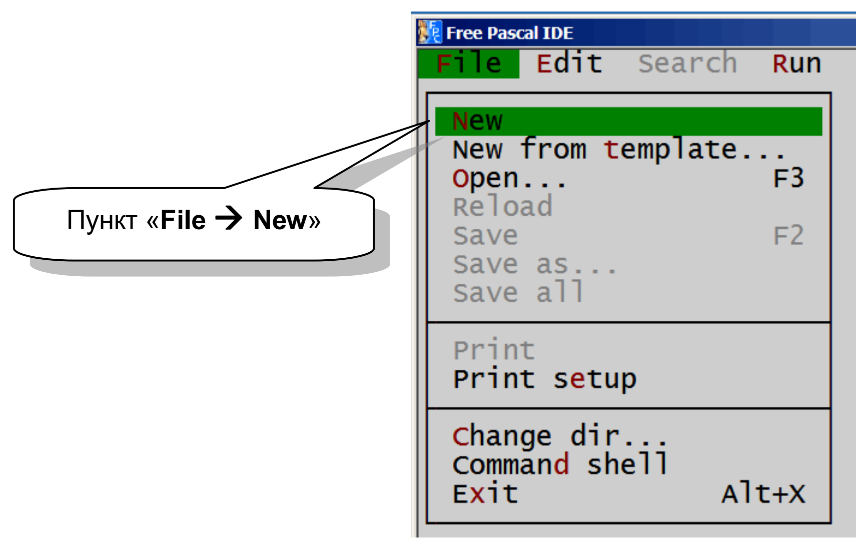


Рис. 13 – Пункт меню для создания нового файла

Сохраните пока ещё пустой файл в своей рабочей папке, у меня это папка «C:\User\Pascal». При сохранении файлу надо придумать подходящее имя. Здесь ваша фантазия ограничена лишь требованиями к именам файлов. Пока вы учитесь, придерживайтесь правил, принятых в MS-DOS: имя файла должно содержать не более восьми символов, не считая расширения имени PAS. В имени используйте только латинские буквы, цифры и знак подчеркивания (пробелы и русские буквы я запрещаю!).

Из этих «кирпичиков» можно составить миллионы имен, — и запутаться в них! Но мы избежим хаоса, применив некоторую систему. Пусть в имени файла

содержится номер главы, где была создана программа. Тогда по имени файла вы найдете надлежащую главу, а по номеру главы — файл.

Итак, имя файла начнем с латинской буквы «P» (от слова «Pascal»), далее последуют две цифры с номером главы и одна цифра — с порядковым номером программы в этой главе. Элементы имени разделим знаками подчеркивания, и тогда для 1-й программы 5-й главы файл получит имя «P_05_1.PAS».

Сохраним его под этим именем. Нажмите клавишу *F2*, — на экране появится диалоговое окно (рис. 14). В верхней строке напечатайте имя файла, а расширение PAS можете не печатать, — оно будет добавлено автоматически. После нажатия клавиши *Enter* или кнопки *OK* файл будет сохранен в рабочей папке, и в заголовке окна появится его новое имя «P_05_1.PAS».

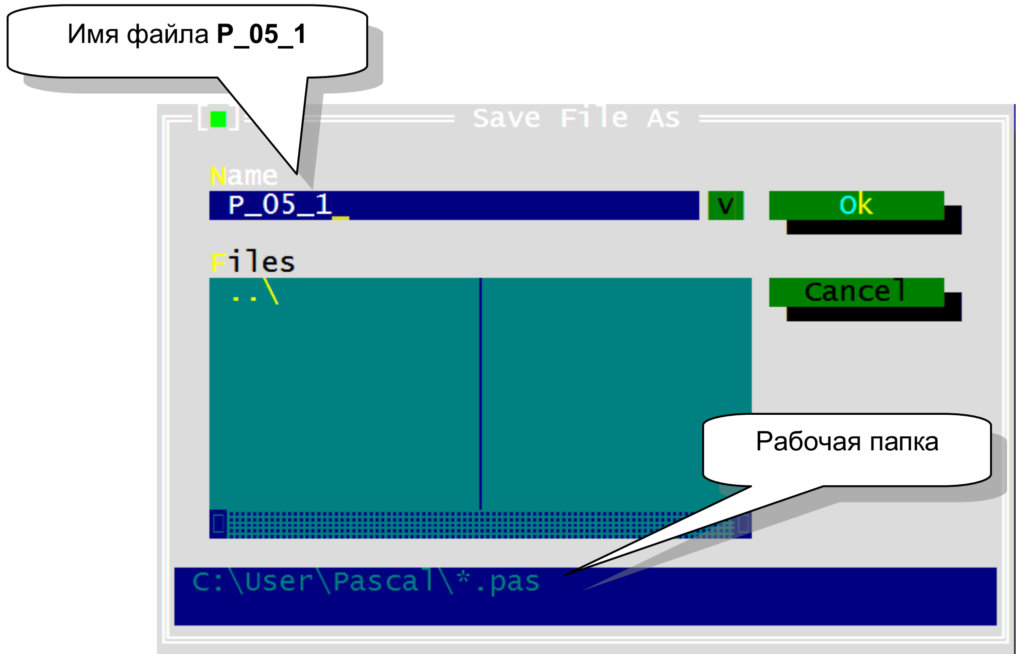


Рис. 14 – Диалог сохранения файла

Наполнение файла

Теперь обратимся к содержимому файла, ведь он пока чист, как белый снег. Нацарапайте на этом снегу парочку английских слов, что показаны ниже.

```
begin

end.
```

Возможно, вам известен их перевод: **BEGIN** — «начало», а **END** — «конец». Зачем они тут? При переводе программы с «человеческого» языка на язык процессора компилятор должен видеть границы программы. Слова **BEGIN** и **END** для того и предназначены, их называют ключевыми. Паскаль содержит десятки

ключевых слов, они перечислены мною в приложении Г. Ключевые слова служат поводырями для компилятора, помогая ему разбираться в программе. Эти слова запрещено использовать по иному назначению!

Итак, слова **BEGIN** и **END** указывают компилятору начало и конец программы. Пару **BEGIN-END** применяют и в иных случаях, как скобки в математике. То есть, после слова **BEGIN** где-то далее в программе обязательно следует слово **END.** Но слово **END** используют и для завершения некоторых других конструкций языка, о которых вы узнаете позже.

Ключевые слова можно печатать и маленькими (строчными) и большими (заглавными) буквами, например: **Begin**, **BEGIN**, **begiN**, — это дело вкуса. То же относится к другим «волшебным» словам языка, о которых вы узнаете позже. Важно помнить, что в этих словах разрешены только латинские буквы. Будьте внимательны: некоторые латинские буквы по начертанию совпадают с русскими («А», «Е», «О»), но для компилятора эти буквы разные, и он обязательно заметит подмену!

Теперь взгляните на точку после слова **END**, — она отмечает конец программы. Без нее нельзя, иначе компилятор попытается читать текст после слова **END**, и, не найдя ничего, сообщит об ошибке.

Итак, напечатав эти две строки с точкой в конце, нажмите ещё раз клавишу *F2* для сохранения файла. Поскольку ранее мы уже дали имя файлу, IDE сохранит его под этим именем, не докучая лишними вопросами.

Примечание. В начале программы иногда пишут необязательное ключевое слово **PROGRAM**, после которого указывают имя программы. Это имя должно совпадать с именем файла без расширения, например:

```
program P_05_1;

begin

end.
```

В наших примерах я не буду вставлять это необязательное ключевое слово.

Теперь поздравьте себя, — вы написали первую программу! И пусть она ещё ни на что не годна, зато синтаксически правильна, — так полагает компилятор. А это важно, — ведь теперь можно создать исполняемый файл, надо лишь откомпилировать этот текст! Раз так, дадим слово компилятору.

Компиляция

А где тут компилятор? Куда спрятался? Не ищите, для создания исполняемого EXE-файла просто нажмите клавишу *F9.* Если две строчки программы были напечатаны верно, появится сообщение об успешной компиляции (рис. 15).

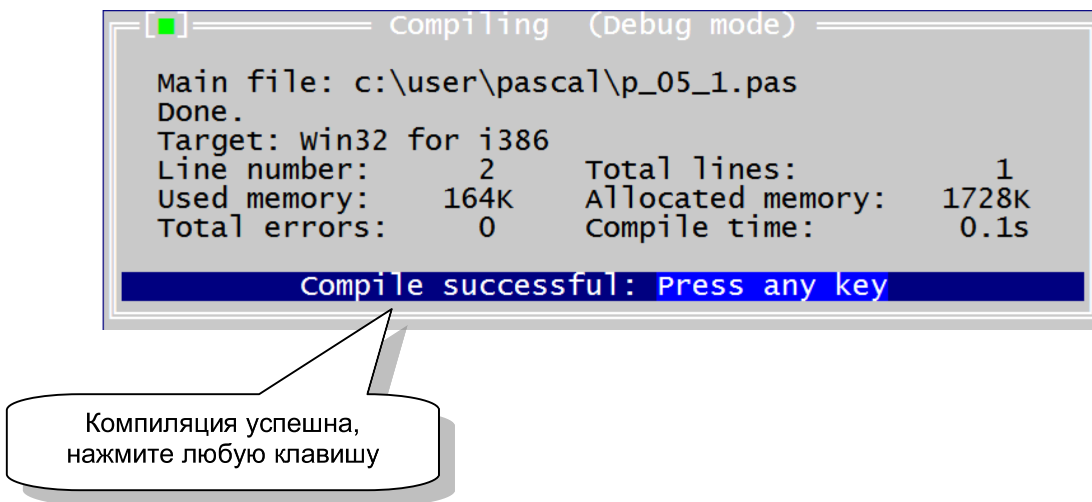


Рис. 15 – Сообщение об успешной компиляции

Закройте окно нажатием любой клавиши, или щелчком по иконке в верхнем левом углу. Заглянув теперь в свою рабочую папку, вы обнаружите там наряду с файлом «P_05_1.PAS» ещё и файл «P_05_1.EXE».

Запустите на выполнение новорожденный EXE-файл и поделитесь своим наблюдением. Зоркий охотник заметит лишь мелькнувшее консольное окно. Чем это объяснить? Запуская консольную программу (именно такую мы сейчас сотворили), Windows создаёт для нее консольное окно, а по завершении программы закрывает его. Поскольку наша программа пока ещё пуста и завершается, ничегошеньки не сделав, консольное окно вмиг исчезает.

Процедура вывода (печати)

Вернемся в IDE и продолжим наполнять нашу программу с тем, чтобы напечатать на экране приветствие. Вообще-то печатать можно только на бумаге, а на экране — высвечивать. Но, со времен пишущих машинок — консолей — слово «печатать» настолько укоренилось, что вывод на экран или файл я иногда буду называть «печатью».

Итак, для вывода приветствия добавим между ключевыми словами **BEGIN** и **END** ещё одну строчку, вот она.

```
Writeln('Привет!')
```

Разберем строку «по косточкам». Прежде всего, мы видим слово **Writeln**. Это сокращение из двух слов: Write — «записывать», и Line — «линия, строка», что вместе значит «написать строку». Слово **Writeln** в Паскале не ключевое — это имя процедуры. В отличие от ключевых слов — поводырей для компилятора, — процедуры определяют выполняемые программой действия. На пути к вершинам Паскаля мы встретим немало ключевых слов и процедур, и всякий раз я буду объяснять, где что.

Вернемся к процедуре **Writeln**, которая дает указание напечатать что-либо на экране. Но, что именно? Ответ находится внутри круглых скобок, где содержатся параметры процедуры. В этих скобках мы видим слово «Привет!», заключенное в апострофы (иногда их называют одинарными кавычками). В Паскале строку, заключенную в апострофы, называют строковой константой. Вот несколько примеров строковых констант:

```
'Привет, Мартышка!'

'--- Free Pascal ---'

'Я понял, что такое строковая константа!'
```

Как видите, любой текст обращается в строковую константу, если заключить его в апострофы. Внутри такой константы компилятор не различает ни ключевых слов, ни процедур, а воспринимает строку «как есть». Длина применяемых нами строк будет ограничена 255 знаками, включая пробелы. А вот примеры «незаконных», ошибочных строковых констант:

```
Нет первого апострофа'

'Нет последнего апострофа

'Апостроф ' внутри строки'

Совсем без апострофов
```

А когда надо вставить апостроф внутрь строки? Тогда ставят два апострофа подряд, например:

```
'Один апостроф '' внутри строки'
```

И, хотя в середине строки поставлены два апострофа, компилятор учтет только один из них, — такая вот хитрость!

Теперь, с оператором печати, наша программа выглядит следующим образом:

```
begin

Writeln('Привет!')

end.
```

Постарайтесь ввести её без ошибок, ведь вы пока не умеете бороться с ними. Готово? Тогда сохраните файл нажатием *F2* и скомпилируйте нажатием *F9*. Если всё нормально, появится знакомое окно успешной компиляции. В противном случае найдите ошибку, исправьте её и повторите компиляцию.

Запуск программы

Теперь мы создали новую версию файла «P_05_1.EXE», запустите его, пожалуйста. Что увидели? Опять мелькнувшее окно? Причина всё та же: операционная система быстренько закрыла консольное окно, поскольку программа завершилась сразу после вывода сообщения, — вы просто не успели его разглядеть! Скоро мы найдем способ притормозить программу, но ждать нам недосуг, — не терпится посмотреть результат! И я покажу, как его увидеть.

Готовую программу мы запустим, не покидая IDE, — нажатием сочетания клавиш *Ctrl+F9*. Нажали? И что, опять ничего?! Спокойно, сейчас разберемся. Дело в том, что IDE закрывает собою всю площадь консольного окна, пряча то, что вывела в это окно наша программа. Чтобы увидеть результат, надо временно убрать IDE нажатием комбинации клавиш *Alt+F5*. Сделайте так, и тогда вам явится долгожданная картинка (рис. 16).

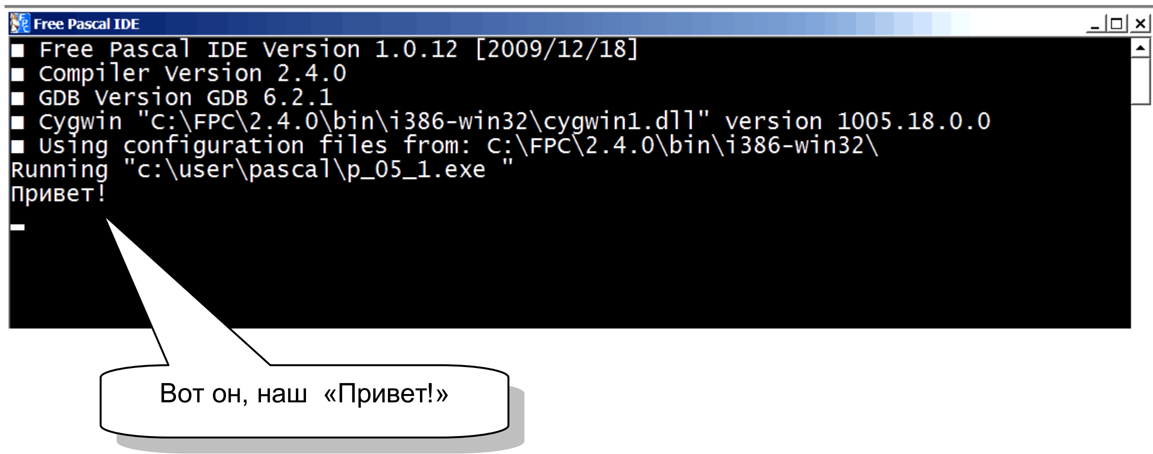


Рис. 16 – Вид консольного окна после скрытия IDE

Первые строки содержат служебное сообщение о запуске IDE Free Pascal, — не смотрите туда. Нам важна последняя строка, где мы видим долгожданный «Привет!». Полюбовавшись на него, вернитесь в IDE, для чего нажмите любую клавишу. Хотите повторить удовольствие? Так запустите программу ещё пару раз (*Ctrl+F9*) и полюбуйтесь на результат (*Alt+F5*).

Итоги

- Создание программы начинается с подготовки текстового файла.
- Программа на Паскале содержит, по меньшей мере, одну пару ключевых слов **Begin – End** с точкой в конце. Между этими ключевыми словами помещают операторы, выполняющие нужные действия.
- Вывод информации на экран выполняется процедурой **Writeln** с параметрами внутри круглых скобок; таким параметром может быть строковая константа.

- Строковая константа – это последовательность символов, заключенная в апострофы. Наши строки будут содержать не более 255 символов.
- Для создания исполняемого EXE-файла вызывают компилятор, это делается нажатием клавиши *F9*. Если программа не содержит синтаксических ошибок, компилятор создаст исполняемый файл и сообщит об успешной компиляции, а иначе доложит об ошибке.
- Запустить исполняемый файл можно непосредственно в IDE. Для этого следует нажать сочетание клавиш *Ctrl+F9*.
- Для просмотра выводимых на экран результатов (временного скрытия IDE) нажимают комбинацию клавиш *Alt+F5*, а для восстановления IDE – любую клавишу.

А слабо?

Начиная с этой главы и далее, я буду предлагать вам каверзные вопросы и задачки. Некоторые из них решаются в уме, другие — на компьютере. Если вы справитесь с большинством задач, спите спокойно — ваша совесть чиста, а голова не пуста. Вот вам первые задания.

А) Найдите ошибки в следующей программе.

```
begn

Writeln(ПрЫветик!)

end
```

Сначала проделайте это в уме, а затем на компьютере. Объясните, почему компилятор не нашел ошибки в слове «ПрЫветик». Или слабо?

Б) Будет ли работать следующая программа?

```
begin Writeln('Begin  End.')  end.
```

В) Попытайтесь написать программу, выводящую на экран не одну, а две строки, например:

```
Без труда

Не выловишь калошу из пруда
```

Здесь нужны две процедуры печати, следующие друг за другом. Подсказка: между процедурами требуется специальный разделитель — точка с запятой (;).

Глава 6
Подготовка к следующему штурму

Перед штурмом следующей крепости подтянем «тылы» и укрепимся на завоеванной позиции.

Ещё об исходных файлах

Работая над первой программой, мы создали и сохранили исходный файл с расширением PAS. Если вам потребуется вновь обратиться к нему, достаточно будет нажать клавишу *F3*, и тогда появится окно открытия файла (рис. 17).

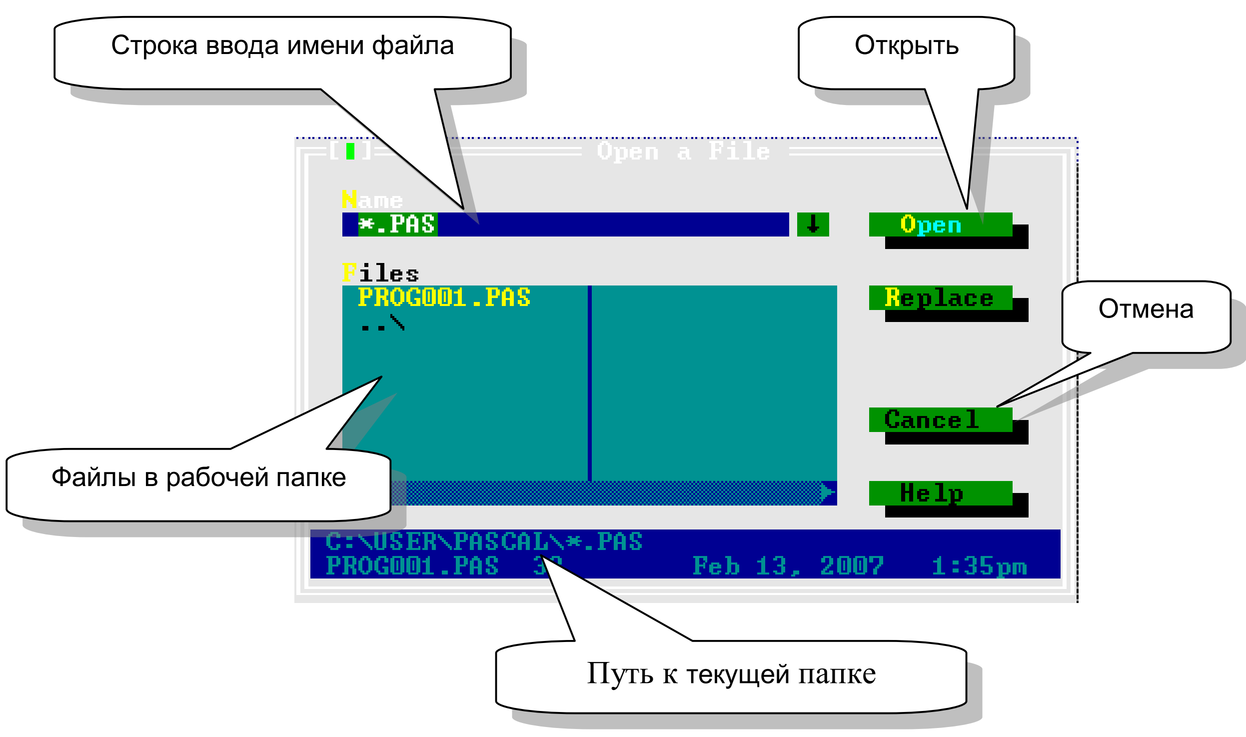


Рис. 17 – Окно открытия файла

В верхней части расположено поле для ввода имени открываемого файла. В центре — список файлов текущей папки (файлов с расширением PAS), а путь к этой папке виден в нижней части окна. Щелкнув мышкой по имени файла, вы переместите его в поле ввода. Теперь достаточно щелкнуть на кнопке *Open*, или нажать клавишу *Enter*, и файл откроется в окне редактора. Так последовательно можно открыть несколько файлов.

Открытие файла — дело нехитрое, но лучше, если в следующий раз вам не придется вспоминать о файлах, с которыми вы работали в предыдущем сеансе. Для этого при выходе из IDE не закрывайте окна, и тогда при повторном запуске IDE автоматически откроет эти файлы.

Управление окном редактора

Тем, кто привык управляться с окнами Windows, обращаться с окнами IDE Free Pascal понравится ничуть не меньше. Взгляните на рис. 18, где представлены средства управления окном.

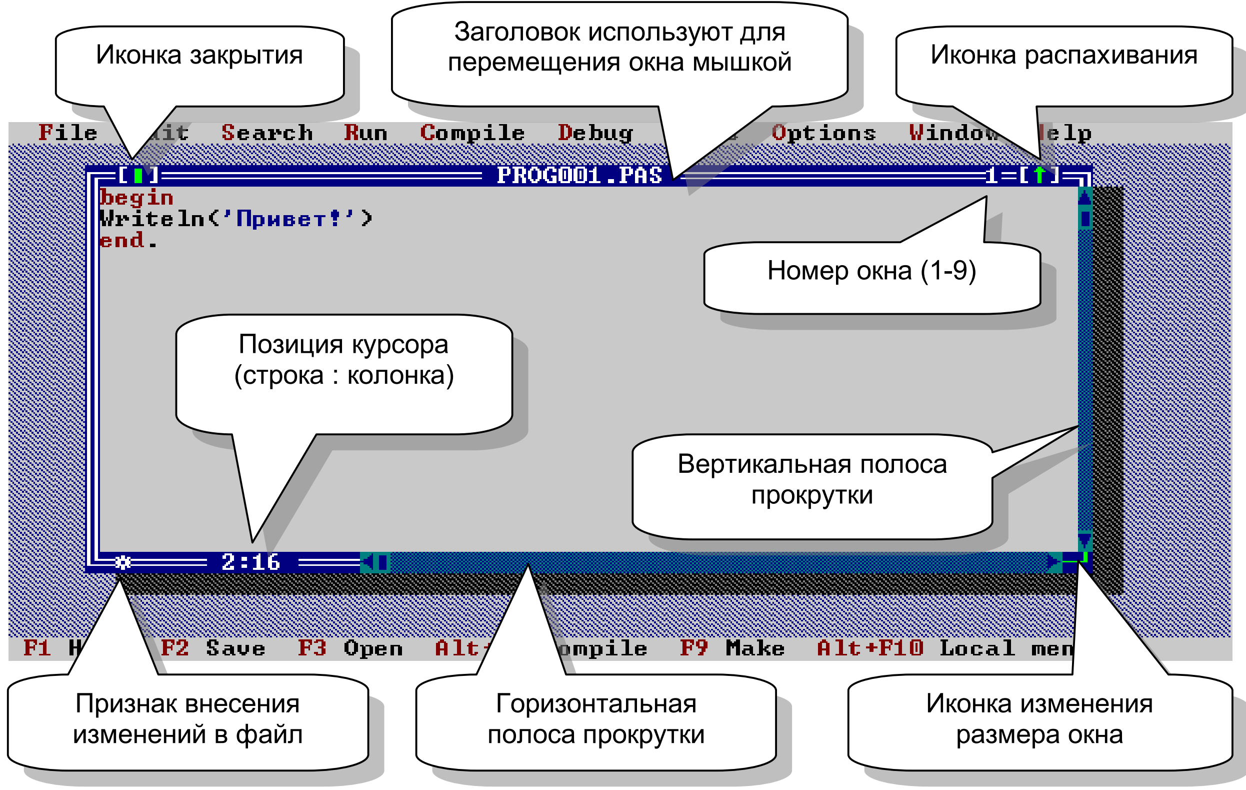


Рис. 18 – Средства управления окном редактора

Иконка в правом нижнем углу изменяет размеры окна. Не верите? Так «схватите» её мышкой и «потаскайте». А «ухватив» мышкой верхнюю часть рамки, вы сможете двигать по экрану всё окно. Иконка [↑] в правом верхнем углу распахивает окно на весь экран или вновь сворачивает его к прежнему размеру.

А что это за звездочка в левом нижнем углу? Она указывает на то, что файл изменялся и не сохранен; звездочка исчезает после сохранения файла клавишей *F2* или командой меню *File → Save*.

В левом верхнем углу видна иконка закрытия окна [■]. Если вы не вносили изменений в файл или уже сохранили изменения клавишей F2, щелчок на этой иконке просто закроет окно. Если же файл изменялся (видна звездочка), то перед закрытием окна IDE запросит подтверждение на сохранение файла (рис. 19).

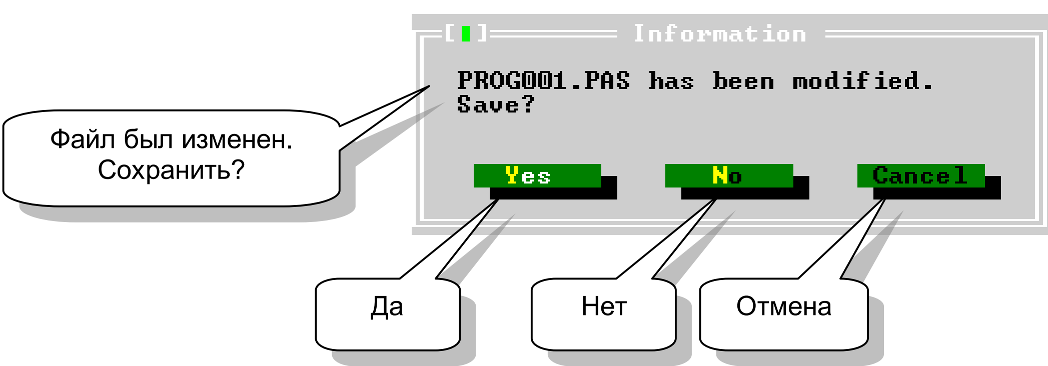


Рис. 19 – Запрос на сохранение файла

Ответ *Yes* приведет к закрытию окна с сохранением последних изменений, ответ *No* — без сохранения. При нажатии кнопки *Cancel* окно не закроется, и вы сможете продолжить редактирование файла.

Повозитесь немного с окном, — это добавит вам уверенности.

Борьба с ошибками

Ошибки, ошибки… — их никому не миновать! Мы тоже не ангелы и будем ошибаться. Но в компьютере всё поправимо. Не страшитесь ошибок, — вы всегда сможете найти и поправить их, и в этом IDE Free Pascal вам поможет.

Ошибки ошибкам рознь. В разговоре и письме мы допускаем ошибки разного рода: грамматические, синтаксические и смысловые. Вот школьная тетрадь, что там нацарапано? «МАлАко»? — ужас! — это грамматическая ошибка, такого слова нет. А если видим: «змея даёт зеленое молоко», — это смысловая ошибка, хоть с грамматикой тут всё в порядке.

Вернемся к Паскалю. Превращая вашу программу в исполняемый файл, компилятор, подобно учителю, читает её слева направо и сверху вниз, зорко изучая вашу писанину. Обнаружив синтаксическую ошибку, он сообщит об этом и остановится. Исполняемый EXE-файл будет создан только при отсутствии синтаксических ошибок. К чему такая строгость? Мы склонны прощать друг другу мелкие огрехи, если понимаем смысл сказанного. Но компьютер не так умен, чтобы додумывать наши человеческие мысли, — вот почему так строг компилятор.

Обратимся к практике. Откройте файл с нашей первой программой и внесите ошибку в первой строке. Например, уберите первую букву в слове **BEGIN**.

```
egin

Writeln('Привет!')

end.
```

Запустите компиляцию этой программы — нажмите *F9*, — и что же? В окне сообщений вы увидите: «BEGIN expected…». Это значит, что компилятор не нашел

обязательного в начале программы ключевого слова **BEGIN**. Компилятор может обнаружить много разных ошибок, вы найдете их перечень в справке по IDE (Appendix C, Compiler messages), а также в приложении Д.

Исправим эту ошибку и сделаем другую: уберем закрывающую кавычку в тестовой константе (после восклицательного знака).

```
begin

Writeln('Привет!)

end.
```

Попытавшись скомпилировать, получим сообщение: «String exceeds line». Это значит, что строковая константа превышает допустимый размер. Стало быть, компилятор не всегда точно определяет место ошибки, и тогда не худо самим «шевельнуть извилинами», — здесь полезна тренировка. Поупражняйтесь в поиске ошибок, внося их в программу сознательно. Запускайте компиляцию, и наблюдайте результат. Так накопите бесценный опыт исправления ошибок, и «ужасные» сообщения уже не запугают вас.

А найдет ли компилятор ошибку внутри апострофов? Как он воспримет слова «прЫвет» и «мАлАко»? Ничего не заметил? Это и понятно, ведь слова внутри апострофов компилятор не проверяет. Не его это дело, — он вообще не знает русского языка! В строковых константах он проверяет, как мы уже убедились, только парность апострофов.

Итоги

- В редакторе IDE можно одновременно открывать несколько исходных файлов с программами.
- При запуске IDE Free Pascal автоматически открывает файлы, открытые в предыдущем сеансе (точнее, окна, не закрытые при выходе из сеанса).
- Элементы управления окном редактора изменяют его размеры, перемещают по экрану, распахивают, сворачивают и закрывают окно.
- Компилятор Pascal проверяет текст программы при каждой компиляции. Обнаружив синтаксическую ошибку, он не создает исполняемый файл, а выводит краткое описание ошибки.

Глава 7
Развиваем успех

Теперь усложним задачу, пусть компьютер обратится к вам вот с таким пышным приветствием:

```
-----------------------
Мой повелитель!
Поздравляю тебя с первой программой!
Твой верный слуга Паскаль
-----------------------
```

Здесь в первой и последней строках для красоты печатается горизонтальный прочерк.

Операторы и разделители

Создадим новый файл и сохраним его под именем «P_07_1.PAS». Напомню, что новый файл создается через пункт меню *File* → *New*, а сохраняется нажатием клавиши *F2.* Покончив с этим, приступим к сочинению программы. Поразмыслив немного, вы наверняка напишите следующие строки.

```
begin
Writeln('-----------------------')
Writeln('Мой повелитель!')
Writeln('Поздравляю тебя с первой программой!')
Writeln('Твой верный слуга Паскаль')
Writeln('-----------------------')
end.
```

Ход вашей мысли ясен: уж если компилятор читает программу слева направо и сверху вниз, то и компьютер будет выполнять её в том же порядке. Вы угадали, так оно и есть! Ну что ж, пробуем скомпилировать свое детище, жмем *F9* и что? Опять видим сообщение об ошибке (рис. 20)!

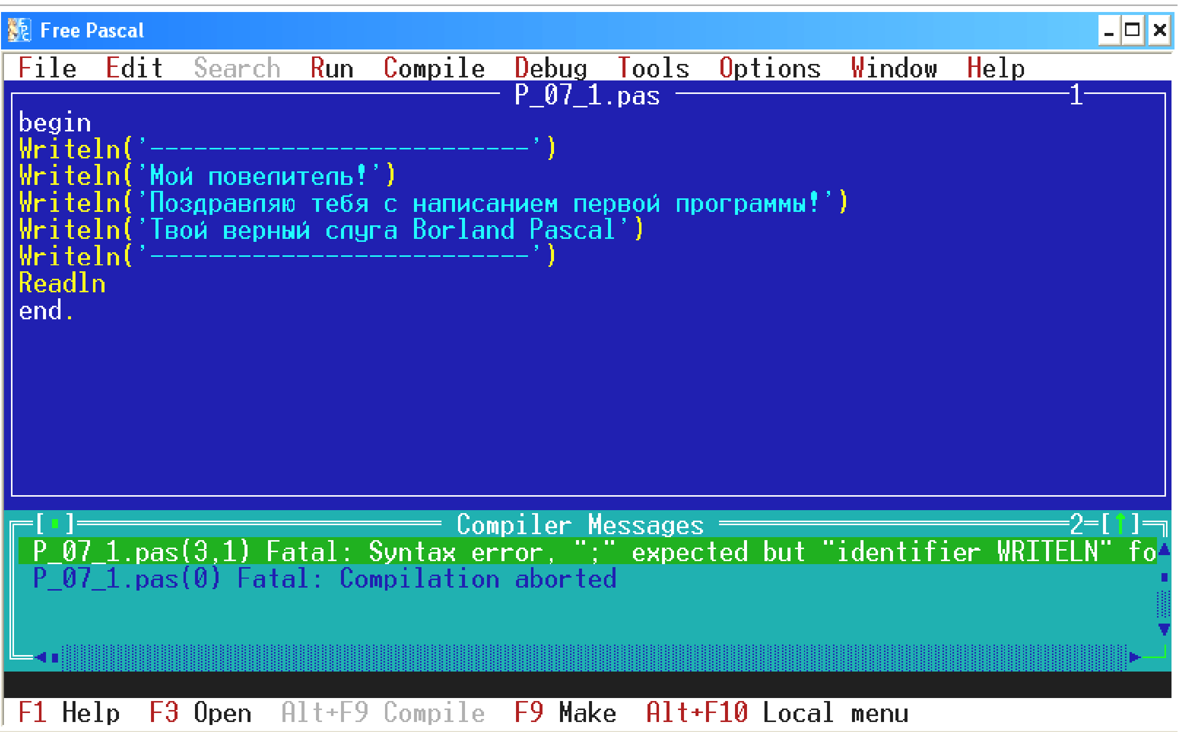


Рис. 20 – Сообщение о синтаксической ошибке

В чем дело? Компилятор утверждает, что где-то пропущена точка с запятой. В скобках указано место ошибки: третья строка, первый символ. Значит, точку с запятой надо ставить здесь? И зачем она нужна?

Познакомьтесь с важным понятием языка — **оператором**. Оператор — это наименьший смысловой «кусочек» программы. Он заключает в себе либо небольшое действие — шаг программы, либо описание каких-то данных. В Паскале есть много разных операторов, процедура печати — один из них. В целом программа — это последовательность операторов и ключевых слов. Читая программу, компилятор должен уяснить, где кончается один оператор и начинается следующий. И здесь он нуждается в вашей помощи! Ему нужна подсказка — разделитель операторов, которым служит точка с запятой (;).

Вернемся к нашей программе. Хотя процедуры печати расположены в разных строках, этого мало, чтобы компилятор воспринял их как отдельные операторы. Порой длинный оператор не помещается в одной строке и пишется на нескольких. А бывает наоборот — в одной строке пишут несколько операторов. Вот почему нужен разделитель.

Итак, между операторами надо поставить точку с запятой, программисты обычно ставят разделитель **после** оператора. Теперь наша программа станет такой.

```
begin
Writeln('------------------------');
Writeln('Мой повелитель!');
Writeln('Поздравляю тебя с первой программой!');
Writeln('Твой верный слуга Паскаль');
Writeln('------------------------')
end.
```

А где разделитель за последним оператором, то есть перед словом **END**? Здесь он не нужен, поскольку **END** — не оператор, а ключевое слово. Но, если вы поставите лишнюю точку с запятой или даже несколько подряд, в этом не будет ошибки. Теперь можно запустить программу нажатием *Ctrl+F9* и полюбоваться на результат её работы, нажав *Alt+F5*.

Программа, стой!

Как хотите, а мне надоело всякий раз нажимать сочетание *Alt+F5* для того лишь, чтобы увидеть результат. Пора избавиться от этого неудобства.

Познакомьтесь с новой процедурой, она называется **ReadLn**. Это слово, как и слово **Writeln**, тоже состоит из двух: Read — «чтение», Line — «линия, строка», что значит «чтение строки». Действует процедура **ReadLn** очень просто: дойдя до её исполнения, компьютер остановится в ожидании, пока вы не нажмете клавишу *Enter*. Вот и всё. И пока он ждет, вы спокойно разглядываете консольное окно. Ясно? Тогда подскажите, где поместить эту процедуру? Ну, очевидно же — самым последним оператором! В результате получим новый вариант программы.

```
begin
Writeln('------------------------');
Writeln('Мой повелитель!');
Writeln('Поздравляю тебя с написанием первой программы!');
Writeln('Твой верный слуга Паскаль');
Writeln('------------------------');
Readln
end.
```

Про точку с запятой не забыли? Отлично! Запускаем программу и убеждаемся, что Паскаль нас снова не подвел (не забудьте нажать *Enter*!).

Алгоритмы

Взгляните на программу ещё разок: печатая строки, компьютер выполняет отдельные действия — шаги программы. Такую последовательность шагов называют алгоритмом. Вам следует привыкнуть к этому слову, ведь алгоритм —

основное понятие в программировании. Вот слегка упрощенное определение алгоритма, запишите: «Алгоритм — это точное предписание исполнителю совершить определенную последовательность действий для достижения поставленной цели за конечное число шагов». Под исполнителем мы понимаем компьютер.

В этом определении угадывается что-то знакомое, не так ли? Ещё бы! То и дело мы получаем указания: сделай то, да сделай это. За что ни возьмись, надо выполнять некий алгоритм. Так, например, одеваясь на улицу, вы соображаете, что и за чем следует напялить на себя: сначала белье, затем рубашку, брюки, носки и ботинки. Даже при ходьбе выполняем простейший алгоритм: левой, правой, левой, правой...

Разбивая сложное действие на ряд простых шагов, вы создаете алгоритм. Алгоритм нашей программы состоит из шагов, выполняемых друг за другом, последовательно. Линейная последовательность — это одна из трех базовых управляющих структур, на которых строится вся гигантски сложная архитектура современных программ (о двух других базовых управляющих структурах я расскажу позднее).

Блок-схемы

Как видите, с алгоритмами связан любой из нас, а не только программисты. Создание напичканных компьютерами сложных систем — заводов, электростанций и тому подобного — требует согласованных усилий специалистов разных профессий. Они объясняют программистам требования к создаваемым системам. Иными словами, эти специалисты заказывают алгоритмы. Увы, не все они владеют программированием. Как быть?

Кто-то догадался изображать алгоритмы графическими схемами, картинками. Этот прием оказался наглядным и понятным даже людям, далеким от программирования. Блок-схемы — так называют эти картинки — стали средством общения между специалистами разных профессий с одной стороны, и программистами с другой.

Впрочем, программисты и между собой общаются посредством блок-схем. Эти схемы помогают обнаружить ошибки в программах. В чем отличие блок-схемы программы от её текста? Текст показывает то, что фактически делает программа, а блок-схема — то, что она должна делать. Сравнивая одно с другим, можно найти ошибки в программном воплощении алгоритма.

Перед вами блок-схемы трех созданных нами программ (рис. 21).

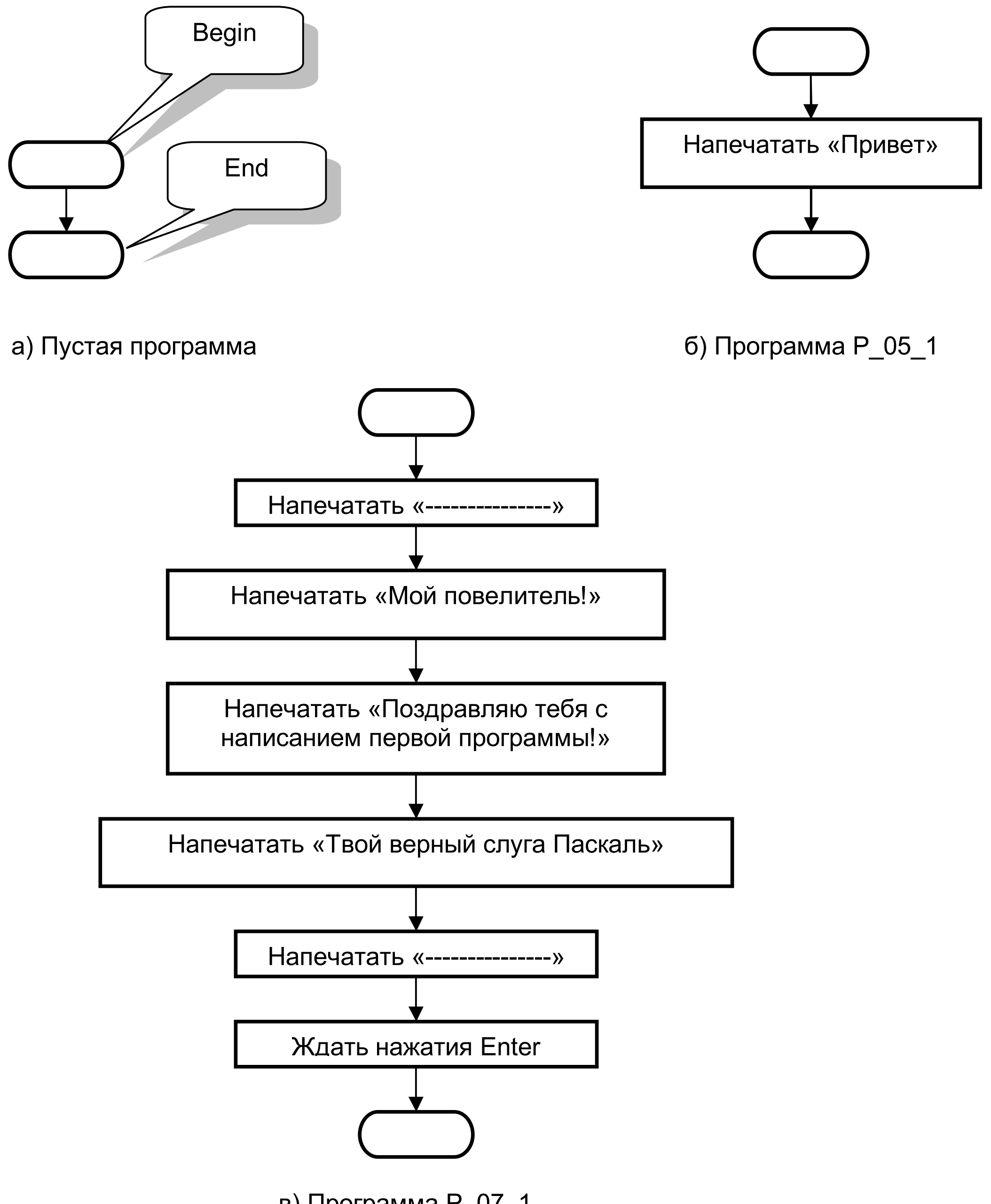


Рис. 21 – Блок-схемы программ

Скругленные прямоугольники означают начало и конец алгоритма, — они соответствуют ключевым словам **BEGIN** и **END**. Исполняемые операторы — это прямоугольники с пояснениями внутри, а стрелки показывают порядок выполнения операторов. Всё просто! Скоро мы изучим другие базовые управляющие структуры, и вы увидите их блок-схемы.

Итоги

- Наименьшая смысловая часть программы называется оператором. Процедура печати **Writeln** и процедура ввода **Readln** – это операторы.

- Программа – это последовательность ключевых слов и операторов.
- Для разделения операторов используют точку с запятой.
- Точное предписание порядка выполняемых действий называется алгоритмом.
- Линейная последовательность – это один из трех базовых алгоритмов.
- Алгоритм может быть представлен словесным описанием, рисунком (блок-схемой), или текстом программы.

А слабо?

А) В нашей программе остался маленький изъян. Со временем вы забудете о том, что для завершения программы надо нажать клавишу *Enter*. Пусть программа сама напомнит об этом, печатая после приветствия напоминание:

```
Для завершения программы нажмите Enter
```

Внесите это изменение в программу. Или слабо?

Б) Измените программу так, чтобы в каждой строке разместилось по два оператора. Откомпилируйте и проверьте программу в действии. Изменилось ли что-то в её поведении?

В) Нарисуйте две блок-схемы, поясняющие, как вы обычно проводите свой будний и выходной день.

Глава 8
Постоянные и переменные

Знаком ли вам Эдсон Арантес ду Насименту? Неужто не слышали о великом Пеле? Ведь оба имени принадлежат одному человеку! В Бразилии полно отменных футболистов, и у всех — пышные имена. Но от футбольных комментаторов вы их не услышите. Бразильцы — а все они фанаты — дали своим любимцам короткие клички. Так на весь мир прославились Пеле, Зико, Ривалдо…

Константы

Придумка бразильских фанатов напомнила решение сходной проблемы программистами. В разных частях программы нередко попадаются одни и те же данные — строчки текста или числа. Взять хотя бы предыдущую программу, где в начале и в конце приветствия печатаются две горизонтальные черты. Если в этих операторах задать линии разной длины, то красота слегка пострадает. Пустяки? Конечно. Но в иных случаях ошибочка обойдется втридорога, например, в программе управления ракетой.

Предположим, что в расчете полета ракеты учитывается масса её полезной нагрузки, и это число разбросано по всем частям программы. Вы должны указать его везде одинаково, без ошибки, иначе ракета улетит «за бугор». А если переделка ракеты повлечет изменение этого числа? Тогда доведется тщательно «прочесать» программу в поисках всех исправляемых операторов.

Проблема очевидна, но Паскаль даёт средство её решения — это символические константы. «Константа» в переводе на русский означает «постоянный», «неизменный». Константа подобна кличке бразильского футболиста: любому элементу данных — числу или строке — вы можете назначить удобное имя, а затем подставлять это имя вместо самих данных. Покажем это на примере нашей программы.

Прежде, чем применить символическую константу, её надо объявить, то есть дать ей имя и значение. Для объявления используют ключевое слово **CONST**, за которым следует нечто, похожее на простую формулу.

```
const Имя_константы = Значение_константы;
```

Слева от знака равенства указывают имя константы, а справа — её значение. Предположим, что длинный прочерк я обозначил словом **Line** — «линия». Тогда объявление константы для линии будет таким.

```
    const Line = '--------------------------';
```

Обратите внимание, что объявление константы — это оператор, и после него следует точка с запятой! Теперь в любом месте программы я могу напечатать прочерк, пользуясь именем этой константы.

```
    Writeln(Line);
```

Параметром процедуры печати **Writeln** здесь по-прежнему является всё та же строковая константа, только теперь она обозначена через свое имя **Line**.

Слово **CONST** открывает секцию объявления констант, внутри которой надо объявить хотя бы одну константу, — секция не терпит пустоты! Вот объявление двух констант, где для наглядности слово **CONST** записано в отдельной строке.

```
Const
    C1 = 'Мой повелитель!';
    Pele = 'Эдсон Арантес ду Насименту';
```

Секцию констант располагают до слова **BEGIN** перед началом выполняемых операторов. Следовательно, новый вариант нашей программы будет таким.

```
const
    Line = '--------------------------';
begin
Writeln(Line);
Writeln('Мой повелитель!');
Writeln('Поздравляю тебя с написанием первой программы!');
Writeln('Твой верный слуга Паскаль');
Writeln(Line);
Readln
end.
```

Программа будет работать точь-в-точь, как и раньше. Но теперь мы уверены, что линии будут одинаковыми. А если потребуется изменить линию и составить её из звездочек? Тогда исправим лишь объявление константы:

```
const    Line = '***************************';
```

и после повторной компиляции программа заработает по-новому.

Константы облегчают жизнь программиста и повышают надежность программ. Но, повторяю, после изменения константы вы должны повторно откомпилировать программу!

Идентификаторы

Для констант придумывают подходящие имена. Впрочем, это касается и других объектов программы, о которых вы скоро узнаете. Выдуманные программистом имена называют идентификаторами (IDENTIFIER). Запомните это словцо, оно ещё «намозолит» вам глаза. Изобретатель идентификаторов ограничен следующими рамками.

- В идентификаторах допустимы лишь латинские буквы, знак подчеркивания и цифры.
- Идентификатор начинают либо с буквы, либо с подчеркивания (но только не с цифры!).
- Идентификатор может содержать до 127 символов, (в Borland Pascal учитываются только первые 63 из них).
- Не допускается совпадение идентификатора с ключевым словом.

Русские буквы и знаки препинания в именах запрещены. Заглавные и строчные латинские буквы равнозначны (регистр букв не учитывается), поэтому идентификаторы **`Pascal`** и **`PASCAL`** считаются одинаковыми.

Вот примеры правильных идентификаторов:

```
A, b, C                   - однобуквенные имена
R1, U28, _13_             - имена с цифрами и подчеркиванием
Cosmos, ABBA              - однословные имена
NextStep, Next_Step       - имена, составленные из двух слов
```

А это ошибочные имена:

```
7Up   - начинается с цифры
End   - совпадает с ключевым словом
```

Изобретая имена, мы будем придерживаться некоторой системы с тем, чтобы меньше путаться в своих придумках. Так, имена констант условимся начинать с латинской буквы «C» (например, **`CLine`**).

Переменные

Согласитесь, наш последний шедевр — программа P_07_1 — пока не слишком умна, при каждом запуске она тупо твердит одно и то же. Сотворим нечто поумней: пусть наша следующая программа сначала спросит имя пользователя, а затем обратится к нему по этому имени. На экране это будет выглядеть так:

```
Как тебя зовут?
Антон
Здравствуй, Антон
Нажми Enter
```

Здесь подчеркнутое мною слово «Антон» во второй строке ввёл пользователь. Такие программы называют **диалоговыми**.

Ясно, что неизвестное имя собеседника в программу заранее не вставишь. Константа тут бесполезна, — ведь она вбивается в программу заранее и не меняется после компиляции. Если данные **вводятся** пользователем в ходе **выполнения** программы, им нужно нечто иное, — этим данным надо отвести место для **хранения** их в памяти. И тогда мы сможем как-то работать с этими сохраненными данными (например, печатать).

Где хранят предметы? В ящиках, карманах, кошельках. Для хранения данных **в памяти** используют **переменные** (**VARIABLE**). Переменная — это своего рода «карман» с именем, данным ему программистом. В ходе работы программа может «укладывать» в переменную данные, и затем обращаться с ними по своему усмотрению. Этот «карман» действует по принципу: что положил, то и взял. Иначе говоря, в переменной хранится то, что было положено последним. Но, в отличие от кошелька, единожды положенное в переменную можно извлекать многократно, — этот «карман» никогда не опустеет!

Прежде, чем пользоваться переменной, её, как и константу, надо **объявить**. Для этого служит **секция** объявления переменных, которую открывают ключевым словом **VAR** (сокращение от **VARIABLE**), секцию помещают до исполняемых операторов — перед словом **BEGIN**. Внутри секции объявляют одну или несколько переменных. Каждое такое объявление содержит два элемента: **имя переменной** и её **тип**, разделяемые двоеточием:

```
var   Имя_переменной : Тип_переменной;
```

Ну, с именем всё ясно — это обычный идентификатор, который вы изобретаете сами. А что такое **тип**, и в чем его смысл? В этой обширной теме мы со временем разберемся основательно, а сейчас затронем лишь слегка.

Укладывая предметы, вы учитываете их размеры, вес и назначение. Пылесосу удобно в своей коробке, а монете — в кошельке. «Каждый сверчок — знай свой шесток». Встретив в программе объявление переменной, компилятор отводит ей место в оперативной памяти с тем, чтобы хранимые данные поместились там. То есть, кроит «карман» подходящего размера. Это первое.

А ещё компилятору надо знать набор допустимых действий с теми данными, что «лежат» в переменной: можно ли их складывать и умножать? Или это строка текста, предназначенная для вывода на экран? Ответ на эти вопросы заключен в

типе переменной. По нему компилятор определяет и размер переменной, и набор допустимых операций с нею.

Паскаль содержит ряд встроенных типов данных, со временем вы познакомитесь с ними, но сейчас нам позарез нужен только один из них. Это тип **STRING**, что в переводе значит «строка» — это ключевое слово языка. Переменная этого типа может хранить в себе строчку какого-нибудь текста.

Объявим переменную для хранения в ней имени пользователя. Как назовем её? Да так и назовем — **Name**, что переводится как «имя». Итак, объявление переменной **Name** строкового типа **STRING** выглядит так:

```
var   Name : string;
```

Напомню, что имя и тип переменной разделяются двоеточием, а завершается оператор точкой с запятой.

Ввод и вывод данных

Теперь, когда мы объявили переменную, попробуем ввести в неё данные, а затем вывести данные на экран.

Ввод данных в переменную выполняется знакомой вам процедурой **Readln**. Мы уже пользовались ею, чтобы заставить компьютер ждать нажатия Enter. Но процедура придумана в основном не для этого, а для ввода разнообразных данных, в том числе строк. С этой целью процедуре передают параметры — переменные, куда вводятся данные. В нашем случае оператор ввода имени будет таким:

```
      Readln(Name);
```

Выполняя этот оператор, компьютер тоже остановится в ожидании нажатия Enter. Но символы, которые пользователь напечатает до этого нажатия, попадут в переменную **Name** и сохранятся там. Так в строковую переменную можно ввести слово, и даже целое предложение, завершив ввод нажатием Enter.

А как напечатать содержимое переменной? Справится ли с этим процедура **Writeln**? Без сомнения! Ведь нечто подобное мы уже проделывали с константой. Вот оператор печати для этого случая:

```
      Writeln(Name);
```

Всё хорошо, да вот незадача! Этот оператор напечатает имя в отдельной строке, а нам хочется объединить его со словом «Привет» в одной строчке. Как это сделать? Очень просто! Ведь в процедуре печати можно указать несколько параметров, разделяя их запятыми, и тогда все они напечатаются в одной строке. В нашем случае через запятую укажем два параметра:

```
Writeln('Здравствуй, ', Name);
```

Здесь первый параметр — строковая константа «Здравствуй, » (с пробелом в конце), а второй — переменная **Name**.

Теперь всё готово для рождения новой программы. Создайте пустой файл с именем «P_08_1.PAS», а затем введите в него плод наших размышлений.

```
var Name : string;
begin
Writeln('Как тебя зовут?');
Readln(Name);
Writeln('Здравствуй, ', Name);
Writeln('Нажми Enter'); Readln;
end.
```

Откомпилируйте программу и проверьте, работает ли она.

Итоги

- Константы полезны для именования неизменяемых данных. Они облегчают работу и повышают надежность программ. Но константы не могут изменяться в ходе выполнения программы.
- Переменные предназначены для хранения в оперативной памяти компьютера изменяемых данных. Переменные могут изменяться в ходе выполнения программы.
- Каждая переменная относится к некоторому типу данных, который определяет и объем занимаемой ею памяти и правила действия с переменной.
- Ввод данных в переменные выполняется оператором **Readln**, а вывод – оператором **Writeln**.
- Процедура **Writeln** может напечатать в одной строке несколько параметров – констант и переменных, разделенных запятыми.
- Имена констант и переменных – это идентификаторы. Программист составляет их по своему усмотрению из латинских букв, цифр и знака подчеркивания.

А слабо?

А) Что напечатает следующая программа, если ваша любимая команда — «Спартак»?

```
const
      Champ = ' - чемпион!';
var
      Team : string;
begin
Writeln('Ваша любимая команда?');
Readln(Team);
Writeln(Team, Champ);
Readln
end.
```

Б) Найдите (и исправьте, если можно) ошибки в следующих программах.

```
begin
const Pele = 'Эдсон Арантес ду Насименту';
Writeln('Лучший футболист мира - ', Pele);
Readln
end.
```

```
begin
Writeln('Как тебя зовут?');
var Name : string;
Readln(Name);
Writeln('Здравствуй, ', Name);
Writeln('Нажми Enter'); Readln;
end.
```

```
const Pele = 'Эдсон Арантес ду Насименту';
begin
Writeln('Лучший футболист мира');
Readln(Pele);
Writeln(Pele);
Readln
end.
```

Глава 9
Переменные: продолжение знакомства

Теперь, после знакомства с переменными, вы умеете объявлять их, вводить в переменные данные и печатать. Отныне мы не расстанемся с ними.

Представьтесь, пожалуйста!

Наша следующая программа P_09_1 спросит у пользователя имя и фамилию, после чего обратится к нему уважительно, как следует. Вот пример такой «беседы» (подчеркнутые слова печатал пользователь).

```
Фамилия?
Скотинин
Имя?
Тарас
Здравствуй, Тарас Скотинин!
Нажми Enter
```

Примечание. Тарас Скотинин — персонаж комедии Д.И. Фонвизина «Недоросль».

Очевидно, что для хранения имени и фамилии одной переменной мало, нужны две. Памятуя о том, что секция объявления переменных допускает несколько операторов, объявим там парочку переменных.

```
var   N : string;
      S : string;
```

Здесь переменные **N** и **S** названы мною по первым буквам слов Name (имя) и Surname (фамилия). Объявить несколько переменных одного типа можно и в одной строке, перечислив их через запятую.

```
var   N, S : string;
```

Тут две переменные объявлены одним оператором, — этот способ ничуть не хуже.

Далее, после ввода данных, надо напечатать в одной строке несколько параметров: приветствие, имя, фамилию, и восклицательный знак в конце, чтобы обратиться к Тарасу Скотинину так:

```
      Здравствуй, Тарас Скотинин!
```

Достаточно ли здесь одного оператора печати? Конечно! Вот он:

```
Writeln('Здравствуй, ', N, ' ', S, '!');
```

Тут мы втиснули в процедуру **Writeln** аж пять параметров! Обратите внимание: в конце добавлен восклицательный знак, а между именем и фамилией печатается пробел, иначе эти слова слипнутся на экране.

После всех пояснений следующая программа должна быть вполне ясной.

```
var   N, S : string;
begin
Writeln('Фамилия?'); Readln(S);
Writeln('Имя?'); Readln(N);
Writeln('Здравствуй, ', N, ' ', S,'!');
Writeln('Нажми Enter'); Readln;
end.
```

Обязательно скомпилируйте её и проверьте в действии.

Из пустого в порожнее

Итак, нам удалось скроить уже два «кармана» для хранения данных. Действительно, переменные сродни карманам, здесь можно и хранить данные, и копировать из одного «кармана» в другой. Для копирования данных в Паскале применяют оператор присваивания, вот примеры копирования данных.

```
A := 'Привет, Мартышка!';      ← копирование строковой константы
B := A;                        ← копирование из переменной A в переменную B
```

Пара символов «**:=**» — «двоеточие» и «равно» — означают операцию присваивания. Слева от знака операции указывают переменную, в которую будут помещены данные, а справа можно указать переменную или константу. Что, по вашему мнению, напечатает следующая программа?

```
var A, B : string;
begin
A:= 'Первая строка';
B:= 'Вторая строка';
Writeln(A);       Writeln(B);
B:= A;
Writeln(B);       Readln
end.
```

Очевидно, что на экране появятся следующие строки.

```
Первая строка
Вторая строка
Первая строка
```

Первые два оператора заносят в переменные **A** и **B** две строковые константы, которые затем печатаются. Третий оператор присваивания **B:=A** скопирует в переменную **B** значение переменной **A**, где уже содержится «Первая строка», — она и будет напечатана последней.

Но, к чему здесь было копировать данные из одной переменной в другую? Сейчас это не имело смысла, согласен. Но последнее слово ещё не сказано!

Сцепление строк

Спросите у любого: для чего нужны компьютеры? Для вычислений, для чего ж ещё? — ответят некоторые. Другие скажут, что для обработки данных. В самом деле, обработка данных — нечто более общее, чем вычисление. Не пора ли и нам приступить к обработке данных? Познакомимся с простейшей операцией обработки строк, которую называют сцеплением или конкатенацией.

Не пугайтесь этого заумного слова, сцепление строк — простейшее дело! Руками это делается так: берете несколько полос бумаги и пишите что-либо, — это ваши строки, — а затем склеиваете полоски. Это и есть конкатенация строк.

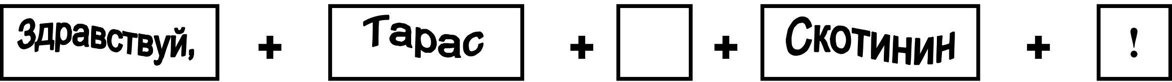


Рис. 22 – «Склеивание» отдельных строк оператором сцепления «+»

На рис. 22 представлено строковое выражение. Знаки «+» здесь обозначают операцию сцепления строк, — точно так же она обозначается и в Паскале. Показанный ниже оператор присваивания занесет в переменную **R** строку, «склеенную» из пяти других строк (здесь **N** и **S** — это переменные, содержащие имя и фамилию человека).

```
R:= 'Здравствуй, ' + N + ' ' + S + '!';
```

Стало быть, справа от операции присваивания « **:=** » может быть не только константа или переменная, но и строковое выражение.

Испытайте теперь второй вариант приветствующей программы с тремя строковыми переменными.

```
var   N, S, R : string;
begin
Writeln('Фамилия?'); Readln(S);
Writeln('Имя?'); Readln(N);
R := 'Здравствуй, ' + N + ' ' + S +'!';
Writeln(R);
Writeln('Нажми Enter'); Readln;
end.
```

Инициализация переменных

Если найдете силы, испытайте и эту программку (в ней есть ошибка!).

```
var   S : string;
begin
Writeln(S);
S:= 'Спартак';
Writeln(S);
S:= S + ' - чемпион!';
Writeln(S);
Writeln('Нажми Enter'); Readln;
end.
```

Здесь переменная **S** будет напечатана трижды. Но что, по вашему мнению, выведет первый оператор **Writeln(S)**? Ни за что не угадаете! Этого даже я не знаю. Всё потому, что при старте программы содержимое всех её переменных не определено, — в этих «карманчиках» может валяться что угодно. Обычно там остаются следы от деятельности предыдущих программ — так называемый мусор. Не пытайтесь напечатать такие переменные или извлечь из них данные, — порой это может вызвать даже аварию программы.

Запомните: прежде, чем взять из «карманчика», туда следует что-либо положить! Надо, как говорят программисты, инициализировать переменную. Это можно сделать двояко: либо вводом данных процедурой **Readln**, либо оператором присваивания.

В последующих операторах этого примера переменная **S** инициализируется, и здесь результат вывода на экран очевиден. А в операторе

```
S:= S + ' - чемпион!';
```

предыдущее значение переменной **S** взято для формирования её нового значения. Теперь там окажется строка «Спартак минус чемпион!». Не обижайтесь, спартаковцы, — пошутил. Обязательно проверьте эту программу!

Типизированные константы

Всем данным в программе свойственен какой-либо тип. Это может быть строка, число или другой тип данных, с которыми вы скоро познакомитесь. То же касается и констант, например:

```
const Pele = 'Эдсон Арантес ду Насименту';      ← это строка (string)
      Number = 12;                              ← это число
```

Здесь тип сам собой определяется тем значением, что дано константе.

Но существует и другая разновидность констант — типизированные константы, которые объявляются с явным указанием типа:

```
const Pele : string = 'Эдсон Арантес ду Насименту';  ← это строка (string)
      Number : integer = 12;                         ← это число (integer)
```

В действительности это тоже переменные, и они могут изменяться в ходе выполнения программы. Но этим переменным изначально присвоены нужные значения, поэтому при запуске программы инициализация их через присваивание уже не требуется.

В Delphi разрешено инициализировать переменные при объявлении:

```
var   Pele : string = 'Эдсон Арантес ду Насименту';
```

Но этот способ не совместим с Borland Pascal, и в данной книге не применяется.

Итоги

- В одном операторе можно объявить несколько переменных одного типа.
- Процедура **`Writeln`** способна напечатать в одной строке несколько параметров. Параметры в списке разделяются запятыми.
- Операция присваивания «**`:=`**» помещает в переменную данные, представленные константой, переменной, или их комбинацией – выражением.
- Конкатенация – это объединение нескольких строк в одну.
- Для инициализации переменной необходимо либо ввести в неё данные процедурой **`Readln`**, либо заполнить оператором присваивания.
- Извлечение данных из переменных, которые не были инициализированы, бессмысленно и нередко вызывает крушение программы.

А слабо?

А) Что напечатает следующая программа?

```
const Pele = ’Эдсон Арантес ду Насименту’;
begin
Writeln(’Pele = ’ + Pele);    Readln;
end.
```

Б) А эта программа что напечатает?

```
var A, B : string;
begin
A:=’123’;   B:=’456’;
Writeln(’A+B= ’ + A + B);     Readln;
end.
```

В) Является ли следующий оператор оператором присваивания?

```
const Pele = ’Эдсон Арантес ду Насименту’;
```

Г) Пусть ваша программа запросит у пользователя его адрес, а именно: город, улицу, номер дома и номер квартиры. А затем напечатает адрес одной строкой в таком виде:

```
Город: ГГГ  Улица: УУУ Дом: ДДД Квартира: ККК
```

Сделайте два варианта программы: один — с печатью нескольких параметров оператором **`Writeln`**, другой — с объединением строк.

Д) Какие из следующих операторов забракует компилятор?

```
const
      Pele = ’Эдсон Арантес ду Насименту’;
      ABBA : string = ’Музыкальный шедевр из Швеции’;
var
      Moscow : string;
begin
      Pele  := ’Лучший футболист мира’;
      ABBA  := ’Распевают частушки’;
      Moscow:= ’Столица олимпиады’;
end.
```

Глава 10
Условный оператор

Согласитесь, наши последние программы слегка поумнели, догнав по интеллекту попугая. Но негоже на лаврах почивать, — научим компьютер принимать осмысленные решения.

Стой! Кто идет?

Вот секретное учреждение, вход в него строго ограничен. А вы — часовой, и пропускаете лишь тех, кто назовет пароль — слово «pascal». Наскучив на посту, вы задумали приспособить вместо себя компьютер. Ваша новая программа P_10_1 должна запросить у пользователя пароль и решить, пропускать ли этого человека.

Вопрос ребром

Что проще должности часового? Пускать или не пускать? Подобные вопросы решаются поминутно: свернуть направо или налево? орел или решка? быть или не быть? От полученного ответа зависят дальнейшие действия.

Обычно мы рассуждаем так: ЕСЛИ некоторое утверждение верно, ТО делаем одно, а ИНАЧЕ делаем другое. Например, ЕСЛИ на улице жарко, ТО наденем футболку, а ИНАЧЕ — свитер. Выделенные мною слова — ключевые в этом рассуждении. Переведя их на английский, получим условный оператор языка Паскаль.

Существуют два варианта условного оператора — полный и неполный. Полный оператор выражается тремя ключевыми словами: **IF** — «если», **THEN** — «то» и **ELSE** — «иначе», и записывается он так.

IF <условие> **THEN** <Оператор_1> **ELSE** <Оператор_2>

Первый оператор выполняется, если условие верно, а второй — если ложно. Стало быть, условный оператор — это сложная конструкция, которая включает в себя другие операторы.

Теперь обратимся к условию, что это такое? Если я скажу, что это логическое выражение, вы ничего не поймете. С логическими выражениями мы скоро разберемся досконально, а здесь ограничимся лишь примером. Воспользуемся простейшим логическим выражением, которое заключается в сравнении двух строк. Предположим, что переменная **S** содержит введенный пользователем пароль, тогда условный оператор проверки пароля будет таким.

```
if S = 'pascal' then Writeln('Проходите!') else Writeln ('Стойте!')
```

Здесь логическое выражение мною подчеркнуто. То же самое можно записать чуть иначе.

```
if 'pascal' = S
    then Writeln('Проходите!')
    else Writeln ('Стойте!')
```

Теперь переменная **S** и константа «pascal» поменялись местами, и это никак не сказалось на условном операторе, поскольку знак равенства в логических выражениях означает сравнение (а не присваивание!).

Части условного оператора **THEN** и **ELSE** называют ветвями (положительной и отрицательной соответственно). Стало быть, и условие, и ветви оператора можно размещать в нескольких строках — это удобно как для чтения, так и для отладки программ.

В главе 7 мы познакомились с графическим изображением алгоритмов. Существуют лишь три базовые управляющие конструкции, из которых вяжется хитроумная паутина современных программ: 1) линейная последовательность, 2) условный переход и 3) цикл. Условный оператор Паскаля — это и есть один из вариантов условного перехода. На блок-схемах его изображают так (рис. 23).

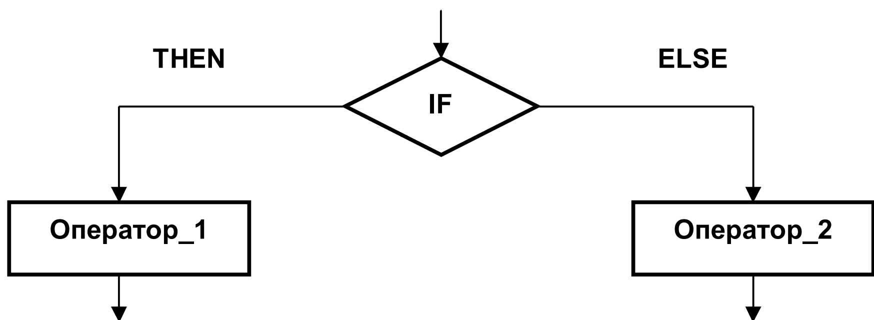


Рис. 23 – Блок схема полного условного оператора

Внутри ромбика или рядом с ним обычно показывают проверяемое условие, а положительную и отрицательную ветви располагают слева и справа от него.

Пост номер один

Вам понятен условный оператор? Тогда обратимся к программе-часовому. Вероятно, вы написали её раньше меня, и нам осталось лишь сверить варианты.

```
var   S : string;
begin
Writeln('Пароль?'); Readln(S);
if S = 'pascal'
      then Writeln('Проходите!')
      else Writeln('Стойте!');
Writeln('Нажмите Enter'); Readln;
end.
```

Почему после оператора **Writeln('Проходите!')** не видно разделителя — точки с запятой? Потому, что внутри условного оператора разделители не ставят! Другое дело — оператор **Writeln('Стойте!')**. Здесь заканчивается условный оператор **IF**, и точка с запятой уместна — она разделяет операторы. Попробуйте нарушить эту запись и узнать мнение компилятора.

Неполный условный оператор

Что за окном? нет ли дождя? ЕСЛИ дождь идет, ТО прихватите зонтик. В этом кратком рассуждении нет отрицательной ветви, поскольку в ней никаких действий не предусмотрено. В таких случаях отрицательную ветвь отбрасывают и получают неполный условный оператор:

IF <условие> **THEN** <Оператор>

Блок-схема такого оператора показана на рис. 24.

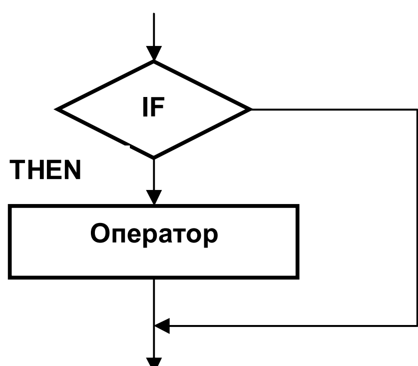


Рис. 24 – Блок-схема неполного условного оператора

Пост номер два

Применим неполный условный оператор ко второй версии электронного часового — программе P_10_2.

```
var   S, R : string;
begin
Writeln('Пароль?'); Readln(S);
R:= 'Стойте!';
if S = 'pascal'
      then R:= 'Проходите!';
Writeln(R);
Writeln('Нажмите Enter'); Readln;
end.
```

Здесь для хранения решения введена переменная **R**, в которую изначально помещается суровое «Стойте!». После успешной проверки пароля значение переменной меняется на благосклонное «Проходите!», а затем решение выводится на экран.

Откомпилируйте и проверьте оба варианта часового. «Поиграйте» с ошибками компиляции. Если компиляция прошла гладко, внесите ошибки сознательно и исследуйте реакцию компилятора.

Теперь вы познакомились с двумя вариантами условного оператора. Ни один серьезный алгоритм не обходится без них. Скоро вам доведется изобретать весьма хитрые алгоритмы и рисовать блок-схемы для них. Значит, надо привыкать к блок-схемам; на рис. 25 представлены схемы наших программ.

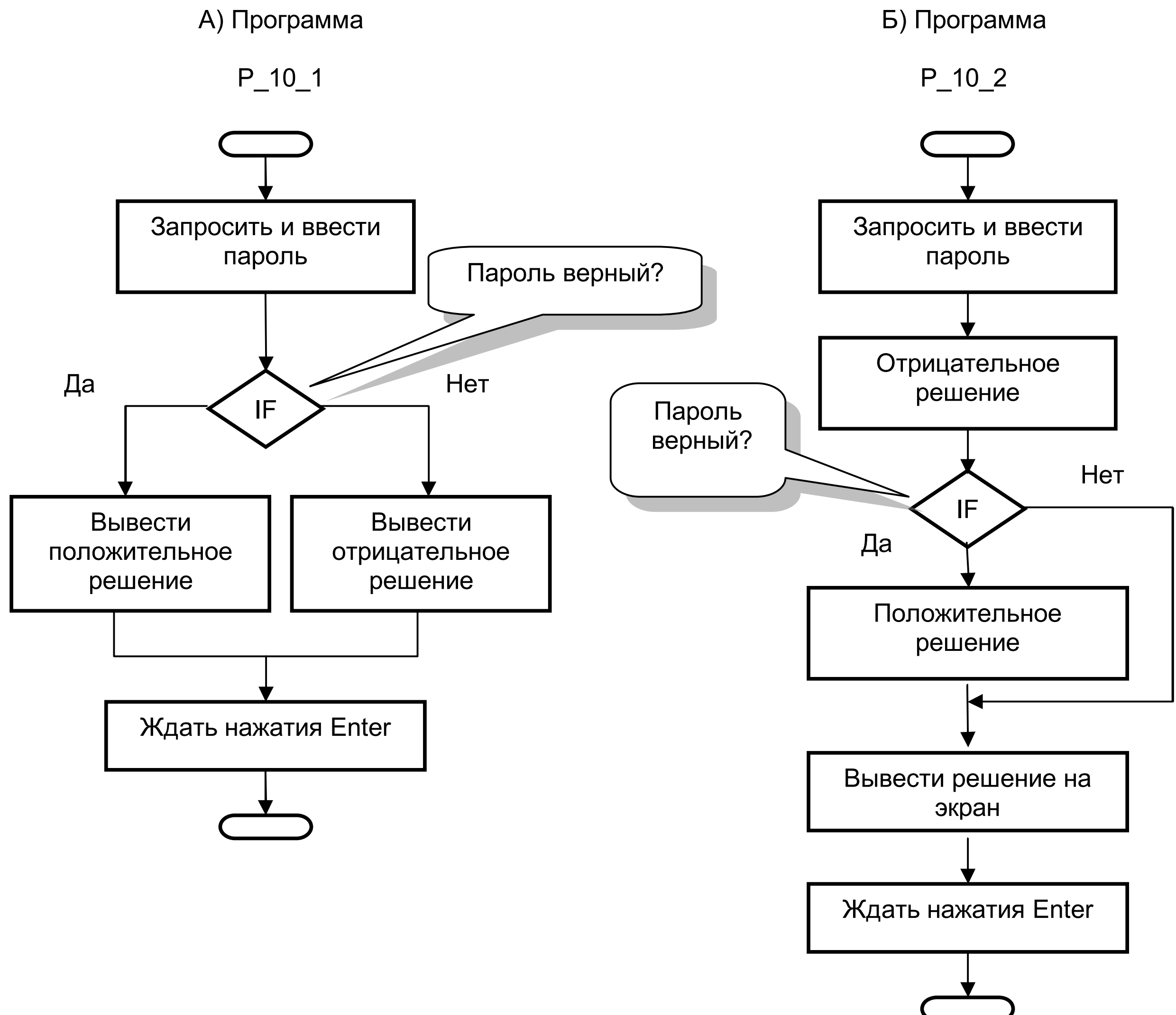


Рис. 25 – Блок-схемы программ с полным и неполным условными операторами

Итоги

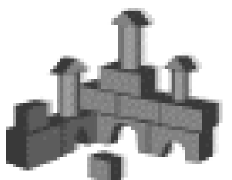
- Условный оператор изменяет порядок действий в зависимости от некоторого условия; оператор может быть полным или неполным.
- Полный условный оператор состоит из условия **IF** и двух ветвей: положительной – **THEN**, и отрицательной – **ELSE**. В каждую из ветвей можно поместить по одному вложенному оператору.
- Неполный условный оператор состоит из условия **IF** и положительной ветви **THEN**.

А слабо?

А) В программах для часового укажите начало и конец условного оператора (то есть, первый и последний его символ, включая вложенные операторы).

Б) Напишите программу, которая спрашивает, идет ли дождь, и на ответ «да» выводит сообщение «А зонта-то у тебя нет!». Воспользуйтесь неполным условным оператором.

Глава 11
Операторный блок



Электронный часовой из 10-й главы пропускает только знающих пароль. Расширим круг его обязанностей. Пусть часовой, приняв верный пароль, отдаст ещё несколько команд, как то: «Распахнуть ворота! Оркестр, музыку!». А для нарушителей команды будут такими: «Тревога! Задержать его!». Разумеется, что команды будут выводиться на экран, причем каждая — в отдельной строке. Усеченная блок-схема программы показана на рис. 26.

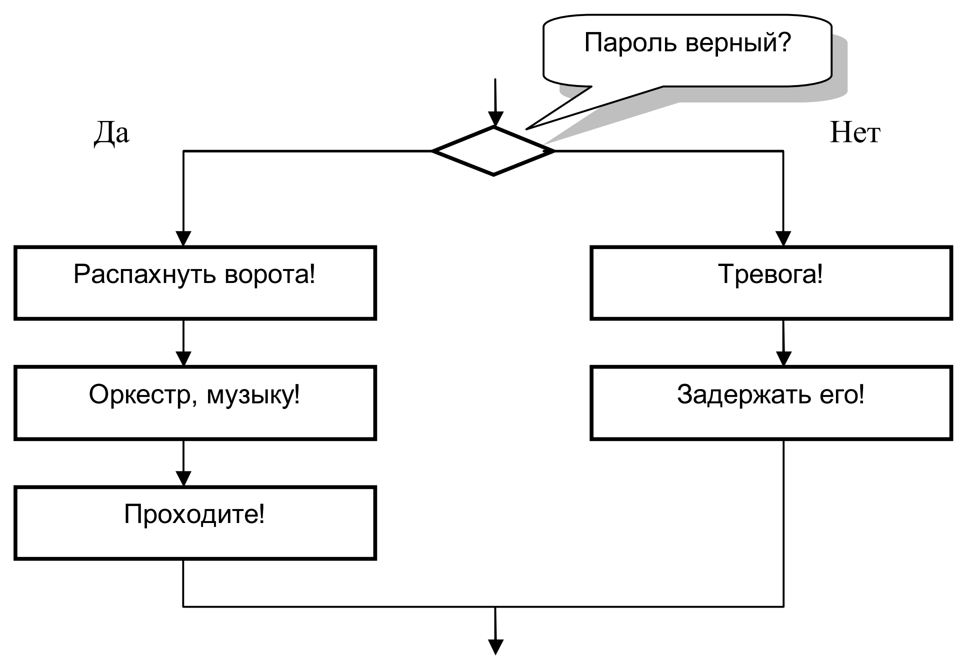


Рис. 26 – Блок схема часового, подающего дополнительные команды

Операторные скобки

На первый взгляд всё ясно, как белый день: если пароль верный, то отдаем команды, что показаны на блок-схеме слева, а иначе те, что справа. Каждую команду выводим отдельным оператором, — так они окажутся в отдельных строках.

```
if S = 'pascal'
      then  Writeln('Распахнуть ворота!');
            Writeln('Оркестр, музыку!');
            Writeln('Проходите!');
      else  Writeln('Тревога!');
            Writeln('Задержать его!');
```

Добавьте в программу часового всё это и откомпилируйте. Что, не вышло? Тут обнажилась проблема с подряд идущими операторами печати. По правилам языка, они разделяются точками с запятой, не так ли? Но разделители «порежут»

на части и условный оператор **IF-THEN-ELSE**, а это недопустимо! Наткнувшись на показанную выше конструкцию, компилятор заявит вам прямо в глаза о синтаксической ошибке. Ведь в каждой ветви условного оператора допускается лишь по одному вложенному оператору, где выход?

«Вероятно, в Паскале что-то предусмотрено на сей счет» — заподозрите вы. Конечно! Здесь выручит операторный блок, который превращает группу операторов в один, скрывая внутри себя разделители — точки с запятой. Блок организуют знакомой вам парой ключевых слов **BEGIN** и **END**. В нашей программе эти слова надо «втиснуть» в ветви условного оператора так.

```
if S = 'pascal'
      then begin
             Writeln('Распахнуть ворота!');
             Writeln('Оркестр, музыку!');
             Writeln('Проходите!')
           end
      else begin
             Writeln('Тревога!');
             Writeln('Задержать его!')
           end;
```

Как видите, внутри блока **BEGIN-END** разделители ставят как обычно — для разграничения операторов.

А сколько операторов вместится в блок **BEGIN-END**? Да сколько угодно! Блок может быть и пустым, — иногда это оправдано. Предположим, вы ещё точно не решили, что будет внутри ветви: один оператор или больше. Тогда вставьте здесь пустой блок **BEGIN-END**, а затем думайте дальше. Вставка лишних блоков не влияет на программу, но может уберечь от синтаксических и логических ошибок.

И последний вопрос: почему после **END** нет точки? Ведь мы ставим её в конце программы! Да, но окончание программы — это единственный случай, когда после **END** ставится точка.

Красиво жить не запретишь

Вероятно, вы заметили, что ветви **THEN** и **ELSE** условного оператора расположены с отступом вправо. Что это, требование языка? Ничуть. Вы вправе написать программу даже в одну строку, и компилятор «проглотит» её. Но каково будет разбираться в такой программе вам или вашему приятелю?

Отступы в программе сделаны для удобства чтения. Строгих правил по этой части нет; оформление — дело вкуса. Но сложились традиции, следование которым облегчит жизнь и вам, и тем, кто будет читать ваши программы. Главная

идея оформления программ состоит в выделении логических уровней. Что это такое?
В данном примере это ветви **THEN** и **ELSE**, — они должны быть хорошо видны в тексте. Полезно, также, выделять блоки операторов. Для этого можно поместить слова **BEGIN** и **END** друг под другом, а содержимое блока сдвинуть относительно них вправо.

Разбирая примеры, вы со временем научитесь разумно оформлять свои программы, — лучше раз увидеть, чем стократ услышать. Вот, в частности, другой вариант оформления программы P_11_1, где обе ветви условного оператора прекрасно видны, хотя скобки **begin-end** и не расположены друг под другом.

```
var   S : string;
begin
      Writeln('Пароль?'); Readln(S);
      if S = 'pascal'   then begin
            Writeln('Распахнуть ворота!');
            Writeln('Оркестр, музыку!');
            Writeln('Проходите!')
      end   else begin
            Writeln('Тревога!');
            Writeln('Задержать его!')
      end;
      Writeln('Нажмите Enter'); Readln;
end.
```

Комментарии

Раз уж мы коснулись оформления, рассмотрим ещё одно средство Паскаля — комментарии, которые служат для пояснения программ. Комментарий — это произвольный текст, заключенный в фигурные скобки {…}, или в круглые скобки со звездочкой (*…*). Вот примеры комментариев.

```
{ Комментарий в одной строке }
{ Многострочный
 комментарий
}
(* Комментарий в скобках со звездочками *)
```

А как воспринимает их компилятор? Да никак. Найдя начало комментария, компилятор ищет его окончание, а всё, что оказалось внутри ограничителей, «пропускает мимо ушей». Поэтому комментарии не оказывают влияния на программу. Есть только одно исключение, о котором я скажу в своё время,

повествуя о директивах компилятора. Последующие программы я буду сопровождать комментариями.

Программисты нередко используют комментарии как «шапку-невидимку». О чем я? Иногда — при поиске ошибок — требуется временно исключить часть операторов из программы. Вместо того чтобы удалять, а затем печатать их заново, лучше закомментировать эту часть текста. То есть, заключить ненужные операторы в фигурные скобки, превратив в комментарий. Такой кусок программы легко восстановить, удалив фигурные скобки.

Примечание. В современных версиях Паскаля и в других языках применяют ещё и однострочный комментарий, который отделяется двумя косыми черточками только с левой стороны.

```
A:= B;      // Копирование переменой – это однострочный комментарий
```

Итоги

- Операторные скобки **BEGIN-END** объединяют несколько операторов в один операторный блок. Операторный блок воспринимается как один оператор.
- Форматирование программы – это оформление её с помощью логических отступов. Форматирование не влияет на программу, но облегчает её чтение.
- Комментарии предназначены для включения в программу пояснений. Комментарии пропускаются компилятором и не влияют на программу.
- Комментарии удобны для временного исключения частей программы.

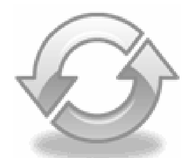
А слабо?

А) Сколько операторов можно поместить в операторном блоке?

Б) Найдите ошибку в этом кусочке программы, проверьте свое решение на компьютере.

```
Writeln('Что дождь? Все ещё идет?'); Readln(S);
if S = 'ага' then
      begin
            Writeln('А зонтик ты так и не купил!');
            Writeln('Сколько раз напоминать?');
      end;
else begin
            Writeln('На этот раз тебе повезло!');
      end;
```

Глава 12
Цикл с проверкой в конце



Подтянем дисциплину

Продолжим воспитывать нашего часового, он ещё нуждается в этом. Проверяя каждого встречного-поперечного, мы принуждены вновь и вновь запускать свою программу. А всё потому, что часовой покидает свой пост без команды, самовольно. Пусть программа проверяет посетителей одного за другим до тех пор, пока мы не скомандуем «отставить!».

Для этого заставим программу «бегать по кругу» так, чтобы она возвращалась к операторам, исполнявшимся ранее. Повторение одних и тех же действий называют **циклом**. Иногда цикл называют **переходом назад**. Блок-схема предстоящей программы показана на рис. 27.

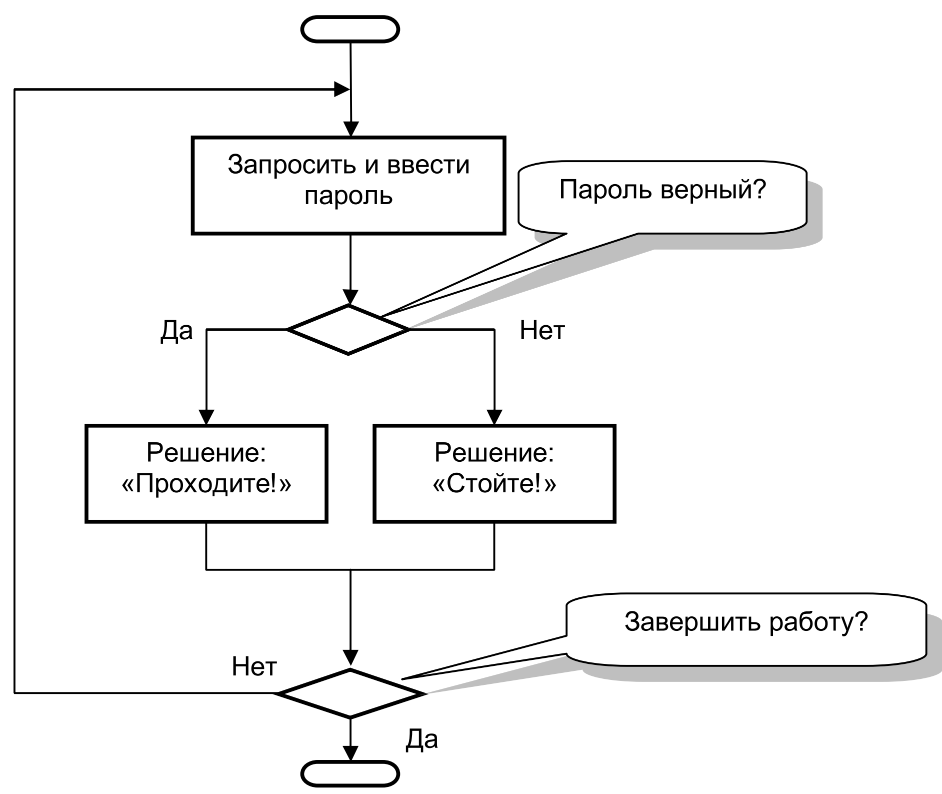


Рис. 27 – Блок-схема циклического часового

Схема содержит два условных перехода, причем второй из них должен, как говорят программисты, передать управление назад, к началу программы. Сейчас нам предстоит, во-первых, найти способ отдать часовому команду покинуть пост и, во-вторых, осуществить переход назад.

Для освобождения часового можно ввести специальную фразу. Например, вместо пароля напечатать фразу «отставить!» или «марш на кухню!». Ещё проще сделать это пустой строкой, которая попадет в переменную **S**, если в ответ на

запрос пароля пользователь, ничего не печатая, нажмет клавишу Enter. Тогда условие завершения программы будет таким.

```
     if S = '' then ...
```

Здесь справа от знака равенства стоят два апострофа, — это пустая строка (между апострофами нет пробела!).

Мы ответили на первый вопрос, но как перейти к началу программы? Не надейтесь на условный оператор, он тут не поможет! Обе его ветви следуют после проверки условия **IF**, поэтому условный оператор передает управление только вперед.

Нанимаем репетитора

Итак, условный оператор тут не помощник, но Паскаль не оставит вас в беде. Для организации циклов в нём предусмотрены три оператора, с одним из них мы ознакомимся немедля. Программистам он известен как цикл с проверкой в конце, и записывается двумя ключевыми словами: **REPEAT** — «повторять» и **UNTIL** — «вплоть до».

Отчасти «репетитор» похож на операторный блок **BEGIN-END**, рассмотренный нами в предыдущей главе. Вам надо повторять выполнение ряда операторов? Тогда поставьте слово **REPEAT** перед первым из них, а проверку условия **UNTIL** — за последним, и получите следующую конструкцию.

```
REPEAT
  <Оператор 1>;
  <Оператор 2>;
  . . .
  <Оператор N>
UNTIL условие
```

По-русски действие оператора можно изъяснить так: ПОВТОРЯТЬ следующие далее операторы, ПОКА условие НЕ соблюдается. На рис. 28 показана блок-схема такой циклической конструкции; здесь операторы 1 и 2 будут исполняться до тех пор, пока НЕ соблюдается условие в конце цикла. При соблюдении условия цикл прекратится, и выполнится оператор 3.

Примечание. Сходство оператора цикла с блоком **BEGIN-END** состоит в том, что **REPEAT-UNTIL** тоже скрывает внутри себя разделители операторов — точки с запятой. Стало быть, он тоже формирует единый блок.

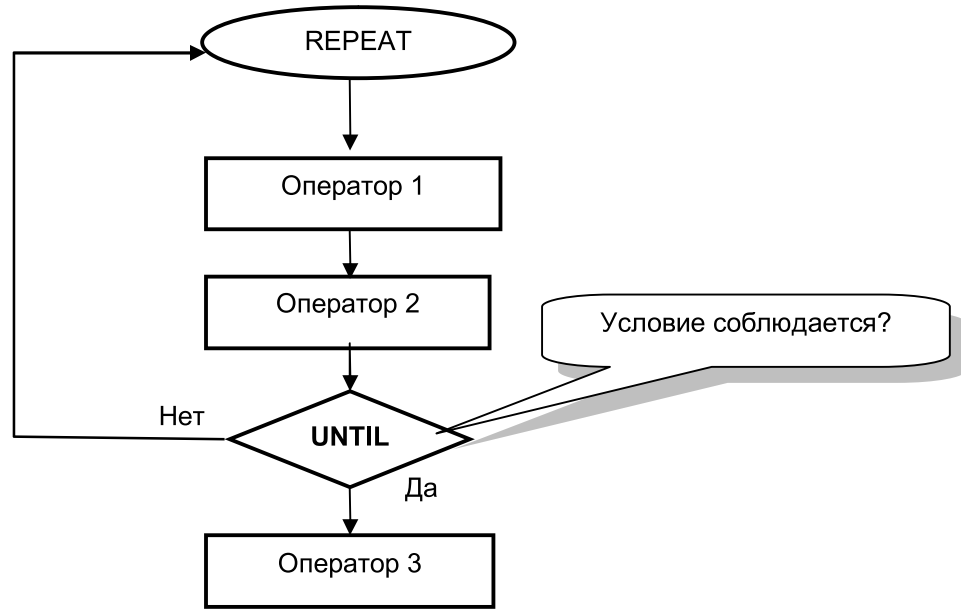


Рис. 28 – Блок-схема оператора цикла с проверкой в конце

Воспользуемся циклом для очередной версии «киберчасового». За основу возьмем простейшую из предыдущих версий — программу P_10_1. Поместив внутрь цикла **REPEAT-UNTIL** все исполняемые там операторы, получим желаемое — программу P_12_1.

```
{ P_12_1 – программа-часовой с циклом }
var   S : string;
begin
      repeat
            Writeln('Пароль?'); Readln(S);
            if S = 'pascal'
                  then  Writeln('Проходите!')
                  else Writeln('Стойте!');
      until S=''; { окончание цикла, если строка S пуста }
end.
```

Проверьте наше новое творение. Обратите внимание на комментарии внутри фигурных скобок, — я буду снабжать ими все последующие программы.

Вежливый часовой

Программа работает? Прекрасно! Но одна шероховатость меня удручает. Покидая пост, часовой почему-то поднимает лишний шум: «Стойте!» — кричит он. Кому он это кричит? своему командиру? Безобразие! Пусть при оставлении поста часовой не проверяет пароль. С этой целью добавим ещё один условный оператор, как показано на рис. 29.

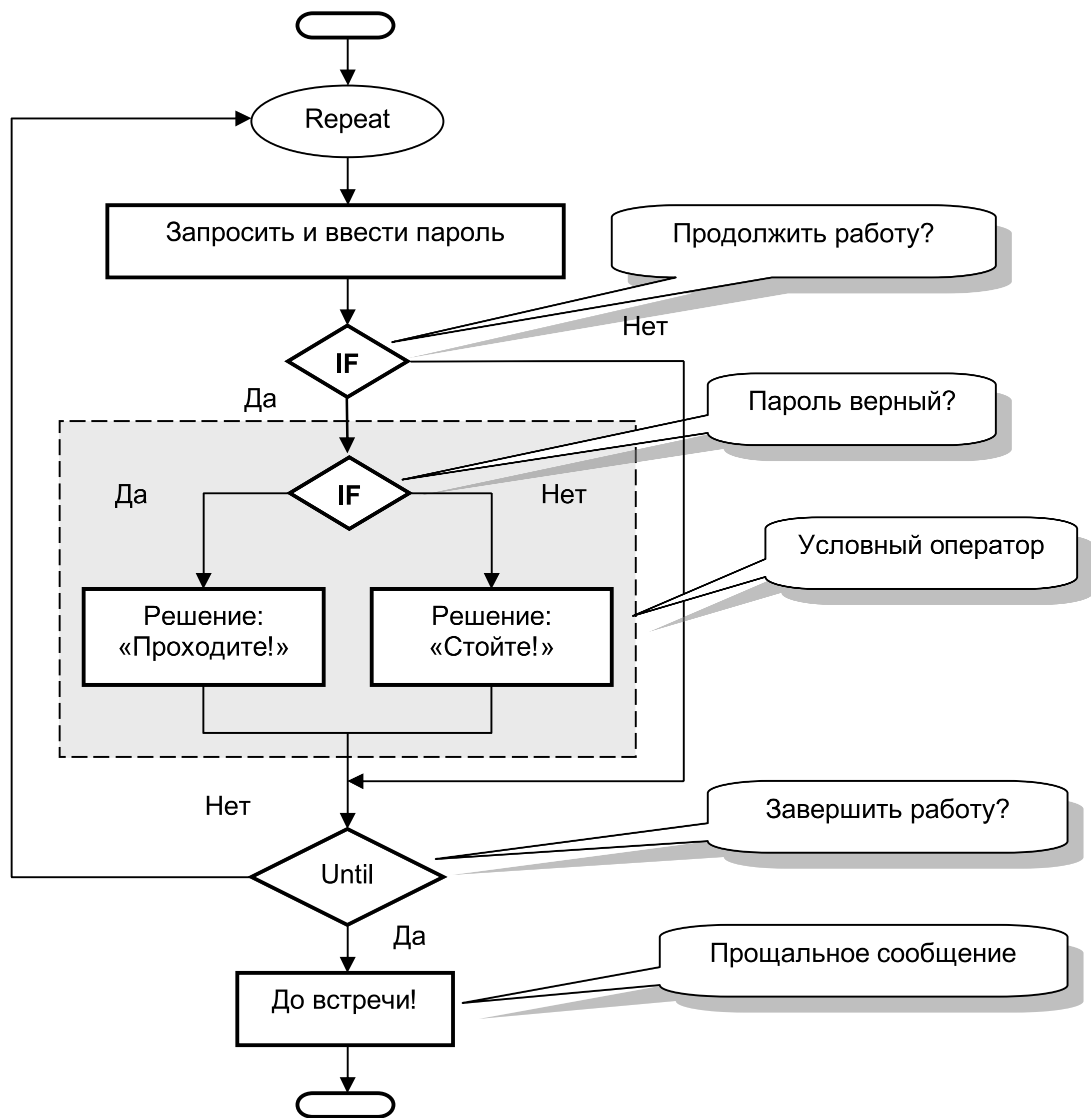


Рис. 29 – Блок-схема часового с корректным завершением

На этой блок-схеме оператор проверки пароля обведен пунктиром; получив команду о завершении работы, программа должна обойти его. Этому служит ещё один условный оператор, проверяющий, не пуста ли строка **S**.

```
if S <> '' then ...
```

Пара знаков «меньше»—«больше» в Паскале означает неравенство. Здесь положительная ветвь **THEN** будет выполнена, если строка **S** не будет пустой. Стало быть, это условие по смыслу противоположно условию **IF S=''**.

А напоследок программа должна вежливо попрощаться, для чего добавим ещё пару операторов печати. Итак, создайте файл «P_12_2.PAS», скопируйте в него предыдущую версию программы и попытайтесь сами внести необходимые изменения, — нет ничего полезней самостоятельной работы! Справившись с задачей, взгляните на мой вариант, он показан ниже. А если не совладаете, тоже посмотрите.

```
{ P_12_2 - вежливый часовой }
var   S : string;
begin
      repeat
            Writeln('Пароль?'); Readln(S);
            { если строка не пуста, проверяем пароль }
            if S<>'' then
                  if S = 'pascal'
                        then  Writeln('Проходите!')
                        else Writeln('Стойте!');
      until S='';
      Writeln('До встречи! Нажмите Enter'); Readln;
end.
```

Я расположил операторы с надлежащими отступами, выделяющими структуру программы. Проверьте, работает ли она?

Досрочный выход из цикла

С какой бы стороны придраться к нашему часовому? Ведь программа делает всё, что положено. Но рассмотрим ещё один её вариант. Дело в том, что условные операторы внутри цикла порой загромождают и запутывают его. Это не относится к нашей теперешней программе, но мы ведь только в начале пути… Ждать ли, пока гром грянет? Или подготовиться к нему заранее? Познакомьтесь с процедурой по имени **BREAK** — «прервать» (боксерам знакомо это слово).

Условие завершения цикла, как вам известно, проверяется в точке **UNTIL**. Но порой это условие удобней проверить где-то в середине цикла, и тогда цикл лучше прервать досрочно, вызвав процедуру **BREAK** следующим образом:

```
      if  <условие_выхода_из_цикла>  then Break;
```

Внимание: вызов процедуры **BREAK** допустим только внутри циклов!

Посмотрите, как изменится блок-схема с оператором **BREAK** (рис. 30), здесь оператор принятия решения я заменил пунктирным прямоугольником.

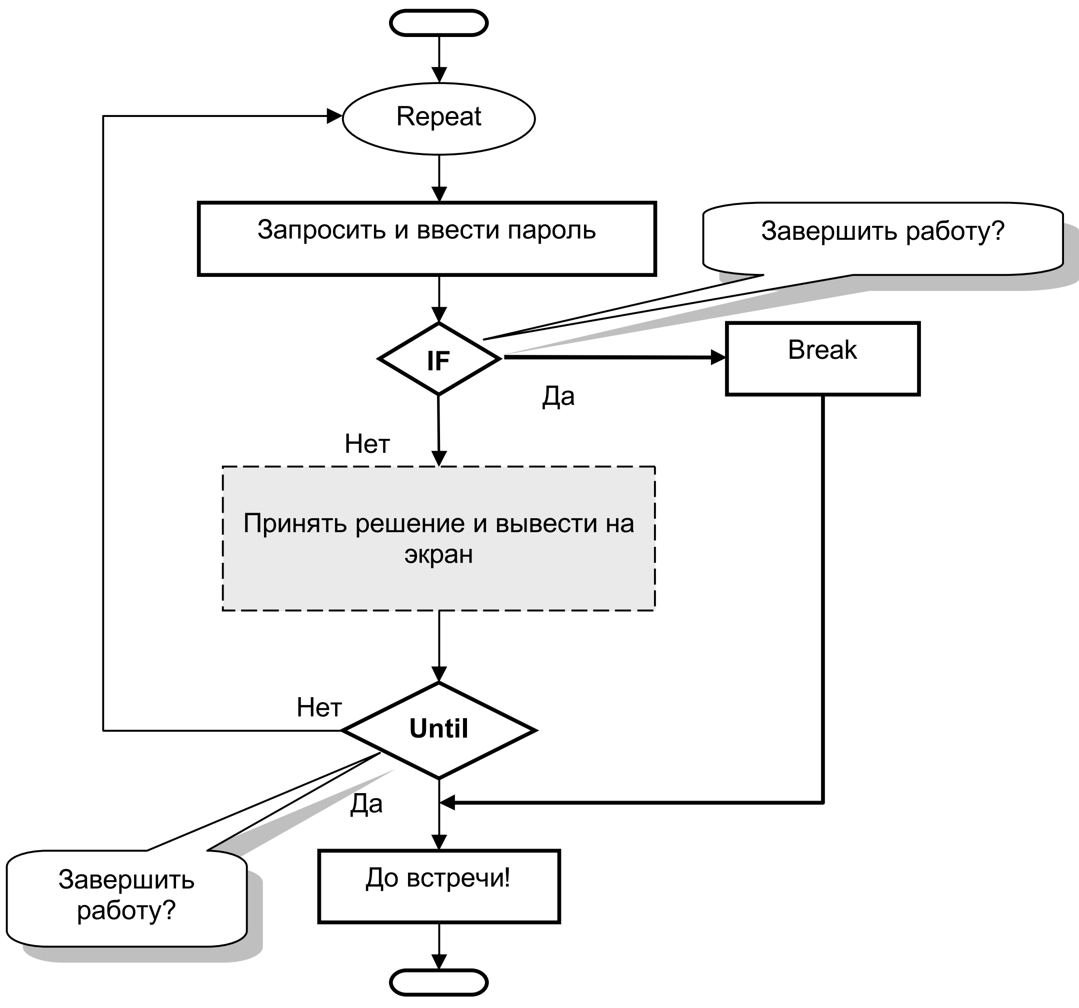


Рис. 30 – Блок-схема циклической программы с оператором Break

Согласно схеме, оператор **BREAK** передаст управление в точку, следующую за **UNTIL**. Применительно к нашей программе условие досрочного выхода из цикла будет таким.

```
      if S='' then Break;
```

Слегка изменив предыдущую версию программы, я получил вариант, показанный ниже.

```
{ P_12_3 – часовой с досрочным выходом из цикла }
var   S : string;
begin
      repeat
            Writeln('Пароль?'); Readln(S);
            { если строка пуста, то выход из цикла }
            if S='' then Break;
            if S = 'pascal'
                  then  Writeln('Проходите!')
                  else  Writeln('Стойте!')
      until S='';
      Writeln('До встречи! Нажмите Enter'); Readln;
end.
```

Досрочный выход из цикла упрощает программу, но пользоваться им надо аккуратно, с умом. Не забывайте, что после **BREAK** программа переходит к оператору, следующему за **UNTIL**.

Итоги

- Оператор цикла **REPEAT-UNTIL** организует многократное повторение операторов, вставленных между этими ключевыми словами.
- Условие выхода из цикла следует за ключевым словом **UNTIL**, цикл повторяется до тех пор, пока условие НЕ соблюдается.
- Оператор **BREAK** выполняет досрочный выход из цикла с обходом условия в **UNTIL**.

А слабо?

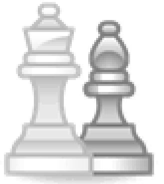
А) Сколько операторов можно вставить между **REPEAT** и **UNTIL**?

Б) Будет ли проверяться условие в **UNTIL** при досрочном выходе из цикла?

В) Возьмите за основу программу P_11_1 и переделайте ее в циклический вариант. Или слабо?

Г) Напишите программу для угадывания слова. Она должна запрашивать от пользователя строки, пока тот не введет слово, предусмотренное в программе.

Глава 13
Правда и кривда



Что приятней, получать подарки или дарить их? Сейчас узнаем: вот вам автомобиль — дарю! Будете в школу на нём гонять. Впрочем, мой подарок не бескорыстен, и взамен я жду вашей помощи.

«Автомобиль — не роскошь, а средство передвижения» — утверждал персонаж книги Ильфа и Петрова. Что бы сказал он сегодня, томясь в унылых пробках? Теперь и вы за рулём, значит, дорожные пробки — наша общая напасть. Так будем бороть её вместе! Ведь всё для этого есть, — в каком веке то живем! Что ни автомобиль — то бортовой компьютер, а космос ломится от спутников! Создадим программу для бортового компьютера автомобиля. Компьютер будет принимать сигналы от спутников системы ГЛОНА́СС (это ГЛОба́льная НАвигацио́нная Спу́тниковая Систе́ма) и сообщать о дорожных пробках. К деталям этого проекта обратимся позже, а начнем, как водится, издалека — из космоса.

Есть ли жизнь на Марсе?

Об этом всё ещё спорят ученые, не находя ответа. «Спросите что попроще, — скажете, — например, мой возраст». Но если бы здесь отвечал компьютер, то вопрос о Марсе он счел бы простым, а вопрос о возрасте — сложным. Почему?

А потому, что о марсианской жизни можно ответить односложно: «да» или «нет». А сообщая возраст, надо указать какое-то число или дату рождения. Ещё сложнее растолковать компьютеру, как выглядит, к примеру, цветок или бабочка, тут не отделаться одним числом, а тем паче ответом «да» или «нет». Такие вопросы требуют сложного ответа, несущего много информации. Стоп! Я упомянул информацию? Вот о ней — об информации — потолкуем подробней.

Информация и её мерило

Компьютер обрабатывает информацию, — это все знают. Только почему, соорудив столько программ, мы всё ещё не знаем, что такое информация? Её трактуют по-разному, одно из определений таково: «информация — это то, что устраняет неопределенность». В самом деле, задавшись неким вопросом, мы испытываем неопределенность, а получив ответ, избавляемся от неё, и на душе становится легче.

Получить ответ — это значит получить информацию. А сколько мы при этом её получаем, чем измерить это количество? Математики догадались, что меньше всего информации заключено в односложном ответе: «да» или «нет». Это количество взято за мерило и названо битом (по-английски «BIT»). Крупные единицы информации содержат много битов: байт — восемь битов, 1 Кбайт (читается «кибибайт») — 1024 байта и так далее, — об этом можно прочитать в школьных учебниках. Стало быть, ответ на вопрос «быть или не быть?» содержит всего один бит информации, — компьютер считает такие вопросы простыми.

Ответы на сложные вопросы (например, как выглядит то, или это) могут содержать миллионы байтов. В этом легко убедиться по размеру файла с какой-нибудь фотографией или фильмом.

Но вернемся к биту. Природная склонность компьютера к «простым» вопросам и односложным ответам объясняется устройством его электронной памяти, состоящей из битовых ячеек — триггеров. Не вдаваясь в технические детали, скажу лишь, что такая ячейка может находиться в одном из двух устойчивых состояний, которые часто обозначают цифрами «0» и «1». С тем же успехом их можно обозначить иначе, например: «да» и «нет». Или так: «истина» и «ложь», «правда» и «кривда», «крестик» и «нолик». Короче говоря, название — дело вкуса, важно лишь то, что триггер хранит один бит информации. Эта особенность компьютеров отразилась во многих языках программирования, в том числе в Паскале.

Булевы переменные

Итак, элемент памяти компьютера — триггер — хранит наименьшую порцию информации — один бит. А в Паскале есть надлежащий тип данных для хранения и обработки битов, он называется булевым (ударение на первом слоге) — по имени английского математика Буля. Другое название этого типа данных — логический. Булевы переменные объявляют так:

```
var  A, B, C : Boolean;
```

Здесь объявлены три переменных булевого типа.

Булевы переменные, подобно триггеру, могут содержать лишь одно из двух значений: **TRUE** — «истина» или **FALSE** — «ложь». Это зарезервированные слова Паскаля, и попытка присвоить логическим переменным другие значения будет пресечена компилятором. Вот примеры правильного обращения с булевыми переменными:

```
A:= true;
B:= false;
C:= B;
```

А вот грубые ошибки:

```
A:= 'true';
B:= 'false';
C:= 'B';
```

Повторяю: **TRUE** и **FALSE** — это зарезервированные слова, а не строковые константы, они пишутся без апострофов.

Ввод и вывод булевых данных

Ввод и вывод — это первое и последнее звено в цепочке обработки данных. Прежде, чем обрабатывать данные, надо освоить их ввод и вывод. Рассмотрим, как это делается с булевыми данными.

С выводом проблем нет, поскольку процедура **Writeln** напечатает их словами «TRUE» и «FALSE». Вот небольшая программа, испытайте её.

```
var   B : Boolean;
begin
      B:= false;        Writeln(B);
      B:= true;         Writeln(B);
end.
```

Вводить булевы данные чуть сложнее, поскольку процедура **Readln**, к сожалению, не умеет этого делать. Как быть? «Нормальные герои всегда идут в обход», — поется в песне. Осуществим хитрый манёвр: для ввода булевых значений воспользуемся переменной другого типа, например, строковой, а затем преобразуем введенную строку в булев тип.

Условимся, что значению **TRUE** будет соответствовать ввод в строковую переменную символа «1», а **FALSE** — любой другой строки. Тогда булево значение в переменную **B** можно ввести следующим манером.

```
var   S : String;
      B : Boolean;
begin
      Writeln('Введите "1" для TRUE и прочее - для FALSE');
      Readln(S);
      if S='1' then B:= true else B:= false;
      Writeln(B);  Readln;
end.
```

Просто? Но можно сделать ещё проще, прибегнув к логическому выражению.

Логические выражения

Данные логического типа можно получать в результате не совсем обычных вычислений. В этих вычислениях порой не увидишь ни чисел, ни арифметических действий, — речь идет о логических выражениях. Например, сравнивая две строки, вы задаетесь вопросом, равны ли они? Ответом может быть либо «да», либо «нет», или, выражаясь на языке Паскаль, **TRUE** или **FALSE**. Следовательно, сравнение строк, которое мы применяли в условных и циклических операторах, — это логическое выражение. А раз так, то результат сравнения можно присвоить

булевой переменной. В приведенном выше примере вместо условного оператора можно записать выражение:

```
      B := S='1';       { равносильно  if S='1' then B:= true else B:= false }
```

Здесь справа от знака присваивания стоит логическое выражение **S='1'**, и в переменную **B** попадет **TRUE**, если **S** будет содержать строку «1» и **FALSE** — в любом другом случае.

Булевы переменные и выражения применяют везде, где требуется проверка условия, например, в условном и циклическом операторах:

```
if B
      then . . . { выполняется, если B=true }
      else . . . { выполняется, если B=false }
```

```
repeat
      { цикл выполняется, пока B=false }
until B
```

Замечу здесь, что «**if B then...**» равносильно «**if B=TRUE then...**».

К чему ещё годны булевы данные? С ними производят логические операции, но к операциям обратимся чуть позже, — пора вернуться к нашей автомобильной задаче.

С высоты птичьего полета

Напомню, что мы работаем над программой для навигатора автомобиля, принимающего сигналы от спутников системы ГЛОНАСС. Из космоса прекрасно видны все улицы и пробки города. Пусть все возможные маршруты от дома до школы известны заранее, а спутник сообщает лишь о том, открыта ли для движения та или иная улица. Если улица открыта, спутник сообщает об этом значением **TRUE**, а иначе — значением **FALSE**. Увы, к настоящему спутнику мы пока не подключены, и вводить данные о пробках придется вручную. Результатом работы нашей программы будет сообщение о том, можно ли проехать в школу (**TRUE**), или нет (**FALSE**).

Вот первый маршрут. Предположим, путь от дома до школы пролегает по двум улицам так, как показано на рис. 31.

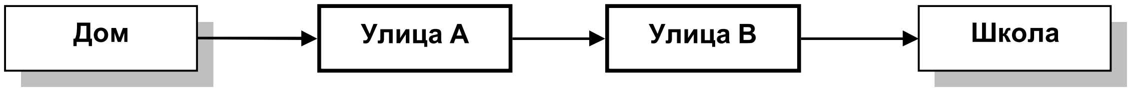


Рис. 31 – – Схема первого маршрута

Очевидно, что отразить состояние двух улиц можно двумя булевыми переменными, назовем их **A** и **B**. Объявим переменные и введем данные в них.

```
var  A, B : Boolean;  S: string;
begin
      Write('Улица A открыта? ');  Readln(S);   A:= S='1';
      Write('Улица B открыта? ');  Readln(S);   B:= S='1';
```

Здесь, как мы условились раньше, значение **TRUE** вводится цифрой «1».

Обратите внимание на новую для вас процедуру **Write**, — это «младшая сестра» процедуры **Writeln**. В отличие от «старшей сестры», после вывода сообщения она не переводит курсор на следующую строчку, — это удобно при запросе данных.

Ну-с, данные со спутника введены, и можно заняться их обработкой. Ясно, что для проезда в школу обе улицы должны быть открыты. Условными операторами это нехитрое рассуждение можно выразить так.

```
      S:='Топай пешком';
      if A then
            if B then S:='Поезжай на машине!';
```

Исходное значение — «Топай пешком» — заносим в переменную **S** заранее. Оно изменится тогда, когда обе булевы переменные станут равны **TRUE**. Согласитесь, это решение из двух условных операторов оказалось несложным. Но до поры до времени. Что, если маршрутов станет много, и каждый будет пролегать через несколько улиц? Программа превратится в нагромождение условных операторов, больше похожее на хаос землетрясения! Страшно? Тогда рассмотрим другой подход. Суть его в том, чтобы выразить решение на обычном человеческом языке, а затем превратить это высказывание в логическое выражение.

Решение нашей задачи можно высказать так: «проехать можно, если открыта улица **A** И открыта улица **B**». Обратите внимание на подчеркнутый мною союз «И». Чтобы превратить это рассуждение в логическое выражение и записать на Паскале, надо лишь перевести союз «И» на английский язык — это будет «AND», а названия улиц заменить логическими переменными **A** и **B**. И вот результат такого перевода.

```
      if A and B
            then S:='Поезжай на машине!'
            else S:='Топай пешком!';
```

Вместо двух условных операторов остался один. Готовая программа будет такой.

```
{ P_13_1 – первый маршрут проезда }
var  A, B : Boolean;  S: string;
begin
      { ввод данных со «спутника» }
      Write('Улица A:');  Readln(S); A:= S='1';
      Write('Улица B:');  Readln(S); B:= S='1';
      { решение }
      if A and B
            then S:='Поезжай на машине!'
            else S:='Топай пешком!';
      Writeln(S); Readln
end.
```

Испытайте программу при разных сочетаниях входных данных и проверьте, не врёт ли она?

Теперь рассмотрим другой маршрут, здесь попасть в школу можно по любой из двух улиц (рис. 32).

Рис. 32 – Схема проезда, второй вариант

Обычным языком молвим так: «проезд возможен, если открыта улица **A** ИЛИ открыта улица **B**». Союз «ИЛИ» тоже припасен в Паскале, по-английски он пишется «OR». В этом случае решение будет таким.

```
      if A or B
            then S:='Поезжай на машине!'
            else S:='Топай пешком!';
```

А вот маршрут на рис. 33 более замысловат.

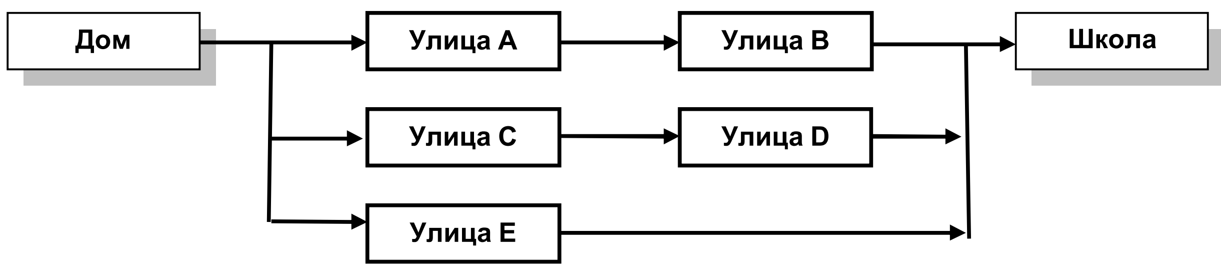


Рис. 33 – Схема проезда, третий вариант

Слабо ли вам выразить решение для этого случая? Сказать на обычном языке легко: «проехать можно, если открыта **A** И открыта **B** ИЛИ открыта **C** И открыта **D** ИЛИ открыта **E**». Слово «улица» я пропустил. Всё, решение готово! Осталось лишь перевести его на язык Паскаль.

```
if A and B or C and D or E
   then S:='Поезжай на машине!'
   else S:='Топай пешком!';
```

Как просто! Здесь опять подчеркнуто логическое выражение. Только теперь оно составлено из булевых переменных и булевых операций **AND** (И) и **OR** (ИЛИ). Иногда эти операции называют логическим умножением и логическим сложением. Сходство с арифметикой здесь в том, что каждая логическая операция обладает в выражении своим старшинством: умножение **AND** выполняется раньше сложения **OR**. Когда эту последовательность надо изменить, применяют скобки. Пример такого рода показан на рис. 34 (перекресток).

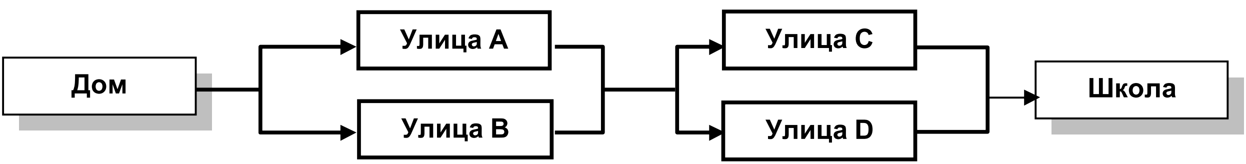


Рис. 34 – Схема проезда, четвертый вариант

Сначала скажем словами: «проехать можно, если открыта **A** ИЛИ открыта **B** И открыта **C** ИЛИ открыта **D**». Переведя на Паскаль буквально, без скобок, получим:

```
if A or B and C or D
   then S:='Поезжай на машине!'
   else S:='Топай пешком!';
```

Поскольку логическое умножение выполняется раньше сложения, Паскаль поймет это так: **A or (B and C) or D**. Но это не то, что мы хотели! Правильно будет записать наше решение со скобками:

```
if (A or B) and (C or D)
    then S:='Поезжай на машине!'
    else S:='Топай пешком!';
```

Наконец, рассмотрим маршрут на рис. 35, где путь преграждает шлагбаум. Договоримся, что закрытому шлагбауму соответствует значение **TRUE** (то есть, в сравнении с улицами тут всё наоборот).

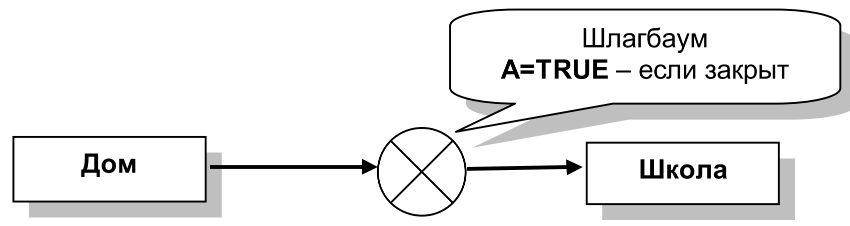


Рис. 35 – Схема проезда, пятый вариант

Рассуждая как обычно, скажем так: «проезд возможен, если НЕ закрыт шлагбаум». Здесь применено логическое отрицание НЕ, что по английски значит «NOT». Решение на Паскале будет таким.

```
if not A
    then S:='Поезжай на машине!'
    else S:='Топай пешком!';
```

В отличие от двух предыдущих операций, логическое отрицание — одноместная операция, ей нужен лишь один операнд. Логическое отрицание имеет наивысший приоритет, и выполняется раньше логического умножения и сложения.

Парад логических операций

Итак, посредством логических операций мы переводим рассуждения с человеческого языка на формальный язык программирования, получая при этом логические (булевы) выражения. Логические данные в Паскале можно сравнивать и выполнять с ними четыре логические операции, три из которых вам уже знакомы. Рассмотрим свойства этих операций.

Логическое отрицание **NOT** («НЕ»). Имеет наивысший приоритет, то есть, при отсутствии скобок выполняется в первую очередь. Это одноместная операция, поскольку требует лишь одного операнда. По своему действию она напоминает знак «минус» для чисел, поскольку изменяет значение операнда на противоположное. Правила для этой операции таковы.

```
NOT   FALSE =    TRUE
NOT   TRUE  =    FALSE
```

Логическое умножение **AND** («И»). Приоритет ниже, чем у **NOT**, но выше, чем у логического сложения **OR**. Требует двух операндов, и в результате дает **TRUE**, если оба операнда равны **TRUE**.

```
FALSE  AND  FALSE  = FALSE
FALSE  AND  TRUE   = FALSE
TRUE   AND  FALSE  = FALSE
TRUE   AND  TRUE   = TRUE
```

Логическое сложение **OR** («ИЛИ»). Приоритет самый низкий, — выполняется в последнюю очередь. Требует двух операндов и в результате дает **TRUE**, если хотя бы один из операндов равен **TRUE**.

```
FALSE  OR   FALSE  = FALSE
FALSE  OR   TRUE   = TRUE
TRUE   OR   FALSE  = TRUE
TRUE   OR   TRUE   = TRUE
```

Итоги

- Информация – это то, что устраняет неопределенность.
- Получая ответ на вопрос, мы получаем информацию. Количество информации можно измерить.
- Наименьшая порция информации – бит – содержится в ответе на простой вопрос («да» или «нет»). Это количество принято за единицу измерения информации.
- Память компьютера состоит из элементарных ячеек – триггеров, каждый из которых хранит один бит информации. Восемь битов составляют один байт.
- Подобие триггеров в Паскале – булевы (логические) переменные. Они принимают только одно из двух значений: **TRUE** (истина) или **FALSE** (ложь).
- Булевы переменные в сочетании с логическими операциями **OR**, **AND**, **NOT** и скобками образуют булево выражение. Скобки нужны для изменения естественного порядка выполнения операций.
- Булевы выражения используют в условных и циклических операторах.

А слабо?

А) Что будет напечатано в результате выполнения следующего фрагмента?

```
S:='123';
Writeln ('123'=S);
```

Б) Переведите на русский язык это выражение.

```
if (S='') and (A or B) then ...
```

В) Напишите программу к бортовому компьютеру для маршрута на рис. 36.

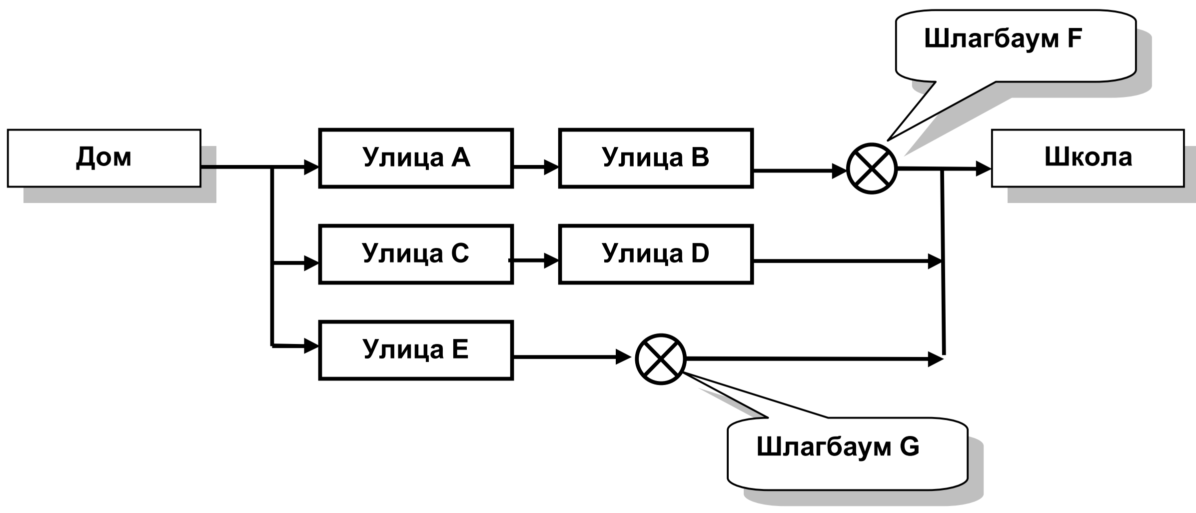


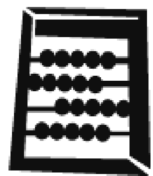
Рис. 36 – Схема проезда к задаче «В»

Г) В переменные **M1**, **M2** и **M3** вводится итог подбрасывания трех монет так, что **TRUE** соответствует «орел», а **FALSE** — «решка». Надо составить пять выражений таких, чтобы они выдавали **TRUE** для следующих случаев:

- у всех монет выпал «орел»;
- у всех монет выпала «решка»;
- все монеты упали одинаково;
- у первой – «решка», у прочих – «орел»;
- у первой – «орел», а две остальные упали одинаково.

Подсказка: логические данные можно сравнивать; сравнение обладает самым низким приоритетом, и потому внутри выражений заключается в скобки, например: **M1 and (M2=M3)**.

Глава 14
Дважды два – четыре



Первые компьютеры назывались электронными вычислительными машинами (ЭВМ). Хотите — верьте, хотите — нет, но тогда на них не документы печатали и не фильмы смотрели, а вычисляли. С тех пор компьютеры научились многому и даже обыгрывают в шахматы чемпионов мира, однако, их способность к счету по-прежнему в цене.

Поможем братьям нашим меньшим

Пора и нам обратиться к вычислительным талантам компьютера. Не будем тратить попусту время, и по ходу дела соорудим полезную программу. Вы сможете испытать её на живом человеке, если найдёте первоклашку, зубрящего таблицу умножения. Уверен, что он с удовольствием подвергнет себя такому испытанию. Итак, наша очередная программа — экзаменатор. Суть её проста: компьютер предлагает ученику два числа и ждет от него ответа — произведения этих чисел. За правильный ответ ученика поощряют, а иначе его ждет «нахлобучка».

Числа и действия с ними

Скажу честно: знакомых нам типов данных — **STRING** и **BOOLEAN** — не хватит для решения поставленной задачи. Для вычислений в Паскале припасены другие типы данных, один из которых называется **INTEGER**, что переводится как целое. Из названия следует, что переменные такого типа могут хранить целые числа (положительные и отрицательные), например 10, 25, -14. Переменные целого типа объявляют следующим образом:

```
var N, M : integer;
```

Таким переменным можно присваивать выражения целого типа, состоящие из чисел, арифметических операций, скобок и других переменных, например:

```
      N := 19;    M :=-25;
      M := 20 + 3*N;
```

К арифметическим операциям относятся:

- сложение (+) и вычитание (–);
- умножение (*) и деление (**DIV**);
- нахождение остатка от деления (**MOD**).

Здесь **DIV** и **MOD** — это ключевые слова языка. Примеры деления и нахождения остатка показаны ниже (в комментариях указаны результаты).

```
      N := 10 div 2;    { =5 }            M := 10 mod 2;    { =0 }
      N := 10 div 3;    { =3 }            M := 10 mod 3;    { =1 }
      N := 10 div 4;    { =2 }            M := 10 mod 4;    { =2 }
      N := 10 div 5;    { =2 }            M := 10 mod 5;    { =0 }
      N := 10 div 6;    { =1 }            M := 10 mod 6;    { =4 }
```

Как видите, операции с целыми числами дают целый результат даже при делении, поскольку дробная часть отбрасывается.

Числовые переменные и выражения можно сравнивать между собой на равенство (=), неравенство (<>), больше (>), меньше (<), больше или равно (>=), меньше или равно (<=). При сравнении получается, как всегда, булев результат, например:

```
var   X, Y: integer;
      B: Boolean;
begin
      X:=5;       Y:=10;
      B:= X=Y;    { B = FALSE }
      B:= X<Y;    { B = TRUE }
      B:= X=Y-5;  { B = TRUE }
end.
```

А как быть с вводом и выводом числовых данных, нет ли тут сложностей? К счастью, нет. Так же как и строки, числовые данные вводятся процедурой **Readln**, а печатаются процедурами **Write** и **Writeln**, например:

```
      Readln(X);
      Writeln(X);
      Writeln('Y=', X+10);
```

В последнем операторе на экран выводится строковая константа **'Y='** и результат сложения **X+10**.

Теперь вы снабжены всем необходимым для написания экзаменатора.

Алгоритм экзаменатора

Прежде всего, уточним алгоритм создаваемой программы. Живой экзаменатор сам придумывает примеры для умножения. Но нам это пока не под силу — маловато знаний — отложим этот вариант до следующей главы. А пока экзаменуемый будет сам «создавать себе проблемы», то есть будет вводить сомножители по запросу программы вручную. Пример диалога может выглядеть, например, так.

```
Первый сомножитель A = 7
Второй сомножитель B = 7
Произведение A*B = 47
Ошибка, повтори таблицу умножения!
```

И так далее. Здесь подчеркнутые числа **7**, **7** и **47** пользователь ввел сам. Разумеется, что задания надо решать многократно, в цикле. Для выхода из цикла нужен какой-то признак, сигнал. Пусть таким сигналом будет ввод нуля в качестве ответа. Тогда блок-схема программы получается такой (рис. 37).

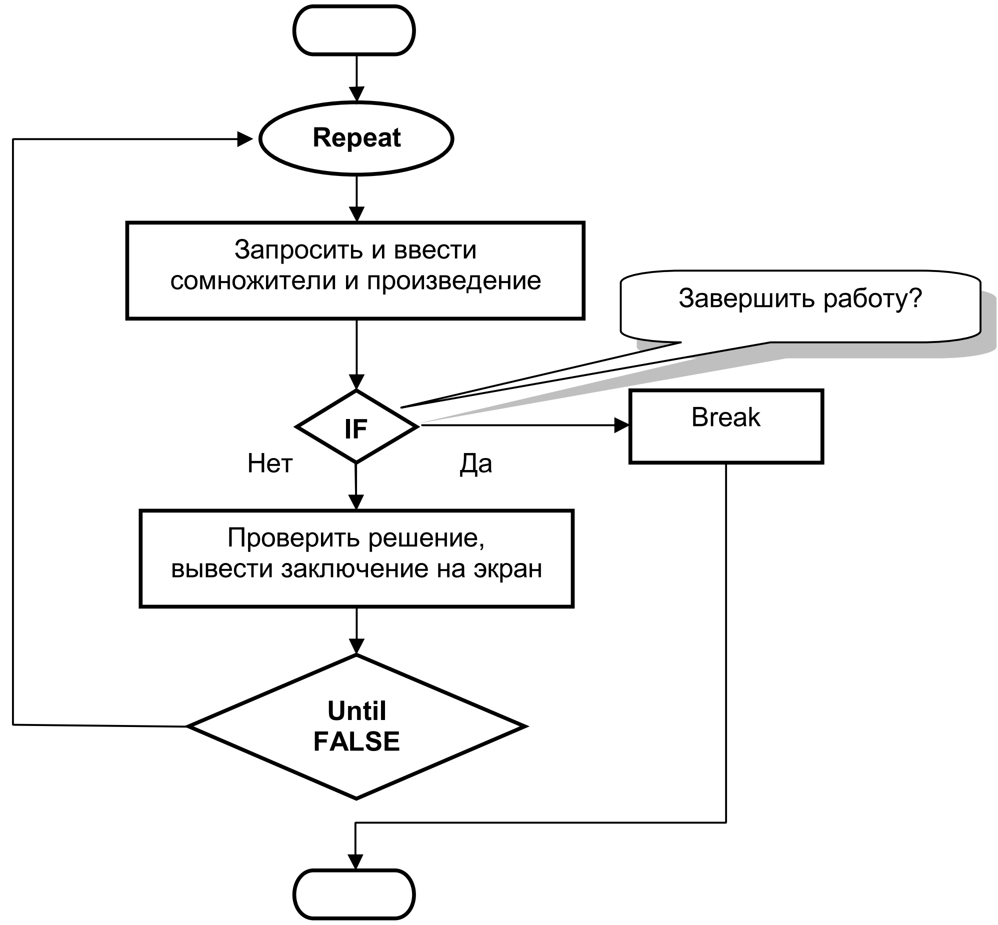


Рис. 37 – Блок-схема программы проверки таблицы умножения

Обратите внимание на условие в операторе цикла **REPEAT-UNTIL**, — оно равно **FALSE**. Такой цикл будет продолжаться бесконечно, и выйти из него можно лишь процедурой **BREAK**, как показано на блок-схеме.

Экзаменатор, первый вариант

Вот теперь всё готово для написания следующей программы.

```
{ P_14_1 - экзаменатор таблицы умножения, первый вариант }
var  A, B, C : integer;  { сомножители и произведение }
      R: Boolean; { результат сравнения }
      S: string;        { сообщение для вывода на экран }
begin
      repeat
            { ввод сомножителей и произведения }
            Write('Первый сомножитель A = ');  Readln(A);
            Write('Второй сомножитель B = ');  Readln(B);
            Write('Произведение A*B = ');  Readln(C);
            if C=0 then break;  { завершение цикла, если C=0 }
            { проверяем правильность вычисления }
            R:= A*B=C;  { R=true, если верно }
            if R
                  then S:= 'Молодец, правильно!'
                  else S:= 'Ошибка, повтори таблицу умножения!';
            Writeln(S);
      until false; { бесконечный цикл }
end.
```

Запустите программу и проверьте её работу. В следующий раз мы научим её придумывать сомножители, — так будет честнее. А пока подведем итоги.

Итоги

- Для вычислений в Паскале предусмотрены данные числового типа (**`INTEGER`**).
- К данным целого типа могут применяться четыре арифметических операции, а также операция нахождения остатка от деления.
- В результате сравнения численных данных получается булев результат, который может быть применен везде, где проверяется условие.
- Числовые данные вводятся оператором **`Readln`** и выводятся операторами **`Write`** и **`Writeln`**;
- Числовым переменным нельзя присваивать строковые значения и наоборот: строковым переменным нельзя присваивать числовые значения.

А слабо?

А) Найдите ошибки в следующей программе и объясните их.

```
var   N, M : integer;
      S : string;
begin
      N:= '10';
      S:= N + 5;
      M:= S - 1;
      if S=N then;
end.
```

Проверьте свои догадки, призвав на помощь компилятор.

Б) Перепишите программу P_14_1, не прибегая к процедуре **Break**. В чем, по-вашему, слабость этого второго варианта? Можно ли обойтись в программе P_14_1 без булевой переменной **R** и строковой **S**? Напишите такой вариант программы. Или слабо?

В) Пусть программа запросит три числа: **A**, **B** и **C**, а затем напечатает большее из них. Подсказка: примените булевы выражения вкупе с операциями сравнения, которые в булевых выражениях надо заключать в скобки, например:

```
      if (A>=B) and (A>=C) then . . .
```

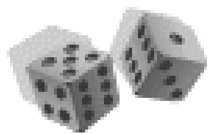
Примечание. Скобки ставят по той причине, что булевы операции можно выполнять и с числами, и такие операции приоритетней операций сравнения. О применении логических операций к числам сказано в главе 48.

Г) В стене прорублено прямоугольное сквозное отверстие со сторонами **A** и **B**. Пусть ваши программы разберутся, пройдет ли в него кирпич с ребрами **X**, **Y**, **Z**. Сделайте две программы для таких случаев:

- Известно, что **A<B** и **X<Y<Z**.
- Соотношение между сторонами неизвестно, и программе самой надо выяснить высоту и ширину, как отверстия, так и кирпича.

Д) Площадь земельного участка вычисляется умножением его сторон **A** и **B**. В программу вводятся стороны двух участков (**A1**, **B1** и **A2**, **B2**), пусть она напечатает ширину и длину того участка, что больше по площади. Ширина должна быть не больше длины.

Глава 15
Айда в Монте-Карло!



Монте-Карло — весёлый пригород в княжестве Монако, славный своими игорными заведениями. Там, по словам Поэта, жертвуют необходимым в надежде приобрести излишнее. Но к чему нам игорный бизнес, — спросите, — когда мы заняты программой-экзаменатором? Не забывайте, однако, что наш первоклашка пока ещё сам придумывает себе примеры, а это неразумно. Избавим его от ввода сомножителей, — пусть программа сама «изобретает» их. Потому и обращаемся к азартным играм.

Куда ни глянь – то процедура, то функция!

Современные программы очень сложны. И, как любое крупное изделие, заключают в себе труд десятков и сотен специалистов. Трудно поверить, но большинство программистов, работающих над крупным проектом, не видят его в целом, что не мешает им выполнять свою часть работы. Как такое возможно?

Чтобы понять это, оглянитесь вокруг. Обойдутся ли жители города или страны друг без друга? Кем бы ты ни был — врачом, водителем или сапожником — не проживешь без услуг иных граждан, — все мы зависим друг от друга! Но спросите, к примеру, сталевара, куда пойдет выплавляемая им сталь? Он только плечами пожмет!

Работа программистов организована по тем же законам — законам специализации и кооперации. Специализация — это углубление в некоторую узкую область, специальность. Скажем, одни программисты наловчились писать драйверы, другие — базы данных, а третьи — графический интерфейс. Кооперация — это слаженное соединение усилий разных специалистов, — за это отвечает руководящая «верхушка» программного проекта (подобно тому, как правительство руководит страной).

Конечно, «руками водить», распределяя работу, может каждый (некоторые так и думают). Но толку будет чуть, если согласованную работу программистов не поддержать техническими средствами. Современные языки программирования, в том числе Паскаль, такие средства дают. Одно из них — механизм процедур и функций. Процедуры и функции — это готовые «кусочки» программ, выполняющие некоторые оговоренные действия. Иногда их называют общим именем — подпрограммы. Такие «кусочки» могут создаваться разными программистами и сохраняться в специальных файлах — библиотеках. Есть библиотеки и в Паскале.

Для применения библиотечной процедуры или функции достаточно знать её имя и список передаваемых ей параметров. А вот думать о том, как устроена эта процедура внутри, не обязательно. Хочешь выполнить какое-то действие из библиотеки? Тогда помести в нужном месте программы вызов подходящей процедуры и укажи параметры. Кстати, мы с вами уже делаем это, вызывая

процедуры **Readln** и **Writeln**. В библиотеках Паскаля припасены процедуры и функции на многие случаи жизни, со временем вы узнаете о них больше.

Чем же отличаются функции от процедур? Функции обладают теми же возможностями, что и процедуры, но вдобавок возвращают значение некоторого типа (числовое, логическое или иное). Поэтому вызов функции можно вставлять внутрь выражения, что очень удобно. Как это работает, вы увидите сей же час.

Госпожа удача

Вернемся к нашему экзаменатору, где надо придумать способ формирования случайных чисел в пределах от 1 до 10. Будь под рукой игральный кубик из Монте-Карло, я бы не связывался с компьютером! Впрочем, в библиотеке Паскаля есть такой «кубик» — это функция по имени **Random**, что переводится как «случайный, беспорядочный». Этой функции необходимо задать один параметр — число **N**, определяющее предел для случайного числа. В ответ функция возвращает некоторое случайное число в диапазоне от нуля до **N-1**. Например, в следующем операторе в переменную **X** попадет некоторое число в диапазоне от 0 до 9.

```
      X:= Random(10);
```

Говорят, что функция **Random** генерирует случайные числа. Чтобы лучше понять, как это работает, введите и запустите следующую программу.

```
{ P_15_1 – пятикратный вызов функции Random(100) }
begin
      Writeln( Random(100) );
      Writeln( Random(100) );
      Writeln( Random(100) );
      Writeln( Random(100) );
      Writeln( Random(100) );
      Readln;
end.
```

Здесь печатаются целые числа, возвращаемые функцией **Random**. И хотя параметр функции во всех вызовах одинаков (100), результаты получатся разными. При этом все они лежат в диапазоне от 0 до 99. Таким образом, параметр функции **Random** управляет диапазоном генерируемых чисел.

Запустите эту программу ещё пару раз и сравните результаты. Вы заметили, что они повторяются? Так и должно быть! Всё потому, что функция **Random** создает псевдослучайную последовательность чисел. «Псевдо» — значит «не совсем случайную». Эта особенность функции полезна при отладке программ. Но в экзаменующей программе надо получать разные последовательности чисел, иначе смышленые школяры приноровятся к экзаменатору!

Этого можно добиться применением ещё одной процедуры. Она называется **Randomize** (что значит «уравнять шансы» или «перемешать») и не требует параметров. Вызвав эту процедуру единожды в начале программы, мы смешаем карты и заставим функцию **Random** при повторных запусках программы генерировать разные последовательности чисел. Итак, вставьте вызов процедуры **Randomize** в начало программы и повторите опыты, запустив программу несколько раз подряд.

```
{ P_15_2 – пятикратный вызов функции Random(100) после Randomize }
Begin
      Randomize;
      Writeln( Random(100) );
      Writeln( Random(100) );
      Writeln( Random(100) );
      Writeln( Random(100) );
      Writeln( Random(100) );
      Readln;
end.
```

Теперь от успешного финиша проекта нас отделяет один шаг: придумаем способ генерировать числа от 1 до 10 (а не от 0 до 9). Очевидно, что простое арифметическое выражение решает эту проблему.

```
      X:= 1+ Random(10);      { генерация чисел от 1 до 10 }
```

Сейчас вы готовы написать второй вариант экзаменатора, вот каким он может быть (новые операторы, как обычно, подчеркнуты).

```
{ P_15_3 – программа-экзаменатор, версия 2 }
var  A, B, C : integer;  { сомножители и произведение }
begin
      Randomize;  { смешиваем «карты» }
      repeat
            A:= 1+ Random(10);      B:= 1+ Random(10);
            Write('Сколько будет ', A,' x ',B, ' ? ');
            Readln(C);
            if C=0 then break;  { завершение цикла, если C=0 }
            { проверяем правильность вычисления }
            if A*B=C
                  then Writeln('Молодец, правильно!')
                  else Writeln('Ошибка, повтори таблицу умножения!');
      until false; { бесконечный цикл! }
end.
```

Обратите внимание на вывод задания для умножения.

```
      Write('Сколько будет ', A,' x ',B, ' ? ');
```

Здесь процедура **Write** содержит уже пять параметров: две числовые переменные и три строковые константы. Так, при **A=3** и **B=7** на экране появится вопрос: «Сколько будет 3 x 7 ?». Остальные операторы программы обойдутся без моих пояснений.

Итоги

- В языках программирования предусмотрены средства для согласованной работы программистов, одно из них – библиотеки процедур и функций.
- Отличие процедур от функций состоит в том, что процедура лишь выполняет оговоренные действия, а функция вдобавок возвращает данные некоторого типа.
- Для генерации случайных последовательностей чисел применяют функцию **Random** и процедуру **Randomize**.
- Функция **Random(N)** возвращает псевдослучайное число, лежащее в пределах от 0 до **N-1**. При повторных запусках программы эта серия чисел повторяется, если заранее не вызвана процедура **Randomize**.
- Вызов процедуры **Randomize** в начале программы приводит к генерации функцией **Random** разных серий псевдослучайных чисел.

А слабо?

А) В каких пределах будут генерироваться числа следующими выражениями:

10+Random(10);

Random(20);

Random(10) + Random(10);

Random(5) + Random(5) + Random(5) + Random(5);

Проверьте себя на компьютере!

Б) Сколько чисел будет напечатано следующей программой? Испытайте на практике.

```
var x : integer;
begin
      repeat
            x := Random(20);
            Writeln(x);
      until x=1;
end.
```

В) А если в начало предыдущей программы вставить **Randomize**? Можно ли предсказать результат? Или слабо?

Г) Найдите способ сформировать ряд случайных булевых значений (**False**, **True**), напечатайте 20 из них. Подсказка: булевы значения получаются сравнением двух случайных целых чисел.

Д) Сгенерируйте два случайных числа (в диапазоне от 1 до 10) так, чтобы они не совпадали. Сделайте то же самое для трех чисел.

Глава 16
Делу время, а потехе час

Наши программы — и часовой, и экзаменатор — такие любопытные! Всё спрашивают что-то: то пароль им подавай, то таблицу умножения! Не поменяться ли с компьютером местами? Теперь мы будем спрашивать, а он — отвечать.

Вот веселая и глупая игра: «вопрос-ответ», суть которой такова. Две колоды карточек — одну с вопросами, а другую с ответами — тасуют и кладут рубашками вверх. Кто-то из сидящей вокруг стола компании берет наугад карточку из «вопросительной» колоды и читает вопрос своему соседу. Тот вынимает наугад карточку из колоды с ответами и оглашает его. К примеру, на вопрос «Как пройти в библиотеку?» можно получить ответ: «Волк, коза и капуста».

Создадим нечто похожее для игры на компьютере, он будет отвечать на вопросы, вводимые пользователем с клавиатуры. Разумеется, что ответы заготовим в программе заранее, а выбирать их будем случайно.

Потемкинская лестница

В первую минуту эта задачка представится вам легкой забавой, — я попробую угадать ход ваших мыслей. Во-первых, не будем обращать внимание на вопрос пользователя, — для подготовки ответа он не важен. После ввода вопроса сгенерируем случайное число и выберем один из заранее подготовленных ответов, согласуясь с этим числом. А как организовать выход из программы? Вот тут вопрос пользователя будет кстати, — приняв пустой вопрос, мы завершим программу. Этим мыслям отвечает блок-схема на рис. 38.

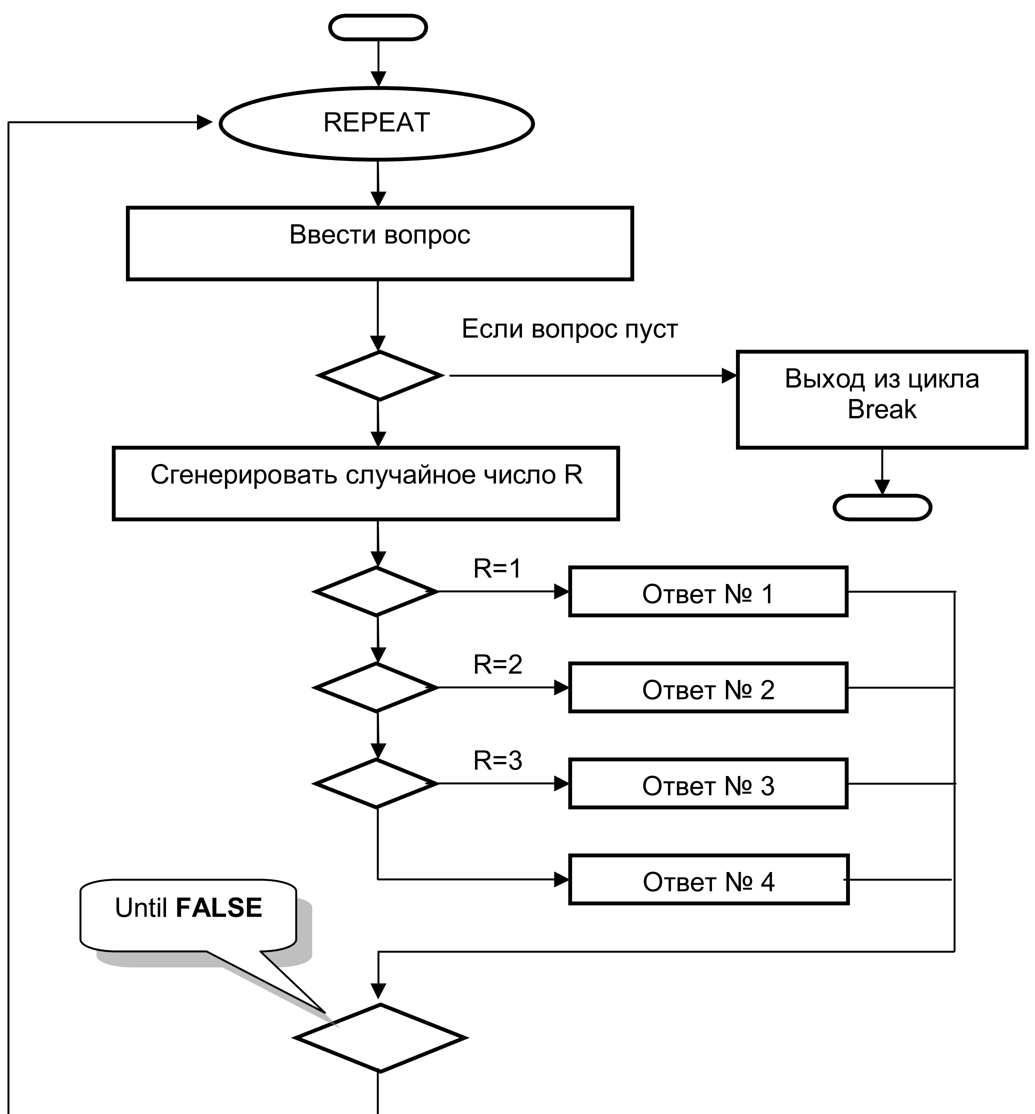


Рис. 38 – Блок-схема выбора из четырех вариантов

Рассмотрим условный оператор, выбирающий один из четырех ответов на основе случайного содержимого переменной **R**.

```
if R=1
      then S:='Ответ 1'
else if R=2
      then S:=' Ответ 2'
else if R=3
      then S:=' Ответ 3'
else S:=' Ответ 4';
```

Вложенные друг в друга условные операторы образуют «лесенку», — такое расположение удобно для чтения программы. А если заготовить больше ответов? Тогда «лесенка» дорастет до потемкинской лестницы, что в чудном городе Одессе!

Эта проблема — типичный случай в программировании. На сей случай в Паскале запасен оператор выбора **CASE** (что так и переводится — «случай»). В отличие от оператора **IF**, содержащего лишь две ветви, в операторе **CASE** их много — на все случаи жизни. Оператор записывают следующим образом:

```
case X of
      n1: Оператор_1;
      n2: Оператор_2;
      . . .
      else Оператор_n
end;
```

Конструкция построена на четырех ключевых словах **CASE-OF-ELSE-END**. Выражение целого типа **X** служит условием, по которому выбирается одна из числовых меток: **n1**, **n2** и так далее (метки — это целые числа). Работает оператор так. Если выражение **X** = **n1**, то выполняется **оператор_1**, если **X** = **n2**, то выполняется **оператор_2** и так далее. Если **X** не соответствует ни одной метке, сработает оператор, указанный после **ELSE**. А если ветвь **ELSE** отсутствует? Тогда ничего не выполняется.

Вот пример. Если в результате вычисления выражения **Random(20)+1** будет получено число от 1 до 3, то переменной **S** будет присвоено соответствующее слово, а иначе она станет пустой.

```
case Random(20)+1 of
      1: S:= 'Первый';
      2: S:= 'Второй';
      3: S:= 'Третий';
      else S:= '';
end;
```

Если оператор **CASE** применить к нашей шуточной (или нешуточной) программе, то получится вот что.

```
{ P_16_1 - игра «вопрос - ответ» }
var  S: string;
begin
      Randomize; { чтобы случайный ряд не повторялся  }
      repeat
            Write('Ваш вопрос: '); Readln(S);
            if S='' then break;  { завершение цикла, если строка пуста }
            case Random(5) of
                  0: S:='Когда рак на горе свиснет';
                  1: S:='После дождика в четверг';
                  2: S:='За углом налево';
                  3: S:='Это элементарно, Ватсон!';
                  else S:='Не знаю, я не местный';
            end;
            Writeln(S); { печать ответа }
      until false; { бесконечный цикл }
end.
```

Добавьте несколько смешных ответов, увеличив соответственно параметр функции **Random**, а затем испытайте программу на своих приятелях.

Итоги

- Для условных переходов со многими ветвями в Паскале предусмотрен оператор выбора **CASE-OF-ELSE-END**.
- Каждая ветвь оператора **CASE** начинается с числовой метки, за которой следует выполняемый оператор.
- Метки могут следовать в любом порядке (не только по возрастанию).
- Ветвь оператора **CASE** выбирается в зависимости от числового выражения в условии. Если ни одна метка не соответствует условию выбора, выполняется оператор, указанный после **ELSE**. Если ветвь **ELSE** не указана, то ничего не выполняется.
- Для исполнения внутри ветви нескольких операторов их объединяют в блок **BEGIN-END**.

А слабо?

А) Какой ответ будет выпадать чаще других, если условием в операторе **`CASE`** нашей программы поставить выражение **`Random(100)`**?

Б) Напишите программу, которая бы запрашивала номер дня недели, и в ответ печатала бы название этого дня («понедельник», «вторник» и так далее).

В) Пусть пользователь введет число — свой возраст в годах. Ваша программа должна напечатать фразу: «Вам столько-то лет» с правильным окончанием, например: «Вам 20 *лет*», или «Вам 34 *года*», или «Вам 41 *год*». Подсказка: надо определить последнюю цифру года операцией **`MOD 10`**. Некоторые числа выпадают из общего правила, их надо проверить особо (например, 11, 12, 13, 14).

Г) Пользователь вводит число — номер месяца от 1 до 12, а программа должна сообщить соответствующее ему время года: зима, весна, лето, осень. Подсказка: в одной ветви можно применить несколько меток, например:

```
case N of
      1, 2, 12 : Writeln('Зима');
```

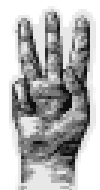
Д) Танк в компьютерной игре может двигаться в одном из четырех направлений, обозначим их числами: 1 — север, 2 — восток, 3 — юг, 4 — запад. Направление движения изменяется тремя командами: 1 — поворот направо, 2 — поворот налево, 3 — поворот кругом. Пользователь вводит начальное направление движения, а затем ряд команд. Программа должна определять и печатать всякий раз новое направление. Выход из цикла — команда 0.

Е) Исходные позиции шахматных фигур известны всякому (если вы — исключение из правила, ознакомьтесь с основами шахмат). Пользователь в цикле вводит число, по которому программа печатает название фигуры, стоящей на соответствующей вертикали шахматной доски (от 1 до 8). Ноль служит для выхода из цикла, а на все прочие числа программа сообщает об ошибке.

Ж) Программа запрашивает в цикле два числа: вертикаль и горизонталь шахматной доски (числа от 1 до 8), а затем печатает цвет клетки на их пересечении. Если хотя бы одно из чисел равно нулю, цикл завершается. Если числа выходят за указанные пределы, сообщает об ошибке и повторяет запрос чисел.

Подсказка: на пересечении 1-й строки и 1-го столбца находится чёрная клетка.

Глава 17
И вновь за парту



Натешившись глупой игрушкой, сотворенной нами в предыдущей главе, с новыми силами набросимся на экзаменатора, ведь он ещё не совсем настоящий. Настоящий экзаменатор выставляет оценку, не так ли? Пусть наша программа оценивает ученика по количеству допущенных ошибок. Ответив, к примеру, на 15 вопросов, ученик получит:

- «отлично» – за ноль ошибок;
- «хорошо» – за 1-2 ошибки;
- «удовлетворительно» – за 3-5 ошибок;
- «неуд» – за 6 ошибок и более.

Цикл со счетчиком

Очевидно, что новая версия экзаменатора будет циклической (рис. 39), только условие выхода из цикла будет теперь другим.

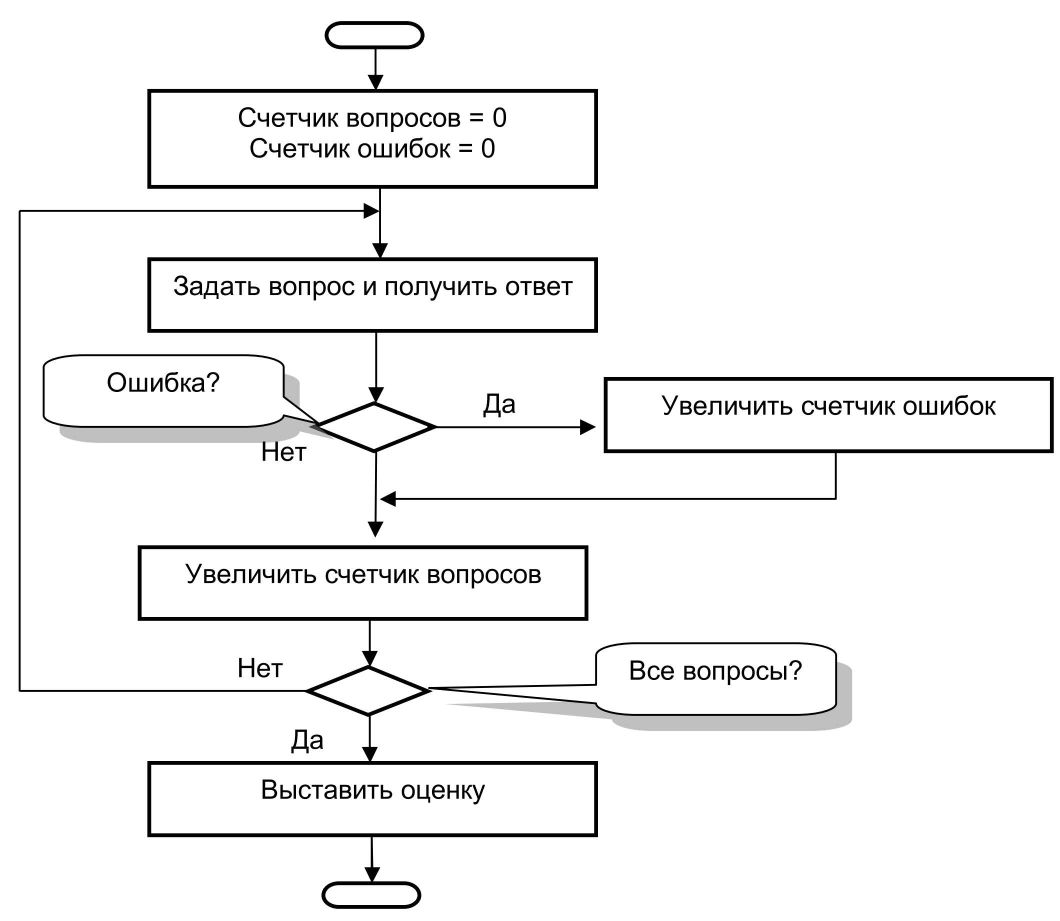


Рис. 39 – Блок-схема экзаменатора, выставляющего оценку

Основное отличие этой версии от предыдущих состоит в применении счетчиков. Один из них подсчитывает количество заданных вопросов (то есть

проходов цикла), а другой — количество ошибок. Что такое счетчик? Это числовая переменная, наращиваемая по ходу выполнения программы. Сначала рассмотрим тонкости, связанные с подсчетом вопросов.

Зададимся простой задачей: распечатать на экране числа от 1 до 10. Вот как это делается оператором **REPEAT-UNTIL**.

```
var   N : integer;      { счетчик }
begin
      N:=1;
      repeat
            Writeln(N);
            N:= N+1;
      until N>10
end.
```

Первый из подчеркнутых операторов устанавливает счетчик цикла в единицу, — программисты называют это инициализацией цикла. Другой подчеркнутый оператор наращивает счетчик. Эта пара операторов, как принято говорить, организует цикл. Слабость такой организации в том, что действуют операторы порознь, а это таит две неприятности.

Человеку свойственно ошибаться, и программисты забывают порой вставить в программу ту или иную строчку. Что случится, если пропустить инициализацию? Значение счетчика **N** останется неопределенным, и цикл выполнится непонятно сколько раз. А если проворонить второй оператор? Счетчик наращиваться не будет, и цикл станет повторяться вечно, — программа, как говорят, зациклится! Во избежание таких ошибок в Паскале предусмотрен цикл со счетчиком.

Цикл со счетчиком объединяет в одной конструкции три действия: инициализацию счетчика, его приращение и проверку условия завершения цикла. Если б написать его по-русски, то оператор выглядел бы так:

ДЛЯ N:= начальное_значение **ДО** конечное_значение **ВЫПОЛНИТЬ** оператор

Но русским Паскаль не владеет, а потому переведем это на английский:

FOR N:= начальное_значение **TO** конечное_значение **DO** оператор

Как видите, конструкция построена на трех ключевых словах: **FOR-TO-DO**. После слова **FOR** следует оператор присваивания начального значения счетчику цикла. За словом **TO** указывают конечное значение счетчика, а после **DO** — выполняемый внутри цикла оператор. Но где наращивается счетчик? А нигде, это происходит автоматически! Теперь задача распечатки чисел может быть решена одним составным оператором.

```
var   N : integer;      { счетчик }
begin
      for N:=1 to 10 do Writeln(N);
end.
```

Испытайте эту программку. Согласитесь, что ошибиться здесь труднее, чем в варианте с **REPEAT**. Как только вы написали **FOR**, то обязаны тут же указать начальное и конечное значения счетчика, а наращивать его Паскаль будет и без вас. В качестве начального и конечного значений вы вправе указать не только числа, но и выражения, — они будут вычислены один раз в начале цикла. Если начальное значение счетчика окажется равным конечному, цикл выполнится единожды. А если конечное значение окажется меньше начального, то ни разу!

Осталось ответить лишь на один вопрос: что, если внутри цикла надо выполнить несколько операторов? Ведь после слова **DO** предусмотрен лишь один. Впрочем, те, кто помнит об операторных скобках **BEGIN-END**, знают ответ. Напомню, что эти скобки превращают группу операторов в единый блок, этим мы и воспользуемся в новой версии экзаменатора.

```
{ P_17_1 – экзаменатор, выставляющий оценку }
var  A, B, C : integer; { сомножители и произведение }
      Q, E : integer;   { счетчик вопросов и счетчик ошибок }
      S: string;
begin
      Randomize;
      E:= 0; { обнуляем счетчики ошибок }
      for Q:= 1 to 15 do begin  { 15 вопросов }
            A:= 1+ Random(10);      B:= 1+ Random(10);
            Write(Q,')_Сколько будет ', A,' x ',B, ' ? ');
            Readln(C);
            { Если ответ неверный, увеличиваем счетчик ошибок }
            if A*B <> C then E:= E+1;
      end;  { цикл и блок завершаются здесь}
      case E of   { выставляем оценку }
            0:    S:='Отлично!';
            1,2:  S:='Хорошо';
            3..5: S:='Удовлетворительно';
            else  S:='Ну оччччень плохо!';
      end;
      Writeln(S, '  Нажмите Enter'); Readln;
end.
```

Рассмотрим изюминки этой программы. В операторе

```
Write(Q,') Сколько будет ', A,' x ',B, ' ? ');
```

вместе с вопросом печатается его порядковый номер **Q**.

Но самое интересное — это метки в операторе **CASE**. Напротив оценки «хорошо» стоит метка из двух разделенных запятой чисел (1, 2), — эта ветвь оператора **CASE** выполнится для этих двух значений. Такие объединенные метки могут содержать несколько чисел. А если числа следуют подряд, их заменяют числовым диапазоном — это два числа, разделенные двумя точками («многоточием»), причем первое число должно быть меньше второго. Такой диапазон (3..5) служит меткой для ветви «Удовлетворительно».

Итоги

- Цикл со счетчиком **FOR-TO-DO** удобен при известном количестве повторений, которое вычисляется при входе в цикл.
- Счетчик цикла внутри оператора наращивается автоматически, цикл завершается, когда счетчик превысит указанное максимальное значение.
- Оператор выбора **CASE-OF-ELSE-END** допускает метки из нескольких чисел, и даже диапазоны целых чисел.

А слабо?

А) Позвольте ученику отказаться от сдачи экзамена. Признаком отказа будет ввод нуля в качестве ответа. В этом случае надо досрочно выйти из цикла и обойти выставляющий оценку оператор (вспомните о процедуре **Break**).

Б) Напишите программу, которая по введенному числу дает заключение о том, какому дню недели оно соответствует — рабочему (1-5) или выходному (6,7), например:

```
День = 2
Рабочий
День = 7
Выходной
День = 20
Ошибка!
```

Здесь подчеркнутые числа напечатаны пользователем.

В) Напишите программу, которая, запросив число **N**, печатала бы числа от 1 до **N** в обратном порядке, например:

```
N = 3
3
2
1
```

Г) Существует вариант цикла **FOR**, где счетчик цикла не наращивается, а уменьшается, этот оператор выглядит так:

FOR N:= начальное_значение **DOWNTO** конечное_значение **DO** оператор

Ключевое слово **DOWNTO** задает счет в обратном порядке (**DOWN** — «вниз»); при этом начальное значение счетчика должно быть больше или равно конечному, иначе цикл не выполнится ни разу. Воспользуйтесь этим оператором для решения предыдущей задачи (задание В).

Д) Пусть программа запросит два числа **N** и **M**, а затем вычислит их произведение без использования операции умножения (*). Подсказка: организуйте цикл суммирования **N** раз числа **M**.

Е) Напишите программу, вычисляющую сумму чисел от 1 до **N**, где **N** — число, вводимое пользователем.

Ж) Напишите программу, вычисляющую сумму только тех чисел от 1 до **N**, которые делятся либо на три, либо на пять.

Задачи на темы предыдущих глав

И) Платный участок трассы протянулся с километра **P1** до километра **P2** (**P1**<**P2**). А пост ГАИ размещен на километре **M**. Попадает ли этот пост на платный участок трассы? Пусть ваша программа разберется с этим.

К) Дорожная служба запланировала ремонт трассы на участке с **R1** по **R2** (**R1**<**R2**). В сочетании с условием предыдущей задачи ваша программа должна определить:

- Будут ли ремонтировать весь платный участок **P1-P2** ?
- Будут ли ремонтировать хотя бы часть платного участка **P1-P2** ? Если да, то определить длину ремонтируемой платной части.
- Будут ли ремонтировать хотя бы часть бесплатного участка? Если да, то определить длину ремонтируемой бесплатной части.

Глава 18
Аз, Буки

Вот вам новая задача: побуквенная распечатка строки. Программа должна запросить строку и напечатать ее по буквам, например:

```
Введите строку: PASCAL
P
A
S
C
A
L
```

Да будь я хоть семи пядей во лбу, спасовал бы перед этой задачей, если бы… если бы не знал внутреннего устройства строки.

Символьный тип данных

Строковые данные, которыми мы так запросто орудуем, не так уж просты, и нам следует разобраться в этом. Вспомните первый класс, с чего всё началось? С освоения букв. Строки тоже складываются из букв, точнее из символов. Символы — это не только буквы, но и цифры, знаки препинания, и даже пробел. Существуют и невидимые, так называемые управляющие символы, но о них мы поговорим в другой раз. Рассмотрим следующую строковую константу:

```
      'Привет, Мартышка!'
```

Сколько символов в этой строке? Здесь 14 букв, к ним надо прибавить запятую, восклицательный знак и пробел, и тогда получится 17.

Для представления отдельных символов в Паскале имеется тип данных **CHAR** — от английского CHARACTER, что значит «символ». Так же, как и строковые, символьные данные могут быть константами и переменными. Переменные символьного типа объявляют так.

```
var c1, c2, c3 : char;
```

Тут объявлены три переменные, которым можно присваивать значения символьных констант или других символьных переменных, например:

```
      c1:= 'A';  c2:= 'B';  c3:= c1;
```

Символьные константы, как и строковые, заключают в апострофы. Но, в отличие от строк, они могут содержать ровно ОДИН символ, — не больше и не

меньше! Для острастки я покажу ошибочные операторы, компилятор их обязательно забракует.

```
      c1:='ABBA';       { нельзя присвоить более одного символа }
      c2:='';           { и менее одного тоже! }
```

Но строковым переменным разрешено присваивать значения символьных данных, например:

```
var   c1 : char;  S: string;
. . .
      S:= c1;
```

Это и понятно, ведь строка может вмещать много символов! Строковые и символьные данные можно «склеивать» операцией сложения, результат получится строковым, например:

```
      c1:= 'A';  c2:= 'B';  c3:= 'A';
      S:= c1 + c2 + 'B' + c3;       { результат равен 'ABBA' }
      S:= 'pascal'+ c1 + S;         { «склеивание» символов и строк }
```

Подобно строкам, отдельные символы вводятся процедурой **Readln**, и печатаются процедурами **Write** и **Writeln**, например:

```
      Readln(c1);
      Writeln(c1);
```

Индексация

Ясно, что «склеить» символы в строку немудрено, но ведь для решения поставленной задачи требуется обратное — разобрать строку на отдельные символы. Взглянем на строку с иной стороны — как на стройный ряд символов. Каждый символ в этом строю, подобно солдатам, занимает свою позицию. Позиции нумеруются слева направо, начиная с единицы. Например, в слове «PASCAL» символ «P» занимает первую позицию, а «L» — шестую.

Оказывается, что по этим номерам можно обращаться к отдельным символам строки, применяя операцию индексации. Она записывается с помощью пары квадратных скобок, расположенных за символьной переменной или константой. Внутрь скобок помещают числовое выражение, указывающее позицию символа в строке. Например, для извлечения 3-го символа строки можно написать

```
c1:= S[3];
```

Выражение, что внутри квадратных скобок, называется индексом. Повторяю, индексом может быть не только число, но и числовое выражение, например:

```
c1:= S[2*N+1];
```

Если **N** равно двум, то в символьную переменную **c1** будет помещен пятый символ строки **S**.

Длина строки

Разумеется, что значение индекса не должно превышать количество символов в строке. Но как избежать таких ошибок? Если строка перед глазами, вы посчитаете символы, тыча в строку пальчиком. А если это строковая переменная?

В Паскале есть функция, определяющая количество символов в строке, или, иначе говоря, длину строки. Эта функция так и называется — **Length** — «длина». Вызвать её можно, например, так:

```
K:= Length(S);
```

Здесь переменной **K** целого типа присваивается значение длины строковой переменной **S**. Вот ещё примеры (в комментариях указаны результаты).

```
S:= '';     K:= Length(S);              { K=0 }
S:= 'PAS'   K:= Length(S);              { K=3 }
K:= Length(S+'CAL');                    { K=6 }
K:= Length('Привет, Мартышка!');        { K=17 }
```

Распечатка строки

Теперь мы достаточно подкованы, чтобы решить поставленную задачу — разбить строку на отдельные символы. Вот как выглядит один из вариантов решения.

```
{ P_18_1 – распечатка отдельных символов строки }
var  S: string;
      C: char;
      k, L : integer;
begin
      repeat
            Write('Введите строку: ');  Readln(S);
            L:= Length(S);  { определяем длину строки }
            for k:=1 to L do begin
                  C:= S[k];   { выбираем очередной символ }
                  Writeln(C); { и печатаем его в отдельной строке }
            end;
      until L=0;        { L=0, если строка пуста }
end.
```

После ввода запрошенной строки определяем её длину, а затем, пробегая по строке, выбираем и печатаем символы. Программа работает, пока пользователь не введет пустую строку; тогда длина строки **L** станет равной нулю, и цикл завершится.

В этом варианте программы я сознательно допустил некоторые излишества, дабы наглядней показать механизм доступа к символам строки. То же самое можно записать короче, а именно:

```
{ P_18_2 – распечатка отдельных символов строки, краткий вариант }
var  S: string;  k : integer;
begin
      repeat
            Write('Введите строку: ');  Readln(S);
            for k:=1 to Length(S) do Writeln(S[k]);
      until Length(S)=0;
end.
```

Здесь функция **Length** вставлена в оператор **FOR**, а параметром процедуры **Writeln** является текущий символ строки **S[k]**. В цикле **FOR** выполняется теперь лишь один оператор, поэтому отпала нужда в блоке **BEGIN-END**. Обратите внимание на условие завершения цикла **UNTIL**, — оно записано с применением функции **Length**.

На этом прервем изучение символов и строк. Однако тема не исчерпана, и к ней мы ещё вернемся.

Итоги

- Строки – это цепочки символов. Для работы с отдельными символами в Паскале предусмотрен тип данных **CHAR**.
- Данные типа **CHAR** можно «склеивать» друг с другом и со строковыми данными, в результате получаются строки.
- Доступ к отдельным символам строки возможен путем индексации. Эта операция обозначается парой квадратных скобок, следующих за строкой; внутрь скобок помещают числовое выражение – индекс.
- Доступ по индексу применяется как для чтения символов строки, так и для их изменения.
- Для обработки строки необходимо знать её длину. С этой целью в Паскале применяется функция **Length**.
- Для последовательной обработки символов строки обычно используют цикл со счетчиком **FOR-TO-DO**.

А слабо?

А) Напишите программу для подсчета букв «А» во введенной пользователем строке. Или слабо?

Б) Напишите программу, меняющую символы «А» строки на символы «Б». Подсказка: изменение символа строки делается оператором присваивания вида **S[i]:=**…

В) Что делают со строкой **S** следующие операторы?

```
for i:=1 to Length(S) do S:= S + S[i];
for i:=Length(S) downto 1 do S:= S + S[i];
```

Проверьте свои предположения на практике.

Г) Записи телефонных номеров обычно содержат дополнительные символы: скобки, черточки, пробелы, например: **8(123)45-67-89**. Предположим, что пользователь их так и вводит. Пусть ваша программа удалит из такой строки все символы, кроме цифр. Например, после ввода указанного выше номера она должна напечатать: **8123456789**.

Д) Пусть ваша программа напечатает введенную пользователем строку вразрядку, добавляя подчёркивание либо пробел после каждого символа, например: **'Pascal'** преобразует в **'P_a_s_c_a_l'**.

Глава 19
Процедуры и функции: разделяй и властвуй

Снежный ком

Чем дальше в лес, тем больше дров, — наши программы становятся всё замысловатей! Чем измеряют сложность программ? — усилиями, что потребны на их осмысление. С ростом размера программы её сложность растет снежным комом: так программа в десять страниц стократ сложней одностраничной! Почему?

Три базовые структуры: линейная последовательность, условный переход и цикл — это строительные блоки наших изделий. В ходе постройки программы эти структуры причудливым образом внедряются друг в друга: условные — внутрь циклов, циклы — внутрь условных операторов и так далее. План «постройки» определяется решаемой задачей — алгоритмом — и тут ничего не упростить. С ростом программы не только запутывается её текст, но и плодятся полчища переменных. «Расползаясь» по телу программы, они затрудняют контроль над собой. Поверьте, продолжая «строительство» в прежнем стиле, вы скоро свихнетесь, — ведь серьезные программы насчитывают тысячи страниц!

В 15-й главе я поведал о соединении усилий программистов в работе над одним проектом, — им на выручку приходят процедуры и функции. Мы уже пользовались ими, извлекая готовенькими откуда-то из «недр» Паскаля (такими, как **Writeln**, **Readln**, **Length**, **Random**). Заботит ли вас устройство и сложность этих процедур? Нет? То-то же! Подобно усердным слугам, они лишь исполняют наши капризы. Но, то — чужие «слуги», созданные другими программистами, не пора ли обзавестись своими? Разбив сложную программу на «кусочки», мы значительно упростим её. Как говорят, разделяй и властвуй!

Описание процедур

Для постройки нашей первой процедуры возьмем знакомый пример. Вот как организована пауза с ожиданием нажатия клавиши Enter в одной из наших первых программ.

```
    Write('Нажмите Enter…');     Readln;
```

Пустяшный кусочек, — всего два оператора. Но их можно заменить одним, если создать процедуру, выполняющую те же самые действия.

Создать процедуру, — это значит дать ей имя и описание. Делается это с применением ключевого слова **PROCEDURE**, после которого указывается имя процедуры и её тело, содержащее операторы. Имя процедуры назначают произвольно по тем же правилам, что для констант и переменных. Сейчас мы заменим пару упомянутых выше операторов процедурой по имени **Pause** (пауза), вот как будет выглядеть её описание (рис. 40).

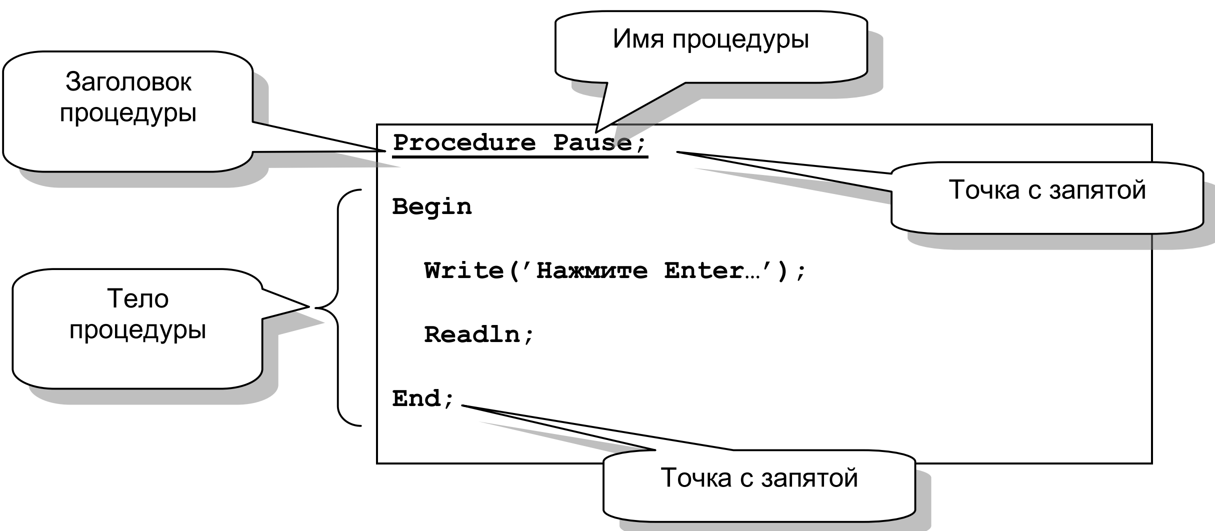


Рис. 40 – Описание процедуры Pause

После заголовка процедуры ставится точка с запятой. Далее следует тело, заключенное в блок **BEGIN-END**. Завершается описание процедуры ещё одной точкой с запятой. В блоке **BEGIN-END** размещают любое количество исполняемых операторов по тем же правилам, что применялись нами ранее. Обратите внимание: блок **BEGIN-END** в теле процедуры обязателен! Даже если внутри блока будет всего один оператор, или не будет вовсе!

Теперь решим, где расположить это хозяйство? На рис. 41 показана знакомая вам структура простой программы. После объявления констант и переменных следует главная программа, где исполняемые операторы заключены между **BEGIN** и **END**.

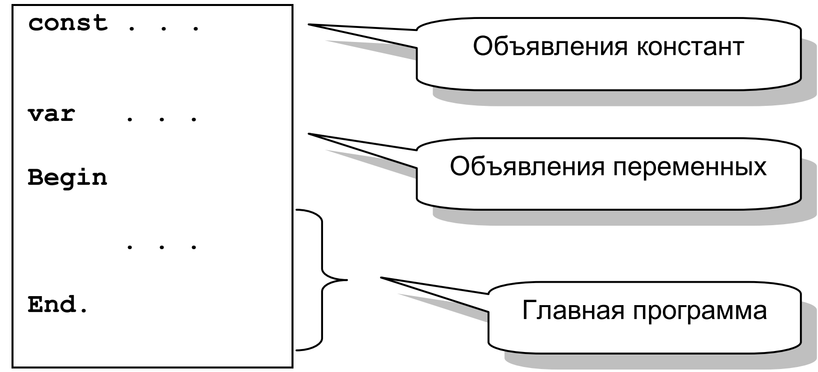


Рис. 41 – Структура простой программы

По правилам языка любой объект программы — константа, переменная, процедура — должен объявляться до своего использования. Стало быть, описание процедуры надо поместить до того, как будет сделан её вызов. Поскольку процедура **Pause** вызывается из главной программы, её описание должно быть помещено перед нею.

```
{ P_19_1 – Пример применения процедуры }
var Man : string;

procedure Pause; {--- описание процедуры ---}
begin
      Write('Нажмите Enter…');
      Readln;
end;

begin       {--- главная программа ---}
      Writeln('Как тебя зовут?'); Readln(Man);
      Writeln('Здравствуй, ', Man);
      Pause;      { вызов процедуры }
end.
```

Но в каком порядке будут выполняться операторы этой программы? Мы знаем, что компьютер исполняет программу, как бы читая её слева направо и сверху вниз. Стало быть, операторы в теле процедуры выполняются первыми?

А вот и нет! Главная программа на то и главная, чтобы исполняться первой. Всё начнется с запроса имени пользователя и так далее. Когда же дело дойдет до вызова процедуры **Pause**, вступят в бой операторы в теле этой процедуры. Последовательность исполнения показана на рис. 42 (обратите внимание на нумерацию строк). Вызов процедуры **Pause** приведет, как говорят программисты, к передаче управления внутрь тела процедуры. После исполнения расположенных там операторов, управление возвращается в главную программу к оператору, следующему за вызовом.

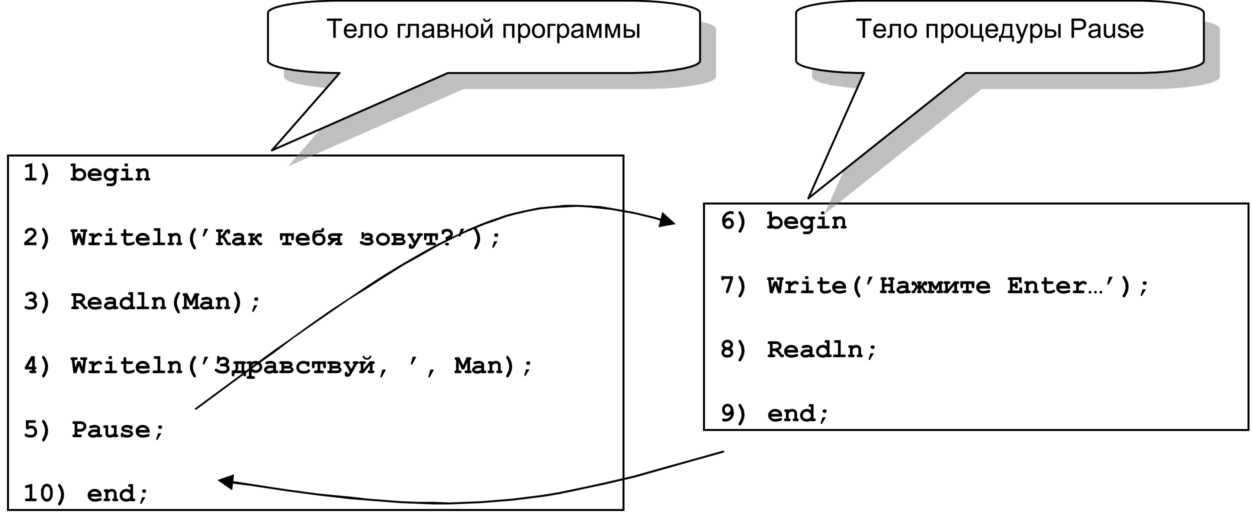


Рис. 42 – Последовательность выполнения операторов

Итак, хотя процедура размещается в тексте выше главной программы, её операторы выполняются позже — после вызова процедуры.

При необходимости вызов процедуры можно повторить. Например, чтобы подразнить пользователя и заставить его трижды нажать клавишу Enter, сделаем так.

```
begin       {--- главная программа ---}
      Writeln('Как тебя зовут?'); Readln(Man);
      Writeln('Здравствуй, ', Man);
      Pause;
      Pause;
      Pause;
end.
```

Процедуры с параметрами

Вам ясна техника объявления и вызова процедур? Тогда рассмотрим ещё один случай: пусть выводимое процедурой сообщение меняется по нашему желанию. Для этого процедуру снабжают параметром. Вы знаете, что параметр указывается в скобках за именем процедуры, например:

```
      Pause ('Будьте любезны нажать Enter!');
```

Попробуйте вызвать процедуру этим способом, что вам скажет компилятор? Ничего хорошего не скажет и будет прав. Откуда процедуре знать о вашем желании вывести именно это сообщение? Но если добавить в заголовок процедуры объявление параметра, дело пойдет на лад. Объявление параметра тоже помещают в скобки; оно похоже на объявление переменной. В нашем случае заголовок процедуры с параметром может выглядеть, например, так:

```
procedure Pause (msg : string);
```

Здесь имя параметра **msg** назначено нами произвольно (это сокращение от слова message — «сообщение»). Параметр, объявленный в заголовке, называют формальным, он доступен только внутри процедуры, где можно обращаться с ним, как с обычной переменной. Например, вывести на экран, как в нашем случае.

```
procedure Pause (msg : string);     { объявление процедуры с параметром }
begin
      Write(msg);  Readln;
end;
```

Что касается вызывающей программы, то имя формального параметра ей неизвестно.

Как действует такая процедура? В момент вызова в главной программе формальному параметру msg автоматически присваивается указанное в вызове фактическое значение, — оно и будет напечатано. Повторяю: присвоение формальному параметру фактического значения происходит автоматически, без участия программиста. Теперь наша программа станет такой.

```
{ P_19_2 - применение процедуры с параметром }

var Man : string;
{--- объявление процедуры с параметром msg ---}
procedure Pause (msg : string);
begin
      Write(msg);  Readln;
end;

begin            {--- главная программа ---}
      Writeln('Как тебя зовут?'); Readln(Man);
      Writeln('Здравствуй, ', Man);
      Pause('Нажмите Enter…');
      Pause('Ещё раз…');
      Pause('И ещё разок!');
end.
```

Здесь процедура **Pause** вызвана трижды с тремя разными фактическими параметрами, испытайте эту программу.

Итоги

- С ростом размера программы стремительно растет её сложность. Для упрощения программ их разбивают на процедуры и функции.
- Чтобы создать процедуру или функцию, необходимо поместить в программе её описание, состоящее из заголовка и тела.
- Внутрь процедуры или функции можно передать один или несколько параметров. Для этого в заголовке процедуры объявляют формальные параметры, а при вызове указывают фактические.
- Тип фактического параметра должен совпадать с типом формального параметра, объявленного в процедуре.

А слабо?

А) Напишите ещё одну версию процедуры **Pause**, выводящую сообщение либо на русском, либо на английском языке. Параметр этой процедуры должен быть булевым и работать она должна так.

```
Pause(true);      { печатается «Нажмите Enter…» }
Pause(false);     { печатается «Press Enter…» }
```

Б) Напишите и испытайте процедуру (назовем её **Line** — «линия»), печатающую строку заданной длины, составленную из звездочек, например:

```
Line(3);    { печатает «***» }
Line(7);    { печатает «*******» }
```

Подсказка: внутри процедуры надо организовать цикл.

В) Напишите процедуру для очистки экрана, она может пригодиться вам в будущем. Подсказка: можно напечатать несколько десятков пустых строк (не менее 25, что зависит от настройки размера консольного окна).

Г) Напишите и испытайте процедуру, принимающую два параметра — числа, и печатающую их сумму и их разность.

Задачи на темы предыдущих глав

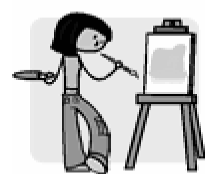
Д) Пользователь вводит строку с телефонным номером (только цифры), количество цифр заранее неизвестно. Ваша программа должна дополнить номер дефисами, разбивающими его на триады, т.е. по три цифры двумя способами:

- начиная с первых цифр, например 112-345-1;
- начиная с последних цифр, например 1-123-451.

Е) Почтальон разносит газеты по улице, состоящей из **N** домов. Четные и нечетные номера расположены по разные стороны улицы. В здравом уме почтальон не рискует лишний раз переходить её. Ваша программа должна напечатать последовательность номеров, по которым будут разнесена почта, когда почтальон начинает работу:

- с первого дома;
- со второго дома;
- с N-го (то есть последнего) дома.

Глава 20
Процедуры: первый опыт



Некоторые считают программирование искусством. Если так, то в чем оно? Искусный программист умеет (кроме прочего) превращать сложную программу в простую, — он равномерно распределяет сложность между процедурами и функциями. Как научиться этому? Усвойте несколько ключевых истин, но главное здесь — практика. Без «шишек» и «синяков» не обойтись. Однако сколько за одного битого небитых дают?

Следующая задача слегка надумана — это всего лишь полигон для испытания наших собственных процедур. Условие задачи таково: пусть пользователь введет одну за другой несколько строк, например, три (потребуется цикл со счетчиком, улавливаете?). В каждой введенной строке надо заменить латинские буквы «A» — если они там есть — на латинские буквы «B». Например, приняв строку «ABBA», программа должна превратить её в строку «BBBB».

Мухи – налево, котлеты – направо!

Рис. 43 избавляет вас от необходимости малевать алгоритм будущей программы. Ясно, что программа не так проста, — она включает условный оператор и два цикла, причем один из них вложен в другой. Внешний цикл отвечает за ввод строк, а внутренний — за их обработку. Можно ли упростить это сооружение? Бывалый программист сразу смекнет, как отделить здесь мух от котлет, — внутренний цикл, отмеченный серым цветом, лучше выделить в отдельную процедуру, и тогда программа распадётся на два несложных алгоритма (рис. 44). Слева на этом рисунке показан алгоритм главной программы, а справа — алгоритм процедуры, которой я дал имя **`Scan`**. Пунктирные линии со стрелками показывают места входа в процедуру и выхода из нее.

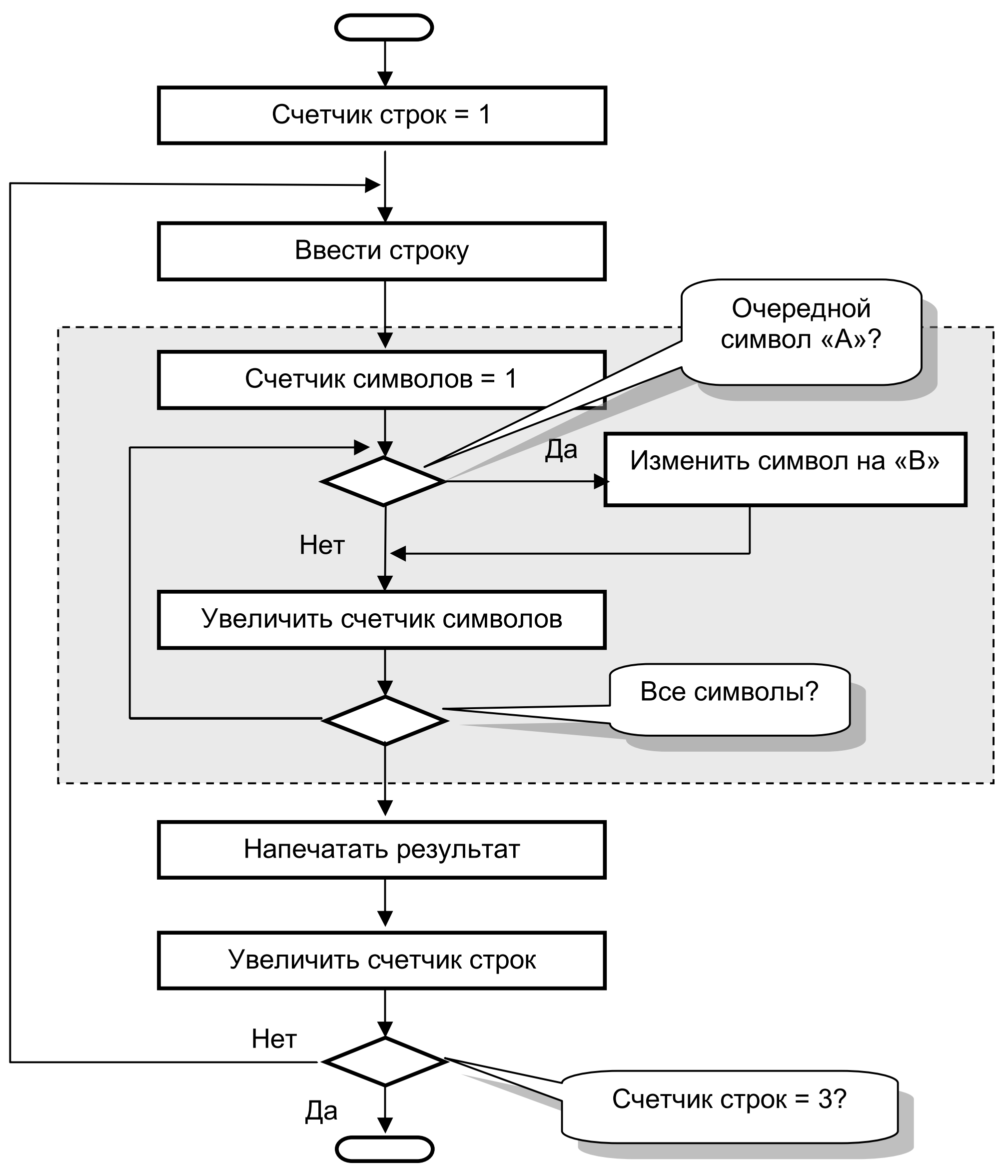


Рис. 43 – Блок-схема программы с двумя циклами

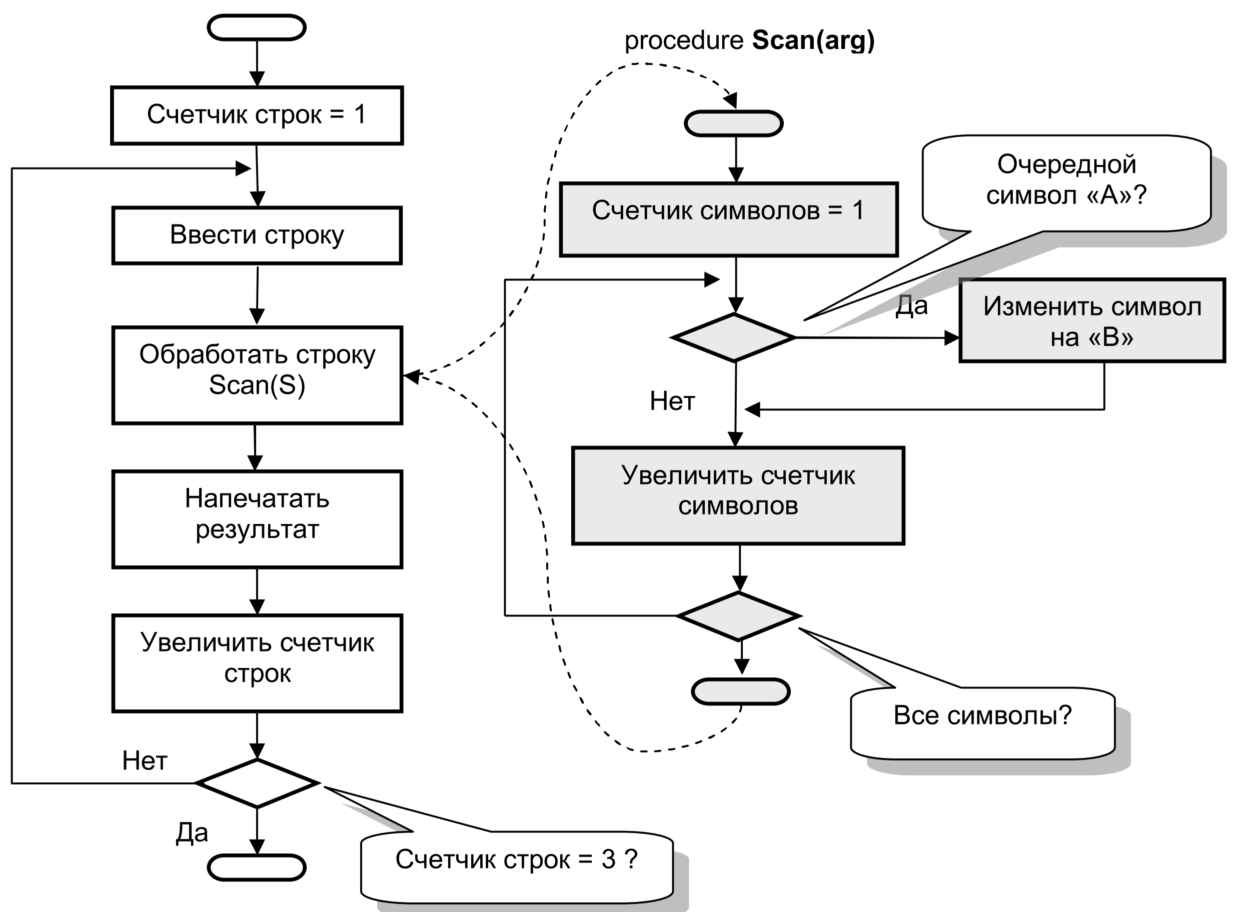


Рис. 44 – Блок-схемы главной программы (слева) и процедуры (справа)

Согласитесь, — каждая из этих блок-схем в отдельности не так уж сложна, значит можно приступить к написанию программы.

Сверху вниз

Легко сказать «приступить», но с чего начать? Настрочить программу целиком и сразу? — вот прекрасный способ запутаться! Нет, профессионалы делают иначе, следуя одному из двух направлений. Первое из них именуется разработкой «сверху вниз», — проект лепят, начиная с главной программы, переходя затем к процедурам. Другое направление противоположно первому и называется разработкой «снизу вверх». Оба направления имеют свои достоинства, поэтому в крупных проектах их иногда используют одновременно. Но сейчас нам лучше подходит первый способ.

Итак, последуем выбранному нами порядку разработки «сверху вниз». Этот подход хорош тем, что на промежуточных этапах получаются почти работающие программы. Почему «почти»? — сейчас поймете. Итак, забудем на время о недостающей процедуре `Scan` и напишем лишь главную программу, вот она.

```
{ P_20_1 – первый этап разработки }
var S: string; k: integer;
begin       {--- главная программа ---}
      for k:=1 to 3 do begin
            Write('Введите строку: '); Readln(S);
            { Scan(S); }
            Writeln(S);
      end;
end.
```

Обратите внимание на закомментированный вызов процедуры **Scan(S)**, — он напоминает о незавершенной части работы. Скелет нашей будущей программы готов, его можно не только скомпилировать, но и запустить, — сделайте это обязательно! Разумеется, программа не выполняет всего задуманного, но уже делает кое-что.

Убедившись в работоспособности скелета, перенесём внимание на процедуру. На этом этапе тоже есть свои хитрости: сначала дадим частичное описание процедуры, создав лишь заголовок и оставив тело пустым. Такую процедуру называют заглушкой или пустышкой. Написав заглушку, уберите комментарий с вызова **Scan(S)**, и тогда на скелете нарастет немного «мяса».

```
{ P_20_1 – второй этап разработки }
var S: string;    k: integer;

      {--- Заглушка процедуры ---}
procedure Scan(arg : string);
begin
end;

begin       {--- главная программа ---}
      for k:=1 to 3 do begin
            Write('Введите строку: '); Readln(S);
            Scan(S);
            Writeln(S);
      end;
end.
```

Процедура **Scan** принимает строковый параметр **arg** (это сокращение от слова argument). Аргумент — так ещё называют параметр процедуры или функции. Теперь снова запустите программу. Если всё в порядке, значит вызов процедуры **Scan(S)**, как говорят программисты, видит описание этой процедуры, и его фактический параметр **S** отвечает формальному параметру процедуры **arg**.

Переходим к третьему этапу, где можно забыть о главной программе и сосредоточиться на теле процедуры **Scan**. Напомню, что ей поручено заменить в строке буквы «A» на буквы «B». С этой несложной работой справится цикл, содержащий вложенный в него условный оператор.

```
      for k:=1 to Length(arg) do
            if arg[k]='A' then arg[k]:='B';
```

Напомню, что **arg** — это переданная в процедуру строка, а **k** — счетчик цикла. Вставив этот цикл в тело процедуры **Scan**, получим готовенькую программу.

```
{ P_20_1 –  третий этап разработки }
var S: string;    k: integer;

procedure Scan(arg : string);
begin
      for k:=1 to Length(arg) do
            if arg[k]='A' then arg[k]:='B';
end;

begin       {--- главная программа ---}
      for k:=1 to 3 do begin
            Write('Введите строку: '); Readln(S);
            Scan(S);
            Writeln(S);
      end;
end.
```

Обратите внимание на счетчик циклов **k**. Он — счетчик — используется нами в двух местах: в главной программе и в процедуре. Налицо экономия памяти, не так ли? Насколько оправдана эта надежда? Скоро узнаем.

Для пишущих на Delphi. Компилятор Delphi не позволит использовать счетчик k так, как сделано в этой программе, — но об этом чуть позже.

Первые раны

Теперь запустите наше творение. Если вам это удалось, значит компилятор не нашёл ошибок. Но вот незадача, — работает программа неправильно! Во-первых, буква «A» не меняется на букву «B». Ещё печальней то, что перестал работать цикл в главной программе. Она, что называется, зациклилась, запрашивая непрестанно всё новые и новые строки. А ведь на скелете цикл работал, — мы проверяли!

Впрочем, если ввести строку из трех символов, программа чудесным образом завершится. Это наводит на размышление, — ведь цикл главной программы тоже считает до трех. Не промахнулись ли мы, доверив переменной **k** «служить двум господам», работая в двух циклах? Ведь внутри процедуры значение счетчика **k** изменяется, что нарушает работу цикла в главной программе. И лишь когда счетчик случайно станет равен трём, программа завершается.

Как исправить ошибку? Объявить для счетчика внутреннего цикла переменную с другим именем? Да, можно. Но я воспользуюсь этой ошибкой, чтобы показать иной подход и лучше раскрыть механизм процедур и функций.

Глобальные и локальные

Процедуры и функции не зря называют подпрограммами. Так же, как в главной программе, внутри подпрограмм можно объявлять свои собственные константы и переменные, — их называют локальными, то есть местными. А всё, что объявлено за пределами подпрограмм, называют глобальным или всеобщим. Рассмотрим механизм действия локальных объектов и связанные с этим выгоды, для чего исследуем следующую программу.

```
const c1 = 'Глобальная';

procedure Local;
begin
      Writeln(c1);
end;

begin       {--- главная программа ---}
      Local;
      Writeln(c1);
      Readln;
end.
```

Очевидно, программа дважды напечатает константу **C1**, — проверьте меня. Теперь добавим объявление локальной константы с тем же именем **C1**, но поместив его между заголовком процедуры **Local** и её телом. К совпадающим именам я прибегнул не от бедности фантазии, — мой умысел скоро прояснится.

```
const c1 = 'Глобальная';

procedure Local;
const c1 = 'Локальная';
begin
      Writeln(c1);
end;

begin       {--- главная программа ---}
      Local;
      Writeln(c1);
      Readln;
end.
```

Известно, что компилятор не допускает совпадающих имен, но здесь — иное дело. Локальная константа `C1` «спряталась» внутри своей процедуры и, как говорят программисты, не видна за её пределами.

Запустив на выполнение этот вариант программы, вы убедитесь, что внутри процедуры будет напечатана локальная константа, а в главной программе — глобальная. Отсюда следуют два правила, имеющих силу и для констант, и для переменных, и для других объектов, о которых вы со временем узнаете. Правила эти таковы:

- локальные объекты (константы, переменные и прочие) видны лишь внутри тех подпрограмм, в которых они объявлены;
- при совпадении имен локального и глобального объектов, внутри подпрограммы имеет силу локальный объект; при этом глобальный объект скрывается локальным.

С учетом сказанного нашу неработающую программу исправим так.

```
{ P_20_1 - вариант программы с локальной переменной }
var S: string;    k: integer; { глобальная переменная }

procedure Scan(arg : string);
var k: integer; { локальная переменная }
begin
      for k:=1 to Length(arg) do
            if arg[k]='A' then arg[k]:='B';
end;
```

```
begin { главная программа }
      for k:=1 to 3 do begin
            Write('Введите строку: '); Readln(S);
            Scan(S);
            Writeln(S);
      end;
end.
```

Теперь совпадение имен локальной и глобальной переменных **k** не нарушает работу программы, поскольку это разные переменные. Они могли бы иметь даже разные типы! Убедитесь, что отныне путаницы в циклах нет.

Локально – это разумно!

Локальные объекты — константы, переменные и прочие — отличное средство разумного распределения данных в пространстве вашей программы.

Во-первых, они облегчают многотрудную жизнь программиста. Если какие-то объекты нужны только внутри процедуры, их следует объявить там же, то есть как локальные. И тогда не придётся гадать, повлияют ли они на другие части программы.

Другой выигрыш заключается в экономии памяти. Все переменные занимают оперативную память (оперативку). Чем больше переменных, тем больше памяти им подавай. Глобальные переменные занимают память в течение всего времени работы программы. А для локальных память выделяется лишь на время работы соответствующей процедуры или функции. Завершилась подпрограмма — освободилась память.

Неподдающаяся строка

Теперь вновь проверим нашу программу. В ответ на запрос строки введите что-нибудь вроде «QAAAW». Если всё нормально, программа напечатает «QBBBW» (буква «A» заменяется буквой «B»). Не вышло? Что ж, тогда идем «на поклон» к отладчику, — мы сделаем это в следующей главе.

Итоги

- Программа упрощается, если вынести части алгоритма в отдельные подпрограммы — процедуры и функции.
- Объекты, используемые лишь внутри подпрограмм, следует объявлять там же — как локальные.
- Локальные объекты (константы, переменные и прочие) доступны лишь внутри тех подпрограмм, где они объявлены.

- Если имя локального объекта совпадает с глобальным, то внутри подпрограммы действует локальный объект, а глобальный становится «невидимкой».
- Локальные объекты упрощают программирование, придают программам надежность и экономят оперативную память.

А слабо?

А) В 17-й главе нами создан экзаменатор, проверяющий знания таблицы умножения. Переработайте программу P_17_1 так, чтобы оценка выставлялась в процедуре, принимающей один параметр — количество допущенных ошибок.

Б) Создайте процедуру, печатающую все числа, кроме единицы, на которые без остатка делится число **N**, где **N** — параметр процедуры. Напишите программу для проверки этой процедуры.

В) Два сотрудника подали своему начальнику заявления на отпуск. Первый попросил отпустить его с **A1** по **B1** день (дни отсчитываются с начала года), второй — с **A2** по **B2** день (полагаем, что **A1<B1** и **A2<B2**). Однако дело требует, чтобы кто-то из сотрудников находился на рабочем месте. Мало того, при смене отдыхающих необходимо не менее 3-х дней их совместной работы — для передачи дел. Напишите программу с процедурой, принимающей четыре указанных выше параметра, и печатающей заключение о том, удовлетворяют ли заявления работников требованиям начальника.

Г) Подойдя к перекрестку, пешеход решает, переходить ли ему улицу или остановиться. На решение влияет характер пешехода и ещё два фактора: сигнал светофора и близость опасно движущегося транспорта. Напишите программу с процедурой, которая принимает и печатает решение в зависимости от переданных в неё трех параметров, а именно.

- Параметр **A = true**, если горит зеленый;
- Параметр **B = true**, если поблизости опасно движется транспорт;
- Параметр **C** – это число, определяющее характер пешехода так:

1 - послушный и осторожный — учитывает и светофор и опасность;
2 - послушный, но беспечный — смотрит только на светофор;
3 - хитрый вольнодумец — идет только на красный, если это ничем не грозит;
4 - непримиримый вольнодумец — идет только на красный;
5 - экстремал — идет только на красный, и так, чтобы грозила опасность;
6 - «безбашенный» — идет, несмотря ни на что;
7 - запуганный — никогда не идет через дорогу, а ищет подземный переход.

Глава 21
Отладка

Предыдущую главу мы покинули, понурив голову, так и не совладав с программой P_20_1. Почему не заменяются символы в строке? — этот вопрос остался без ответа. Эх, знать бы, что творится внутри программы! Сейчас она для нас — загадочный «черный ящик», и мы видим лишь то, что входит и выходит из него. К счастью, в IDE есть средство для доступа внутрь этого «ящика», и мы воспользуемся им. Это средство называется отладчиком. Так же, как редактор текста и компилятор, отладчик встроен в интегрированную среду разработки.

Отладчик

Отладчик — это набор инструментов для исследования «потрохов» программы. Посредством отладчика можно следить за выполнением отдельных операторов, делая остановки в нужных местах или на каждой строке программы. Застопорив программу, вы сможете выяснить значения тех или иных переменных и даже изменить их. Одним словом, отладчик — это чудо-оружие!

Инструменты отладчика доступны через два пункта меню: *Run* — запуск и *Debug* — удаление багов (жучков). Программные ошибки прозвали багами — «жучками».

В пункте *Run* собраны команды для управления ходом выполнения программы (рис. 45).

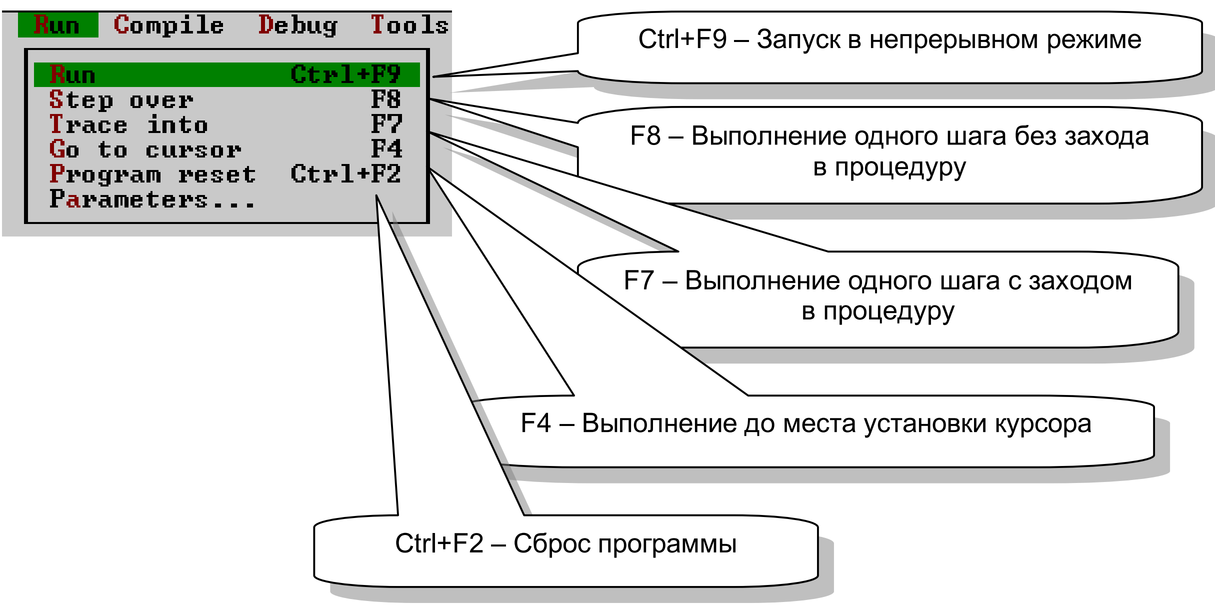


Рис. 45 – Пункты меню *RUN* для доступа к отладчику

Примечание. В данной главе показаны окна отладчика для Borland Pascal, в IDE Free Pascal они выглядят чуть иначе.

В табл. 1 даны пояснения к пунктам этого меню.

Табл. 1 – Описание пунктов меню *Run*

Команда	Горячая клавиша	Пояснение
Run	Ctrl+F9	Запускает программу в непрерывном режиме.
Trace into	F7	Выполняет одну строку программы (шаг). Если в строке есть вызов процедуры, то останов происходит на входе в нее, — так можно «войти» внутрь процедуры и следить за ходом её выполнения.
Step over	F8	Выполняет одну строку программы. Если в строке есть вызов процедуры, то процедура выполняется целиком, без остановки.
Go to cursor	F4	Выполняет программу, пока не будет достигнута строка, где установлен текстовый курсор. Курсор надо предварительно установить на нужной строке!
Program Reset	Ctrl+F2	Сброс программы. Если программа остановлена в пошаговом режиме, она перейдет в исходное состояние.
Parameters...	нет	Используется для отладки программ, принимающих параметры через командную строку.

Обратите внимание: за один шаг отладки выполняется либо одна строка программы, либо один оператор, если он занимает несколько строк. Стало быть, операторы, помещенные в одной строке, будут выполнены за один шаг. Если компилятору безразлично, как вы располагаете операторы, но при отладке это важно. Не размещайте операторы в одной строке, если при отладке намерены выполнять их раздельно.

Теперь обратимся к пункту меню *Debug* (рис. 46), где собраны команды для просмотра переменных, их редактирования, а также для просмотра выводимых программой результатов. Эти результаты можно увидеть либо на экране (User screen) либо в специальном окне (Output). Рядом с командами показаны соответствующие им горячие комбинации клавиш.

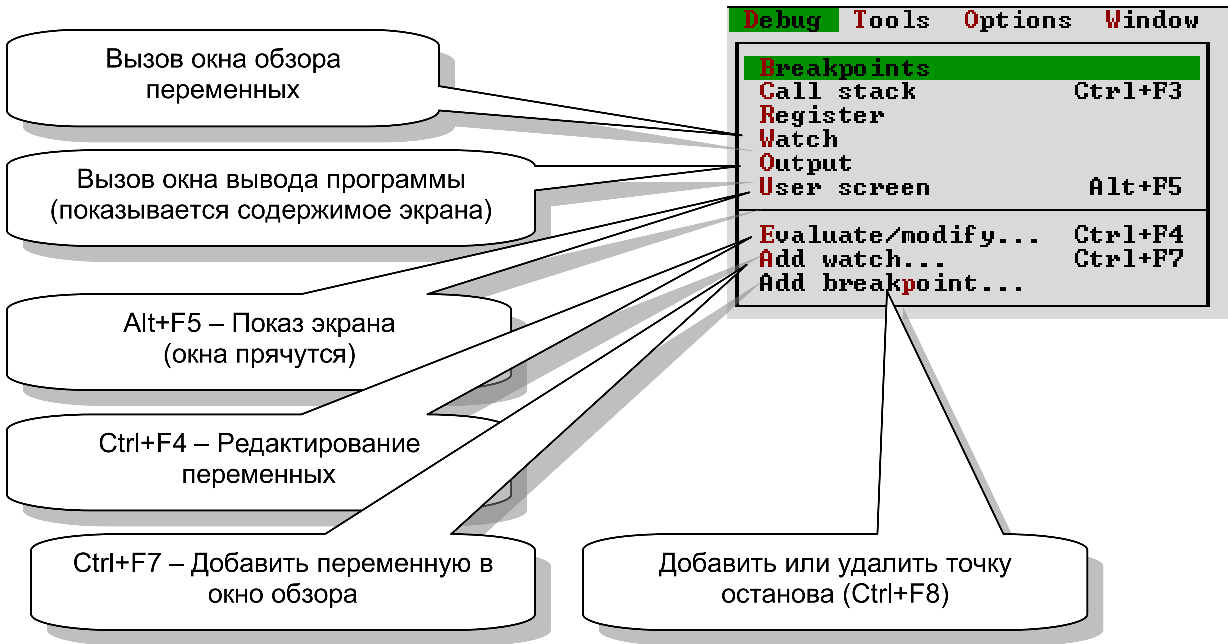


Рис. 46 – Пункт меню *Debug* для просмотра результатов работы программы

Теперь испробуем основные команды отладчика на своей программе.

Жучки, вылезайте!

Итак, приступим к поиску жучков, притаившихся в программе P_20_1. Хорошо бы проследить за изменением переменных в ходе выполнения программы. Для этого вставим переменные в окно обзора «Watches». Откомпилировав программу, поместите курсор под переменной **k** и нажмите *Ctrl+F7*. Появится диалоговое окно для добавления переменной в окно обзора (рис. 47).

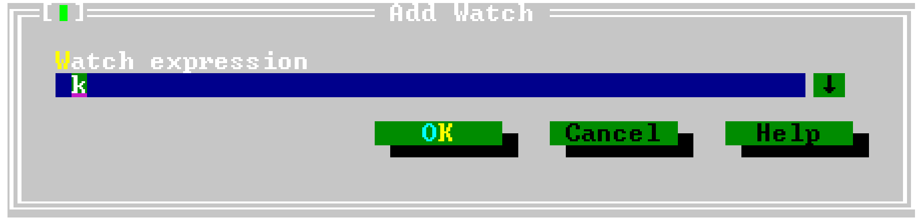


Рис. 47 – Добавление переменной в окно обзора

Поскольку переменная **k** была взята на мушку заранее, поле уже содержит её имя. Теперь щелчок по кнопке *OK* отправит переменную в окно обзора (рис. 48). Если же поле «Watch expression» пусто, или содержит нечто другое, значит, вы промахнулись, не попали курсором. Тогда впечатайте имя нужной переменной и щелкните *OK*. Действуя так, добавьте в окно обзора все интересующие вас переменные (рис. 48).

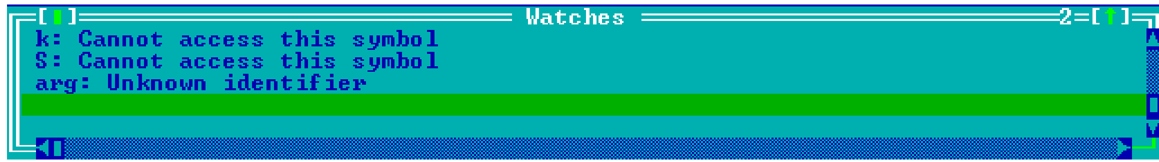


Рис. 48 – Окно обзора переменных Watches

Пока программа не запущена, напротив имен переменных выводится сообщение о невозможности доступа к ним, — пусть вас это не смущает. Лучше взгляните на то, как расположено окно «Watches». Сейчас оно занимает нижнюю часть экрана и закрывает собой часть окна с программой. Это неудобно, а посему обратитесь к пункту меню *Window* → *Tile* (Окна → Рядом) как показано на рис. 49.

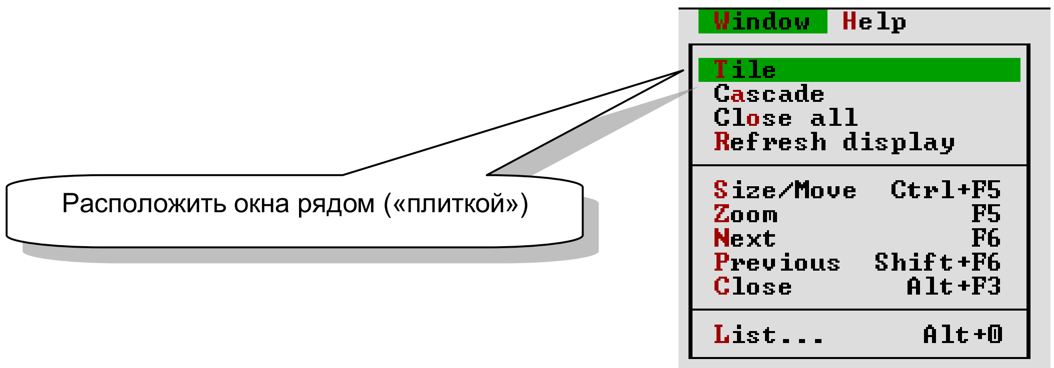


Рис. 49 – Пункт меню *Window* → *Tile*

В результате окна с текстом программы и списком переменных поместятся, не перекрывая друг друга (рис. 50). То же самое можно сделать и мышкой, перетаскивая и меняя размеры окон.

Теперь станем выполнять программу по шагам, следя за изменением переменных. Вместо привычной комбинации *Ctrl+F9*, для пуска программы в пошаговом режиме нажимают клавишу *F7* (команда *Run* → *Trace*). Тогда отладчик остановит программу перед первым оператором, подсветив его особым образом. Последующие нажатия клавиши *F7* заставят выполняться следующие строки программы, и очередная строка будет выделяться цветной полоской.

Нажав клавишу *F7* четыре раза, мы достигнем оператора **`Readln`**, — здесь программа остановится в ожидании ввода строки. Введите как обычно строку из латинских букв «QAAAW» и нажмите *Enter*, — и тогда программа остановится перед входом в процедуру **`Scan`**, как показано на рис. 50.

Примечание. В отладчике IDE Free Pascal при вводе строки необходимо нажать клавишу *Enter* дважды.

```
 File  Edit  Search  Run  Compile  Debug  Tools  Options  Window  Help
=[■]================== PROG022.PAS ======================1=[↑]=
var S: string; k: integer;

procedure Scan(arg : string);
var k: integer;
begin
     for k:=1 to Length(arg) do
         if arg[k]='A' then arg[k]:='B';
end;

begin
  for k:=1 to 3 do begin
    Write('Введите строку: '); Readln(S);
    Scan(S);
    Writeln(S);
  end;
end.
======= 1:1 =======
============================= Watches ==========================2=
 k: 1
 S: 'QAAAW'
 arg: Unknown identifier
F1 Help  F7 Trace  F8 Step  F9 Make  Alt+F10 Local menu
```

Рис. 50 – Состояние программы перед входом в процедуру Scan

В этом месте рассмотрим переменные в окне «Watches». Счетчик циклов **k** равен единице, — это глобальная переменная **k**, поскольку локальной переменной с этим же именем пока не существует. Переменная **S** содержит то, что мы ввели с клавиатуры. Параметр **arg** тоже пока не виден отладчику, о чём говорит сообщение «Unknown identifier».

Нажмите клавишу *F7* ещё пару раз, пока цветная полоска не перескочит внутрь процедуры **Scan**. Здесь параметр **arg** примет то же значение, что и глобальная переменная **S**, — это прекрасно видно в отладчике. Продолжайте нажимать клавишу *F7*, пока цветная полоска не дойдет до слова **END** в конце процедуры. Вы увидите, как параметр **arg** постепенно принимает значение «QBBBW», — это то, что нам нужно. Состояние программы в этот момент показано на рис. 51.

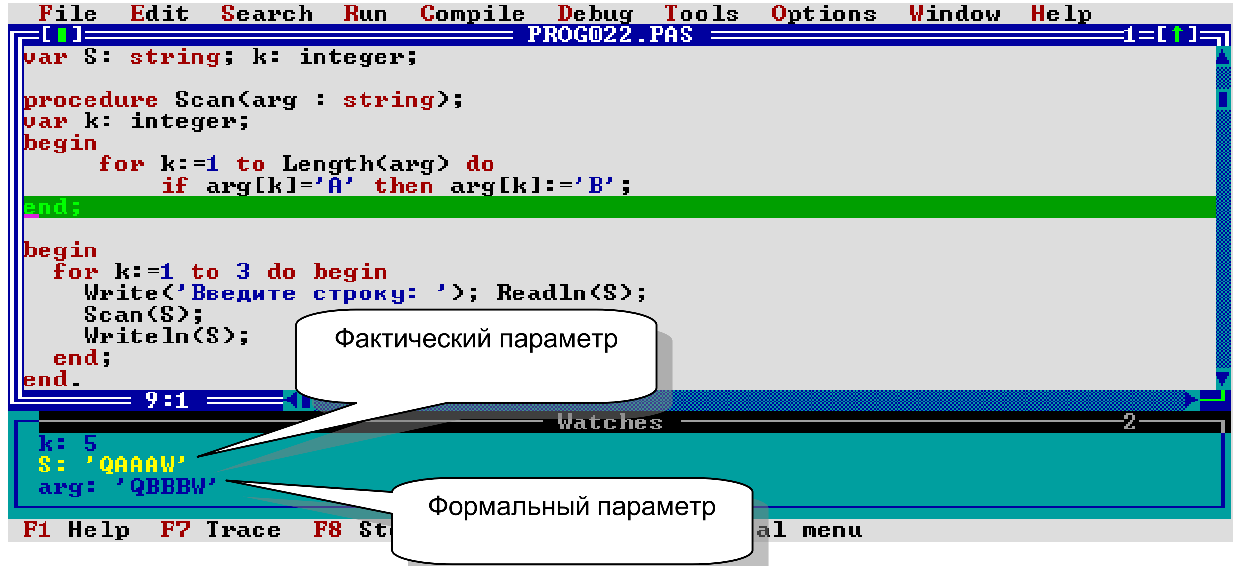


Рис. 51 – Состояние программы перед выходом из процедуры Scan

Теперь переменная **k** равна пяти, — это длина введенной строки. Но это уже другая, локальная переменная **k**, поскольку её глобальная тёзка внутри процедуры не видна. Но переменная **S** (тоже глобальная) по-прежнему видна внутри процедуры, ведь её имя не перекрывается локальной переменной.

Нажмите клавишу *F7* ещё раз, — программа выйдет из процедуры и цветная полоска перепрыгнет на строку, следующую за вызовом процедуры **Scan** (рис. 52).

```
 File  Edit  Search  Run  Compile  Debug  Tools  Options  Window  Help
                              PROG022.PAS                           1
var S: string; k: integer;

procedure Scan(arg : string);
var k: integer;
begin
     for k:=1 to Length(arg) do
         if arg[k]='A' then arg[k]:='B';
end;

begin
  for k:=1 to 3 do begin
    Write('Введите строку: '); Readln(S);
    Scan(S);
    Writeln(S);
  end;
end.
      15:1
                              Watches                               2
 k: 1
 S: 'QAAAW'
 arg: Unknown identifier
F1 Help  F7 Trace  F8 Step  F9 Make  Alt+F10 Local menu
```

Рис. 52 – Состояние программы после выхода из процедуры Scan

Итак, к чему мы пришли? Сравнив это состояние с тем, что было до входа в процедуру (рис. 50), находим, что значения переменных не изменились. Переменная **k** снова стала равна единице, и это понятно — ведь теперь это глобальная переменная. Беда в том, что не изменилась и переменная **S**, а ведь именно этого мы добивались. В чем же дело?

Причина кроется в способе передачи параметра. При вызове процедуры **Scan** фактический параметр **S** копируется в формальный параметр **arg**, и далее внутри процедуры работа введется с этой копией. Иначе говоря, данные передаются только внутрь процедуры, но не обратно. Этот способ передачи параметров называют передачей по значению. Стало быть, глобальная переменная **S** не должна была измениться! Здесь надо что-то исправлять!

Ссылка на переменную

Рассмотрим ещё раз объявление параметра в процедуре **Scan**.

```
procedure Scan(arg : string);
```

Оказывается, что при таком объявлении формальный параметр **arg** представляет собой локальную переменную. От прочих таких переменных он

отличается лишь тем, что при вызове процедуры в него автоматически копируется значение фактического параметра. Вы знаете, что локальные переменные существуют, пока выполняется процедура, а по её завершении исчезают. Потому результат обработки и не возвращается назад в вызывающую программу.

Впрочем, добиться нужного результата несложно: достаточно вставить в объявление параметра ключевое слово **VAR**.

```
procedure Scan(var arg : string);
```

Это мелкое изменение влечет важное следствие: теперь **arg** — не локальная переменная, а ссылка на другую переменную. Это значит, что в момент вызова процедуры данные не будут копироваться, но параметр **arg** на время исполнения процедуры станет дублером фактического параметра **S**. Теперь, изменяя параметр **arg**, мы тем самым будем изменять и переменную **S** — наш фактический параметр.

Передавая параметр по значению, вызывающая программа как бы говорит вызываемой процедуре: «вот тебе данные (строка, число и т.п.), сохрани их у себя внутри и делай с ними, что угодно, — их дальнейшая судьба меня не интересует».

Передавая же параметр по ссылке, вызывающая программа «говорит» иначе: «нужные тебе данные находятся в такой-то глобальной переменной, и ты вправе поступать с нею как угодно».

Обратите внимание на двоякое предназначение ключевого слова **VAR**. Во-первых, оно открывает секцию объявления переменных, а во-вторых, служит для указания ссылки на переменные в параметрах процедур.

Вернемся к программе. Если она остановилась в пошаговом режиме, прервите её комбинацией *Ctrl+F2*. Затем исправьте заголовок процедуры указанным выше манером, откомпилируйте программу и вновь пройдите по шагам. Находясь внутри процедуры **Scan**, вы заметите, что переменные **arg** и **S** теперь изменяются синхронно (рис. 53). Это то, что нам нужно, стало быть, проблема решена!

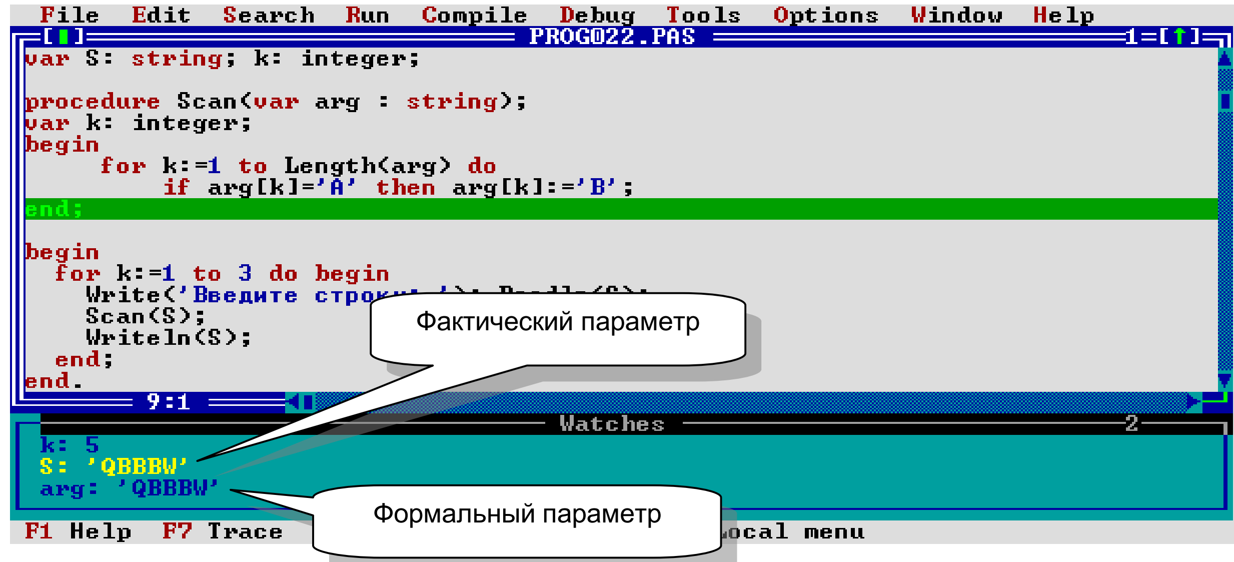


Рис. 53 – Синхронное изменение формального и фактического параметров

Далее можете «толкнуть» программу в непрерывном режиме, нажав комбинацию *Ctrl+F9*.

Итоги

- Для поиска ошибок применяют встроенный отладчик, который позволяет выполнять программу по шагам, а также просматривать переменные и менять их значения.
- При объявлении параметра без ключевого слова **VAR**, данные передаются только внутрь процедуры (по значению). Такой параметр используют как локальную переменную.
- Для передачи данных как внутрь процедуры, так и обратно, параметр объявляют с ключевым словом **VAR**. Тогда он служит ссылкой на другую переменную и меняется синхронно с нею.

А слабо?

А) Комбинация клавиш *Ctrl+F8* устанавливает так называемые точки останова на исполняемых операторах. Эта же комбинацией отменяет их. Точка останова — это строка, на которой отладчик задерживает выполнение программы и ждет команды на её продолжение в непрерывном или пошаговом режиме..

Установите точку останова на выходе из процедуры **Scan** (на строке **END**) и запустите программу в непрерывном режиме (*Ctrl+F9*). Что произойдет? Чем, по-вашему, удобны точки останова?

Б) Перед запуском программы установите курсор внутри процедуры **Scan** и испытайте действие команды *Run → Go to cursor* (клавиша *F4*).

Глава 22
О передаче параметров

Современные программы — даже не самые сложные — насчитывают тысячи строк. Как же распределена эта сложность? Почти вся она «размазана» по процедурам и функциям, а главную программу составляют обычно несколько строчек. Процедуры и функции, вызывая друг друга, передают данные словно эстафету по цепочке. Будущий профессионал должен овладеть тонкостями этого механизма.

Процедура обмена

Рассмотрим процедуру с несколькими параметрами. Пусть надо обменять значения в переменных **A** и **B**, это можно сделать так.

```
      T:= A;      { временно запомнить A }
      A:= B;
      B:= T;      { поместить в B то, что раньше было в A }
```

Здесь **T** — переменная для временного хранения данных. Поручим эту простенькую работу процедуре, которую назовем **Swap** (обмен). Создавать процедуру начнем, как водится, с заголовка. Поскольку в обмене участвуют два числа, оба их надо передать через параметры. Для разделения формальных параметров используют точку с запятой. Если заголовок процедуры будет таким:

```
procedure Swap (x: integer; y: integer);
```

мы не добьемся своего, поскольку при передаче по значению результаты не вернутся в вызывающую программу. Правильным будет заголовок с двумя ссылками на переменные.

```
procedure Swap (var x: integer; var y: integer);
```

Если формальные параметры имеют одинаковый тип и способ передачи, то заголовок можно сократить так.

```
procedure Swap (var x, y: integer);
```

Принцип объединения в заголовке тот же, что при объявлении однотипных переменных в секции **VAR**.

Теперь напишем процедуру **Swap** и программу P_22_1 для её проверки.

```
{ P_22_1 – процедура обмена и программа её проверки }

{ процедура обмена }
procedure SWAP(var x,y : integer);
var t: integer;
begin
      t:= x;      x:= y;      y:= t;
end;

var A, B : integer;
begin       {--- главная программа ---}
      A:= 10; B:= 20;
      Writeln('A= ', A, ' B= ', B);
      SWAP(A, B);
      Writeln('A= ', A, ' B= ', B);
      Readln;
end.
```

Работает ли эта программа? Обязательно проверьте!

Замена символов в строке

Вернемся к программе P_20_1, где возможности процедуры **Scan** небогаты: допускается менять только символы «A» на символы «B». А если надо менять символы по своему усмотрению? Пожалуйста! Добавим в заголовок процедуры пару формальных параметров, например, так.

```
procedure Scan(var arg: string; Ch1, Ch2: char);
var k: integer;
begin
      for k:=1 to Length(arg) do
            if arg[k]= Ch1 then arg[k]:= Ch2;
end;
```

Здесь параметры **Ch1** и **Ch2** определяют, что и на что надо поменять. Поскольку параметры однотипны, они разделяются запятой. Порядок объявления формальных параметров в заголовке не важен. Но важно, чтобы при вызове процедуры порядок фактических параметров был таким же. Вот пример правильного вызова (символ «1» меняется на символ «2»).

```
Scan(S, '1', '2');
```

А вот ошибочные:

```
Scan(S, '1');                { указаны не все параметры }
Scan('1', S, '2');           { нарушен порядок следования параметров }
Scan(S, '1', '2', '3');      { указан лишний параметр }
Scan(S, 1, 2);               { неверный тип параметров }
```

За соответствием фактических параметров формальным жестко следит компилятор. Исключение составляют встроенные в язык процедуры ввода-вывода, такие как **Readln** и **Writeln**, где допускается гибкая передача параметров разных типов.

Переработайте программу P_20_1 с тем, чтобы испытать новую версию процедуры замены символов, а затем исследуйте её в пошаговом режиме.

О передаче строк

Передача строковых данных таит свои тонкости. Рассмотрим процедуру **Calc** для подсчета заданного символа в некоторой строке.

```
procedure Calc(arg: string; Ch: char; var Res: integer);
var k: integer;
begin
      Res:=0;
      for k:=1 to Length(arg) do
            if arg[k]= Ch then Res:= Res+1;
end;
```

Процедура принимает три разнотипных параметра: строку **arg**, символ **Ch** и ссылку на переменную **Res** — в ней возвращается результат. Здесь всё правильно. Но недаром говорят: «меньше знаешь, — крепче спишь», — мой сон тревожит параметр **arg** строкового типа.

Поскольку строка может содержать до 255 символов, параметру **arg** отводится немалая память — 256 байтов! При передаче по значению все эти байты копируются в параметр **arg**, и на это тратится время. Если же параметр **arg** будет ссылкой на строку, то копирования не потребуется, и программа заработает быстрее. Вдобавок мы и память сэкономим, ведь ссылка на строку занимает в памяти всего 4 байта! Раз так, объявим процедуру иначе.

```
procedure Calc(var arg: string; Ch: char; var Res: integer);
```

Этот вариант лучше, но не сработает, если в вызове процедуры указать строковую константу, например:

```
      Calc('PASCAL', 'L', Result);
```

Здесь компилятор воспротивится не на шутку, требуя в первом параметре переменную. И будет прав, поскольку ключевое слово **VAR** в заголовке процедуры объявляет ссылку на переменную, а не на константу. Что делать? Вернуться к первому способу? Нет, есть лучшее средство: вместо ключевого слова **VAR** укажите в заголовке слово **CONST**, вот так.

```
procedure Calc(const arg: string; Ch: char; var Res: integer);
```

Такая ссылка будет годна как для переменной, так и для константы.

```
      Calc('PASCAL', 'L', Result);        { вызов с константой }
      Calc(S, 'L', Result);               { вызов с переменной }
```

Слово **CONST** перед формальным параметром, так же, как и **VAR**, определяет ссылку на данные, но без возможности их изменения. Обратите внимание на двойное назначение слов **CONST** и **VAR**: их применяют и для открытия соответствующих секций, и для объявления ссылочных параметров.

Итоги

- Количество фактических параметров, их тип и порядок следования в вызове должны совпадать со списком формальных параметров процедуры.
- Для экономии памяти и повышения быстродействия строковые данные (и другие сложные типы данных) передают по ссылке с применением ключевых слов **CONST** и **VAR**.
- Если строку передают по ссылке только внутрь процедуры, используют ключевое слово **CONST**, а если обратно или в оба направления – слово **VAR**.
- Если строка передается только внутрь процедуры и далее применяется там как локальная переменная, то ключевые слова **CONST** и **VAR** в объявлении параметра не ставят (так происходит передача параметра по значению).

А слабо?

А) Введите в компьютер программу P_22_1 и проверьте её работу.

Б) Измените программу P_20_1 так, чтобы заменяемый и замещаемый символы передавались в процедуру **Scan** через параметры.

В) Напишите программу для проверки рассмотренной выше процедуры **Calc**, подсчитывающей символ в строке.

Глава 23
Функции

Процедуры и функции — сестры-близнецы, потому и носят общее имя — подпрограммы. Всё, что сказано о передаче параметров, относится и к тем, и к другим. И всё же функции чем-то отличаются от процедур, иначе, зачем их придумали? А затем, чтобы упростить возвращение результата.

Нередко таким результатом бывает число, строка, символ или булево значение. Конечно, вернуть результат можно и через ссылку на переменную, но функция сделает это удобней — через своё имя. Результат, возвращаемый функцией, можно вставлять внутрь выражений наряду с переменными и константами. Взять хотя бы знакомые нам функции **Random** и **Length**, вызовы которых можно применить, например, так.

```
    x:= 1+ Random(10);      { арифметическое выражение }
    Writeln(Length(S));     { печатается длина строки S };
```

Функции избавляют программиста от объявления лишних переменных, упрощая программы и повышая их надежность. Сейчас мы научимся создавать собственные функции.

Объявление функции

Подобно объявлению процедуры, объявление функции состоит из заголовка и тела. Тело строят по тем же правилам, что и для процедур, а вот заголовок выглядит немного иначе.

```
function Имя_Функции : Тип;                      { функция без параметров }
function Имя_Функции (Параметры) : Тип;          { функция с параметрами }
```

Отличий от процедуры всего два. Во-первых, вместо ключевого слова **PROCEDURE** указано слово **FUNCTION**. А во-вторых, завершает заголовок тип функции (тип возвращаемого ею результата), — его указывают после двоеточия.

Пример функции

Разберем всё это на примере. Создадим функцию, выбирающую большее из двух чисел. Разумеется, что функция будет принимать два параметра — сравниваемые числа, и возвращать будет тоже число. Стало быть, её заголовок может быть таким:

```
function Max(arg1, arg2 : integer) : integer;
```

Имя функции выбираем на свой вкус, здесь имя **Max** вполне подходит, оно означает MAXIMUM (наибольший). К этому заголовку прилепим тело функции, состоящее из одного условного оператора.

```
function Max(arg1, arg2 : integer) : integer;
begin
      if arg1 > arg2
            then Max:= arg1
            else Max:= arg2
end;
```

Но откуда взялась переменная **Max**, которой присваиваем значение? Ведь мы её не объявляли! А её и не надо объявлять, — это имя нашей функции, оно и принимает в себя результат. Мало того, если результату не присвоить значение, он останется неопределенным, и это будет ошибкой!

Созданная нами функция может вызываться так.

```
      A:= Max( 20, 10 );            { A = 20 }
      Writeln( Max( A, B ) );       { печатается большее из A и B }
```

Вызов функции можно использовать даже как фактический параметр в её собственном вызове, то есть организовать вложенные вызовы, например:

```
      A:= Max ( Max( 20, 10 ), 40 );                 { A = 40 }
      A:= Max ( Max( 20, 10 ), Max( 200, 100 ) );    { A = 200 }
```

В первом случае сначала вызывается функция **Max(20,10)**, вставленная как первый фактический параметр, а затем **Max(20,40)**, — то есть результат первого вызова подставляется параметром во второй. Похоже работает и другой пример, только функция вызывается трижды. Полезно понаблюдать за такими вызовами через отладчик. Напишите главную программу для исследования функции **Max** и прогоните её в отладчике.

Подсчет символов в строке

В прошлой главе я предложил вам написать процедуру для подсчета заданного символа в строке. Если вы справились с той задачей, то для возврата результата наверняка воспользовались ссылкой на переменную. Теперь рассмотрим решение с применением функции.

Начнем, разумеется, с заголовка функции, дадим ей имя **Count** (подсчет).

```
function Count(const Str : string; Ch : char): integer;
```

Функция принимает два параметра: ссылку на строку и символ, который надо подсчитать. Напомню, что ключевое слово **CONST** в объявлении параметра позволяет ссылаться и на константу, и на переменную. Тело функции строим на базе цикла со счетчиком.

```
function Count(const str : string; ch: char): integer;
var N, i: integer;
begin
     N:=0;  { обнуляем счетчик }
     for i:=1 to Length(str) do
         if str[i]=ch then N:= N+1;
     Count:= N;   { определяем результат функции }
end;
```

Подсчет символов в массиве ведется в локальной переменной **N**, и лишь по завершении цикла результат копируется в имя функции. Грубой ошибкой было бы накапливать счетчик прямо в имени функции:

```
      if str[i]=ch then Count:= Count+1; { - это ошибка! }
```

Запомните: в теле функции её имя применяется только слева от оператора присваивания! Есть исключения из этого правила, но мы пока не будем их касаться.

И, наконец, напишем программу P_22_1 для проверки функции **Count**. В главной программе функция вызывается сначала для переменной **S**, а затем для константы «BANAN». Причем во втором случае она вызывается дважды, а результат суммируется. Испытайте эту программу.

```
{ P_23_1 - подсчет заданных символов в строке }

function Count(const str : string; ch: char): integer;
var N, i: integer;
begin
     N:=0;  { обнуляем счетчик }
     for i:=1 to Length(str) do
         if str[i]=ch then N:= N+1;
     Count:= N;   { передаем результат через имя функции }
end;
```

```
var S: string;
begin {--- главная программа  ---}
   S:='PASCAL';
   Writeln( Count(S, 'A'));
   Writeln( Count('BANAN', 'N') + Count('BANAN', 'B'));
   Readln;
end.
```

Возврат строк

Вернемся к программе P_20_1, заменяющей символы «A» на символы «B». Помните сколько крови она попортила прежде чем заработать? Заменив процедуру **Scan** на функцию с тем же именем, мы решим проблему возврата результата. Результат, разумеется, должен иметь строковый тип. Обратите внимание на то, что ключевые слова **VAR** или **CONST** в заголовке не указаны, а потому параметр **arg** можно употребить в теле функции в качестве локальной переменной.

```
{ P_23_2 – замена символов в строке с применением функции }

function Scan(arg : string): string;
var k: integer;
begin
      for k:=1 to Length(arg) do
            if arg[k]='A' then arg[k]:='B';     { замена в параметре arg }
      Scan:= arg;
end;
var S: string;    k: integer;
begin {--- главная программа ---}
      for k:=1 to 3 do begin
            Write('Введите строку: '); Readln(S);
            Writeln(Scan(S));
      end;
      Readln;
end.
```

Когда результат не важен

Хорошая функция возвращает правильный результат, а отличная делает ещё что-нибудь полезное. Программисты нередко поручают одной функции несколько дел, вот пример: напишем функцию **Swap** (обмен) булевого типа, принимающую ссылки на две переменные. Функция должна сравнить эти переменные и вернуть **TRUE**, если первая из них окажется больше второй. Мало того, в этом случае она

должна обменять значения этих переменных (как в процедуре **Swap**, рассмотренной ранее). Короче, функция будет такой.

```
function Swap( var a1, a2 : integer) : Boolean;
var t: integer;
begin
      if a1 > a2
            then begin
                  { обмен значений переменных }
                  t:=a1; a1:=a2; a2:=t;
                  Swap:= true
            end
            else Swap:= false
end;
```

Где применить такую функцию? Пусть переменные **N1**, **N2**, **N3** содержат три разных числа. Переложим эти числа так, чтобы в **N1** оказалось наименьшее, а в **N3** — наибольшее число, то есть, чтобы соблюдалось условие: **N1 < N2 < N3**. Такая сортировка выполняется тремя вызовами функции **Swap** (в комментариях показаны результаты обмена).

```
      Swap(N1, N2);              { N1 < N2 }
      if Swap(N2, N3)            { N2 < N3 }
            then Swap(N1, N2);   { N1 < N2 < N3 }
```

Здесь в первой и третьей строках функция вызывается как процедура, поскольку возвращаемый ею булев результат не используется. Во второй строке она вызывается как функция, поскольку результат использован оператором **IF**.

Возможность вызывать функцию как процедуру называют расширенным синтаксисом (Extended syntax), — он должен быть разрешен в настройках компилятора, иначе вызов функции как процедуры компилятор сочтет ошибкой.

Неявная переменная Result

Современные версии компиляторов дают новую возможность в части построения функций. Так, компилятор Delphi позволяет, наряду с именем функции, для возврата результата использовать автоматически объявляемую переменную **Result**. Тип переменной **Result** совпадает с типом функции. Тогда функцию подсчета символов можно упростить так.

```
function Count(const str : string; ch: char): integer;
var i: integer;
begin
     Result:=0;   { обнуляем счетчик }
     for i:=1 to Length(str) do
         if str[i]=ch then Result:= Result + 1;
end;
```

Как видите, переменную **Result** можно использовать как в левой, так и в правой части оператора присваивания. Последнее значение переменной станет результатом функции.

Итак, потратив три главы на изучение процедур и функций, мы готовы, наконец, к настоящему делу. Сколько можно в цацки играть? В следующей главе приступим к шифрованию файлов!

Итоги

- Функции – это подпрограммы, возвращающие результат через свое имя. Тип возвращаемого результата указывают в заголовке функции.
- В теле функции обязательно присваивают значение функции (через её имя), иначе результат останется неопределенным, случайным.
- Вызовы функций можно использовать в выражениях наряду с константами и переменными.
- Когда результат функции не используется, её вызывают как процедуру. При этом через настройки компилятора должен быть позволен расширенный синтаксис – «Extended syntax».

А слабо?

А) Напишите функцию для поиска буквы в заданной строке. Она должна возвращать **TRUE**, если в строке есть хоть одна эта буква, и **FALSE** в противном случае. Напишите программу для проверки функции. Или слабо?

Б) Напишите функцию для определения позиции буквы в заданной строке. Функция должна вернуть позицию первой такой буквы или ноль, если буквы в строке нет. Напишите программу для проверки функции.

В) Напишите функцию и программу для её проверки, принимающую число и возвращающую строку: слово «четное» или «нечетное» в зависимости от четности или нечетности параметра. Подсказка: для проверки четности числа **N** надо проверить остаток от его деления на два: **if (N mod 2) = 0 then ...**

Глава 24
Криптография

Говорят, что хороший разведчик стоит целой дивизии. Ещё бы! Ведь лишенный секретов противник почти безоружен. Но вот умолкли пушки, а разведка не спит, — у мирного времени свои тайны: коммерческие и технические секреты. Впрочем, если секретов нет, их можно придумать, — почему бы нам не поиграть в шпионов? Приятно сознавать, что «отмыленное» приятелю письмо никто, кроме вас двоих, не прочтет, — надо лишь зашифровать его. Придумана уйма способов шифрования, есть даже наука об этом — криптография; сейчас и мы коснемся краешка этой премудрости.

Секреты Юлия Цезаря

Римскому полководцу Юлию Цезарю выпали лихие времена: письмо, отправляемое им с гонцом в отдаленный уголок империи, могло стать добычей недругов, — ведь на дорогах было неспокойно. Это надоумило Юлия шифровать свои письма. В чем заключался метод полководца?

Цезарь просто-напросто заменял одни буквы другими путем кругового сдвига алфавита на несколько позиций. На рис. 54 показано превращение букв при сдвиге алфавита на две позиции. Буква «А» становится буквой «В», буква «Б» — буквой «Г» и так далее. Двум последним уготовано превратиться соответственно в буквы «А» и «Б». Такое шифрование превращает письмо в дикую абракадабру!

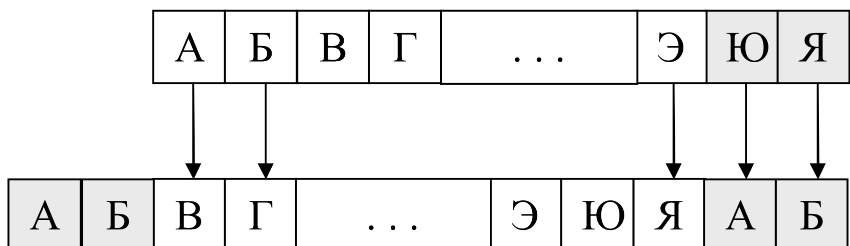


Рис. 54 – Принцип шифрования Юлия Цезаря

Как расшифровать её? Очень просто — сдвинуть буквы в обратную сторону. Но надо знать количество сдвигов — это число называют ключом шифра (в примере на рисунке ключ шифра равен двум). Разумеется, что ключ шифра и метод шифрования знали лишь двое: получатель письма и сам Юлий Цезарь.

Пойдем и мы вслед римскому полководцу, — создадим программу для зашифровки текстового файла и его расшифровки. Скажу прямо: задача непростая, а потому решать её будем в два этапа. Вначале освоим шифрование отдельной строки, а уж потом замахнемся на файл. Но начнем с шифрования отдельного символа.

Суть проблемы

Зашифровать строку — значит зашифровать каждый её символ. Будь у нас готовая функция шифрования символа, задача решалась бы вмиг. Так займемся ею и начнем с заголовка. Дадим нашей функции имя **Encrypt** — «шифровать», она должна принимать исходный символ и возвращать другой, зашифрованный. Значит, заголовок функции может быть таким:

```
function Encrypt (X: char): char;
```

Теперь сосредоточимся на теле функции и рассмотрим известные нам приёмы. Один из них состоит в применении каскада условных операторов:

```
if X='A'
      then Encrypt:='B'
else if X='Б'
      then Encrypt:='Г'
else . . .
```

Насколько удачно это решение? Прикинем количество вложенных операторов в этой лесенке. В русском алфавите 33 буквы, если взять заглавные и строчные, то получится 66. А если надумаем шифровать ещё и латинские буквы, и цифры и знаки препинания, то наберется около двух сотен символов. Такая лесенка условных операторов растянется на несколько этажей!

Не прибегнуть ли к оператору выбора **CASE**? Тогда тело функции будет намного проще:

```
case X of
      'A': Encrypt:='B';
      'Б': Encrypt:='Г';
      . . .
end;
```

Обратите внимание, что метками оператора **CASE** здесь служат символы, — скоро вы узнаете, почему такое возможно. Этот вариант очевидно лучше первого, хотя две сотни меток — тоже не подарок. Но главное неудобство в ином: при изменении ключа шифра придется переписать все ветви оператора **CASE**, а это, согласитесь, скучно. Не поискать ли иного решения, простого и гибкого?

О кодировании символов

Первые компьютеры принесли инженерам массу неудобств. Взять хотя бы ввод и вывод данных. Дисплеи, принтеры и звуковые карты — тогда никто не слышал о них! Результат размышлений цифрового мозга высвечивался лампочками

на инженерной панели ЭВМ, и в эту двоичную цветомузыку был посвящён лишь узкий круг мудрецов. Со временем изобрели простые принтеры, способные печатать лишь цифры, а затем и более совершенные — для печати букв и других символов. Как действуют подобные устройства?

Процессор компьютера, как известно, оперирует с числами. А людям подавай то текст, то картинку. Как связать одно с другим? Здесь инженеры вспомнили об алфавите. Ведь буквы в нём упорядочены, а значит, каждой букве можно сопоставить число; например, букве «А» — 1, «Б» — 2 и так далее. Такое сопоставление называют кодированием, оно и решает проблему представления символов. Намерившись напечатать некоторый символ, компьютер передает его код на принтер, а уж принтер знает, как поступить с этим числом. При вводе с клавиатуры происходит обратное преобразование: нажатие клавиши заставляет клавиатуру отправить в процессор код соответствующего символа.

Итак, символы внутри компьютера кодируются числами. Мы посчитали, что общее количество букв, цифр и других знаков составляет более двухсот. Инженеры не поскупились и отвели для кодирования символов 256 чисел — от 0 до 255 включительно. Почему именно 256, а не 300 или 500?

Дело в том, что в двоичной системе счисления 256 — это круглое число, оно равно двойке в восьмой степени (если вам знакомо слово «байт», то речь о нём). Так был создан алфавит для компьютеров, он включает буквы, цифры, знаки препинания и управляющие символы, — последние выполняют специальные действия с печатающим устройством, например, перевод на следующую строку.

Понятно, что можно придумать несметное количество вариантов кодирования символов. Желая добиться взаимопонимания между техническими устройствами разных изготовителей, инженеры договорились о единой системе кодирования. Теперь она известна под именем ASCII (читается «аски») — American Standard Code for Information Interchange — американский стандартный код для обмена информацией. Со временем этот стандарт стал международным. Ныне в ходу несколько стандартов кодирования, один из них (для MS-DOS) представлен в приложение И.

Все кодируемые символы разбиты на три группы. Первую составляют управляющие символы с кодами от 0 до 31. Их названия пока вам мало о чем скажут, обратите внимание лишь на символы с кодами 10 и 13, — они служат для разбивки текста на строки.

Вторую группу составляют символы с кодами от 32 до 127, — здесь собран весь латинский алфавит, цифры и основные знаки. Коды латинских букв следуют в порядке латинского алфавита, причем разница между кодами одноименных заглавных и строчных букв составляет 32.

Наконец, в третьей группе собраны русские буквы, символы псевдографики (применяются для рисования таблиц) и редко используемые знаки. Коды

большинства русских букв тоже следуют в порядке русского алфавита, но некоторые выпадают из этой последовательности. К тому же, разница между кодами заглавных и строчных русских букв не всегда составляет 32.

Заметьте также, что символы русского и латинского алфавитов со схожими начертаниями (такие, как «A», «B», «P») представлены разными кодами!

Чудесные превращения

Итак, символы в компьютере представлены своими кодами, то есть числами. А с числами работать легко: для превращения кода одного символа в код другого надо лишь прибавить либо вычесть некоторое число — шифрующий ключ.

Но как превратить символ в число и наоборот, — число в символ? Ведь это данные разных типов. Паскаль поможет вам своими встроенными функциями. Преобразовать число в символ можно либо функцией **CHR**, которая изначально присутствовала в языке, либо её современным аналогом по имени **CHAR**, которым я иногда буду пользоваться в дальнейшем. Обе функции принимают число, а возвращают символ, вот их объявления:

```
function Chr(arg : integer) : char;
function Char(arg : integer) : char;
```

Случайно ли, что имя функции **CHAR** совпадает с именем типа данных? Нет, ведь на самом деле обе эти функции ничего не делают! Они лишь подсказывают компилятору, что число в скобках — это код символа. И всё! С такими «ненастоящими» функциями мы ещё встретимся, их применяют для преобразования типов данных. Вот как можно напечатать нескольких символов с их кодами.

```
var n: integer;
begin
      for n:=40 to 50 do
            Writeln(n,'  ', Chr(n))
end.
```

Для обратного преобразования — символа в число — применяют другую «ненастоящую» функцию по имени **ORD** (от ORDER — «порядковый номер»). Сейчас компиляторы предлагают и её аналог по имени **Byte**. Функция действительно возвращает порядковый номер символа в таблице ASCII, вот её заголовок:

```
function Ord(arg : char) : integer;
```

Воспользовавшись ею, мы тоже можем напечатать символы с их кодами.

```
var c: char;
begin
      for c:='A' to 'F' do
            Writeln(c,'  ', Ord(c))
end.
```

Здесь счетчиком цикла **FOR** служит переменная символьного типа. Паскаль допускает это, поскольку «знает», что каждому символу соответствует некоторое число — его код.

Итак, научившись превращать числа в символы и наоборот, мы оказались в шаге от поставленной цели — шифрования символа.

Шифрование символа

Вернемся к функции шифрования **Encrypt**, теперь мы можем упростить её до предела.

```
function Encrypt(arg: char): char;
begin
      Encrypt:= Chr(Ord(arg) + CKey);
end;
```

Здесь **CKey** — ключ шифра, хранящийся где-то в глобальной константе или переменной (Caesar — «цезарь»). Превратив символ **arg** в число, мы прибавляем к нему ключ, а полученную сумму вновь превращаем в символ.

Всё хорошо, прекрасная маркиза, за исключеньем пустяка: сумма в скобках может оказаться больше 255, а символа с таким кодом не существует! Как тогда поступит функция **Chr**? Она вернет символ, укоротив его код на 256. Например, функция **Chr(260)** вернет символ с кодом 260 – 256 = 4. Устроит нас это? Никак нет, поскольку первые 32 символа таблицы (от 0 до 31) — это управляющие коды. Оказавшись в файле, такие символы нарушат его структуру, и редактор не сможет прочесть его.

Значит, при передаче суммы в функцию **Chr** надо проверить, не превышает ли она 255? Если да, то обрубим ей «хвост» и сдвинем ещё на 32 позиции выше, чтобы попасть в область видимых символов с кодами от 32 и далее.

```
if X>255 then X:=X-256+32;   { смещаем «хвост» - в начало видимых символов }
```

Так получаем окончательный вариант функции шифрования символа.

```
function Encrypt(arg: char): char;
var x: integer;
begin
      x:= Ord(arg)+ CKey;
      if x>255    then x:= x-256+32;
      Encrypt:= Chr(x);
end;
```

Расшифровка символа

Понятно, что для расшифровки символа мы выполним обратный сдвиг. После вычитания ключа проверим, не попадает ли полученная разность в область управляющих символов? Если попадает, поправим её, сместив в область видимых символов. Вот текст функции расшифровки **Decrypt**.

```
function Decrypt(arg: char): char;
var x: integer;
begin
      x:= Ord(arg)- CKey;
      if x<32     then x:= x+256-32;
      Decrypt:= Chr(x);
end;
```

Теперь всё готово для построения программы зашифровки и расшифровки строки P_24_1.

```
{ P_24_1 - Шифрование строки}
const CKey = 2; { Ключ Цезаря }

{----- Зашифровка одного символа -----}
function Encrypt(arg: char): char;
var x: integer;
begin
      x:= Ord(arg)+ CKey;
      if x>255    then x:= x-256+32;
      Encrypt:= Chr(x);
end;
```

```
{----- Расшифровка одного символа -----}
function Decrypt(arg: char): char;
var x: integer;
begin
      x:= Ord(arg)- CKey;
      if x<32     then x:= x+256-32;
      Decrypt:= Chr(x);
end;

{----- Зашифровка строки -----}
procedure EncryptStr(var arg: string);
var k: integer;
begin
      for k:=1 to Length(arg) do
            arg[k]:= Encrypt(arg[k]);
end;
{----- Расшифровка строки -----}
procedure DecryptStr(var arg: string);
var k: integer;
begin
      for k:=1 to Length(arg) do
            arg[k]:= Decrypt(arg[k]);
end;
                  {----- Главная программа -----}
var   S: string;
      Oper: integer;
begin
      repeat
            Write('Введите строку: '); Readln(S);
            Writeln('Укажите операцию: 1- шифровать,’+
                              ’ 2- расшифровать,’+
                              ’ Прочие - выход');
            Readln(Oper);
            case Oper of
                  1: EncryptStr(S);
                  2: DecryptStr(S);
                  else Break;
            end;
            Writeln(S);  { печатаем результат }
      until false;
end.
```

Программа нуждается лишь в кратких пояснениях. Глобальная константа **CKey** содержит ключ шифра. Если со временем захотите сменить его, достаточно будет изменить константу и заново откомпилировать программу. Далее следуют описания двух функций: **Encrypt** и **Decrypt** — для зашифровки и расшифровки символа. Процедуры **EncryptStr** и **DecryptStr** шифруют и расшифровывают строки, передаваемые им по ссылке **VAR**. И, наконец, в главной программе организован цикл для ввода шифруемой строки и кода выполняемой операции (**Oper**).

Откиньтесь в кресле и полюбуйтесь простотой блоков, составляющих эту программу! А во что бы мы превратили её, свалив в кучу эти простые алгоритмы? В заключение приведу протокол шифрования: я ввел слово «pascal» и зашифровал его, получив слово «rcuecn». Затем ввел строку «rcuecn» и расшифровал её, получив назад данное мною слово.

```
Введите строку: pascal
Операции: 1 - шифровать, 2 - расшифровать, прочие - выход
Введите операцию: 1
rcuecn
Введите строку: rcuecn
Операции: 1 - шифровать, 2 - расшифровать, прочие - выход
Введите операцию: 2
pascal
Операции: 1 - шифровать, 2 - расшифровать, прочие - выход
Введите операцию: 3
```

Вряд ли я удержу вас от испытания столь полезного изделия. Во избежание ошибок шифруйте строки небольшой длины. Рекомендую также хотя бы разок пройти программу по шагам.

Итоги

- В памяти компьютера символы представлены своими кодами — числами.
- Общее количество символов составляет 256, из них первые 32 — это управляющие, а остальные — видимые символы.
- Для преобразования числового кода в символ применяют функцию **Chr**. Для обратного превращения — символа в число — пользуются функцией **Ord**..
- Паскаль «знает» о том, что символы кодируются числами, поэтому в счетчике цикла **FOR** допустимы символьные переменные, а в метках оператора **CASE** — символьные константы.

А слабо?

А) Измените программу шифрования с тем, чтобы ключ задавать с клавиатуры и передавать в процедуры и функции через параметр. Заголовки процедур и функций сделайте такими:

```
function EncryptChar(arg: char; key: integer): char;
procedure EncryptStr(var arg: string; key: integer);
```

Кажется, что проще держать ключ в глобальной переменной, но крупные программы этот приём запутывает, — там передают данные через параметры.

Б) Предположим, вы пятикратно зашифровали строку. Можно ли расшифровать её? И как это сделать?

В) Для введенной пользователем строки напечатать позиции всех входящих в неё символов (кроме пробелов) в алфавитном порядке. Для символов, которые встречаются несколько раз, напечатать их позиции в одной строке. Например, для слова «PASCAL»:

```
A – 2 5
C – 4
L – 6
P – 1
S – 3
```

Г) Для введенной пользователем строки напечатать позиции всех встречающихся в ней символов, кроме пробелов, в порядке их следования в строке. Например, для слова «PASCAL»:

```
P – 1
A – 2 5
S – 3
C – 4
L – 6
```

Д) Строки текстовых файлов порой содержат управляющие символы, например символ горизонтальной табуляции (код 9). Обработка этих символов нашей программой нарушит структуру файла. Исправьте функции **`Encrypt`** и **`Decrypt`** так, чтобы они не изменяли символы, коды которых меньше 32.

Глава 25
Текстовые файлы

Мы мастерим программу шифрования текста. Шифрование отдельной строки освоено нами в предыдущей главе. Теперь научимся читать строки из одного файла и записывать их в другой.

Файлы хорошие и разные

Файлы — это хранилища данных, там может быть всё что угодно: музыка, фильмы, книги. Ясно, что эта информация как-то закодирована, то есть, представлена в виде чисел — байтов. Файл любого типа — это набор байтов, хранящийся на диске (говорим пока о дисковых файлах). Каждому типу файлов нужен свой подход: к файлу нужна программа, «понимающая» его содержимое. Вам угодно слушать музыку? — к вашим услугам медиа-проигрыватель. Или надо печатать текст? — тогда запустите редактор текста. Но не наоборот! А всё потому, что каждый тип файлов обладает структурой, понятной лишь соответствующей программе. Таким образом, файл и программа для работы с ним составляют логическое единство, — одно без другого лишено смысла.

Стало быть, структура или формат файла — его важнейшая характеристика. Все файловые форматы можно разделить на две категории:

- текстовые файлы;
- все прочие файлы, — их называют двоичными или бинарными.

О формате файла можно судить по его расширению. К текстовым относятся файлы с расширениями TXT — текст, BAT — пакетный файл, LOG — файл протокола и многие другие. Файлы наших программ с расширением PAS — тоже текстовые. А вот документы в формате Word (с расширением DOC) обладают сложной структурой, правильнее отнести их к бинарным. То же скажем о книгах PDF-формата. В отличие от DOC и PDF, текстовые файлы открываются простыми редакторами текста — вроде «Блокнота» или редактора нашей IDE, который тоже работает лишь с текстовыми файлами.

Формат текстовых файлов

Итак, любой файл — это набор байтов, записанных на диске. Как же расположены байты в текстовых файлах? — мы должны это знать. Воспользуемся неким «волшебным микроскопом» и рассмотрим через него отдельные байты небольшого текстового файла, составленного из четырех строк: в первой помещены три символа «1», во второй — два символа «2», третья строка пуста, а четвертая содержит символ «3».

Примечание. Вы можете исследовать текстовый файл в HEX-режиме просмотра такими программами, как Far, Total Commander и им подобными.

```
111
22

3
```

Наш воображаемый микроскоп изобразит этот файл цепочкой чисел (здесь показаны десятичные числа, хотя в HEX-режиме видны шестнадцатеричные).

```
49 49 49  13 10  50 50  13 10  13 10  51  13 10
```

Числа 49, 50 и 51 — это коды символов «1», «2» и «3» (по кодировке ASCII), а подчеркнутые мною числа 13 и 10 — это парочка управляющих байтов, разбивающая файл на строки. Открыв такой файл редактором, мы не увидим управляющих байтов, но в файле они есть! Любая программа, работающая с текстовыми файлами, умеет находить эти ограничители строк при чтении текста и вставлять их в файл при записи в него.

История названий ограничителей исходит из глубины веков. Символ с кодом 13 назван Carriage Return — «возврат каретки» или сокращенно CR. Те, кто застал электрические пишущие машинки прошлого, помнят: перед печатью следующей строки, каретка такой машинки сдвигалась в крайнюю правую позицию, — это и есть возврат каретки.

А управляющий символ с кодом 10 назван Line Feed (LF) — «подача строки». Он заведовал подачей бумаги в продольном направлении с тем, чтобы следующая строка печаталась после предыдущей. Вот так и работал консольный интерфейс прошлого: барабанил буквочку за буквочкой, пока не получал управляющие коды CR и LF. Тогда каретка со скрежетом сдвигалась вправо, барабан, дёрнувшись, слегка смещал бумагу вперед, и печаталась следующая строка.

С годами формат текстовых файлов не изменился, и будет жить, пока существуют компьютеры. Секрет его живучести — в простоте и универсальности. В некоторых операционных системах текстовые файлы разбивают на строки не парой символов CR+LF, а лишь одним из них. Это по сути ничего не меняет, — файл по-прежнему являет последовательность строк-макаронин, нарубленных управляющими символами.

Доступ к текстовым файлам

В Паскале можно работать с файлами любых типов — и текстовыми, и бинарными. Сейчас нас интересуют лишь текстовые, о бинарных пока умолчим.

Сложно ли работать с текстовыми файлами? Расслабьтесь, — это совсем не больно! Вы уже работаете с ними, даже не подозревая об этом. Чтение и запись строк в текстовые файлы выполняют всё теми же процедурами **Readln** и **Writeln**. Но с одним маленьким отличием: в первом параметре этих процедур

дается ссылка на файловую переменную типа **TEXT**, которая должна быть объявлена в программе следующим образом:

```
var F: Text;      { файловая переменная }
```

Тогда чтение и запись через переменную **F** выполнятся так:

```
Readln (F, S);    { Чтение одной строки файла в переменную S }
Writeln (F, 'Эта строка запишется в файл');
```

Где тут сложности? Их не видно, но пока неясно вот что:

- С каким именно файлом мы работаем? Ведь на диске их так много!
- В каком месте файла будет прочитана строка, и куда она будет помещена при записи?

Для ответа рассмотрим порядок чтения книги. Обычно я поступаю так:

1. Выбираю книгу на полке.
2. Открываю её в начале.
3. Читаю, пока не прочту или не усну.
4. В конце концов закрываю книгу и возвращаю на полку.

Точно так же — в четыре счета — обрабатывается файл. Далее в этой главе мы займемся чтением из файла, а запись в него рассмотрим в следующей главе.

Чтение из файла

Пусть нами объявлена файловая переменная **F** типа **TEXT**. Прежде чем воспользоваться ею для чтения некоторого файла, надо связать имя этого файла с файловой переменной. Это похоже на выбор книги для чтения — первый шаг в нашем списке. Связывание выполняют процедурой **Assign** — «назначить», в неё передают два параметра: файловую переменную и имя файла, например:

```
Assign(F, 'C:\AUTOEXEC.BAT');
```

Имя файла можно задать константой, переменной или их комбинацией — строковым выражением. Оно должно отвечать правилам, действующим в операционной системе. Указанный файл должен существовать, и система должна знать, где его найти. Впрочем, процедура **Assign** ничего не проверяет, она лишь помещает имя файла внутрь файловой переменной. И, если файла с указанным именем нет, процедура «не заметит» этого, но ошибка обнаружится на следующем шаге — при попытке открыть файл.

Второй шаг подготовки к чтению — открытие файла. Это вроде открытия книги на первой странице, оно выполняется процедурой **Reset** (что значит сброс или установка в исходное состояние). Этой процедуре нужен лишь один параметр — файловая переменная.

```
Reset(F);
```

Процедура **Reset** готовит файл к чтению, обращаясь при этом к операционной системе. Система выделяет память для работы с файлом, а также блокирует его, не давая другим программам удалить файл. После успешного открытия файловую переменную можно использовать далее в процедуре **Readln** так, как это было сказано выше. А если имя файла оказалось неверным или файл не существует? Тогда вызов процедуры **Reset** приведет к ошибке: программа сообщит: «File not found» — файл не найден, и аварийно прекратит работу.

После успешного открытия файла переходят к третьему этапу — собственно чтению из него (чтению книги). С этим вы уже знакомы, поскольку чтение выполняется известной процедурой **Readln**. Например, прочитать строку из файла можно так.

```
Readln(F, S);
```

Здесь **S** — это переменная строкового типа. Обратите внимание: в переменную **S** попадут только видимые символы строки, а управляющие коды — разделители строк — останутся «за бортом».

Но которая из строк файла будет прочитана? Первая, вторая или иная? При первом вызове после **Reset** процедура **Readln** прочтет первую строку файла, при втором — вторую и так далее. Если организовать цикл, то чтение продолжится вплоть до последней строки.

Применительно к чтению файлов говорят о позиции чтения, хотя увидеть эту позицию нельзя. Вызов процедуры **Reset** устанавливает эту воображаемую позицию в начало первой строки файла. Последующие вызовы процедуры **Readln** сдвигают её к началу очередной строки.

А что случится после чтения последней строки? Позиция достигнет конца файла, и очередной вызов процедуры **Readln** вызовет ошибку — событие крайне нежелательное. Чтобы избежать его, надо отслеживать достижение конца файла. Паскаль даёт для этого функцию по имени **EOF**, что означает End Of File — «конец файла». Булева функция **EOF** принимает один параметр — файловую переменную, и возвращает **TRUE**, когда позиция чтения «упирается» в конец файла.

```
    if Eof(F)
        then { достигнут конец файла }
        else { можно продолжать чтение }
```

Как видите, функцией **EOF** нельзя определить позицию чтения (то есть, номер читаемой строки); она сообщает лишь о том, достигнут конец файла или нет.

Что делать с прочитанной книгой? — закрыть и вернуть на полку. Так же поступают и с файлом — закрывают его. Эту операцию выполняют процедурой **Close** — «закрыть».

```
    Close(F);
```

Закрытие файла освобождает память, выделенную для него операционной системой, и снимает блокировку, давая возможность другим программам делать с файлом всё, что угодно. Закрытие освобождает и саму файловую переменную, — теперь ею можно воспользоваться для доступа к другому файлу.

На рис. 55 показаны этапы чтения данных из файла.

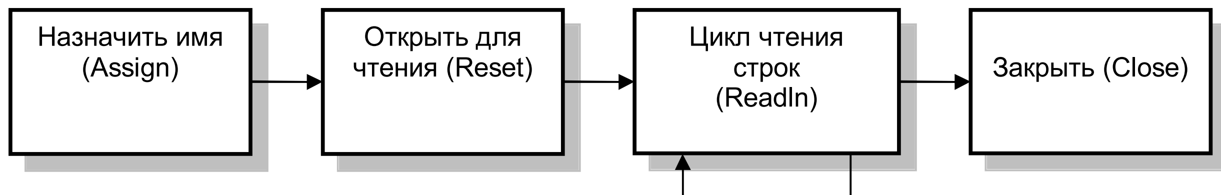


Рис. 55 – Четыре этапа чтения из файла

Последовательный доступ к файлу

Как видите, читать текстовый файл можно только последовательно, строку за строкой — от начала к концу файла, — нельзя читать строки в ином порядке. Поэтому текстовые файлы относят к файлам с последовательным доступом. В отличие от них, бинарные файлы (например, файлы баз данных) допускают произвольный доступ.

Впрочем, механизм последовательного доступа не запрещает программисту в любой момент вернуться к началу файла и повторить чтение — достаточно вызвать процедуру **Reset**.

Самореклама

Теперь испытаем то, что узнали о чтении текстовых файлов. Напишем небольшую программу, выводящую на экран свой собственный исходный текст, вот её первый вариант.

```
{ P_25_1 - распечатка текста программы }
var   F: text;          { файловая переменная }
      S: string;        { строка }
begin
      Assign(F, 'P_25_1.pas');          { назначаем собственное имя }
      Reset(F);                         { открываем файл для чтения }
      repeat
            if Eof(F) then Break;       { прекратить, если конец файла }
            Readln(F, S);               { прочитать строку из файла }
            Writeln(S);                 { вывести строку на экран }
      until false;
      Close(F);         { закрываем файл }
      Readln;           { ждать Enter }
end.
```

Подчеркнутый оператор проверяет достижение конца файла, и делает это перед чтением строки. Если же проверять в конце цикла

```
      . . .
until Eof(F);
```

это неизбежно приведет к ошибке после чтения последней строки файла.

Цикл с проверкой в начале

Достижение конца файла надо проверять своевременно! Для этого в Паскале есть подходящий оператор цикла, — пора познакомиться с ним. До сих пор мы обходились двумя циклическими операторами, а именно:

- циклом с проверкой условия в конце **REPEAT-UNTIL**;
- циклом со счетчиком **FOR-TO-DO**.

Новый для нас оператор цикла строится из двух ключевых слов, вот его формат:

WHILE <условие> **DO** <оператор>

По-русски это читается так: «ПОКА условие истинно, ВЫПОЛНЯТЬ оператор такой-то». После ключевого слова **DO** допускается лишь один оператор, но на практике требуется больше. Потому здесь часто вставляют операторный блок **BEGIN-END**, в итоге получается такая конструкция.

```
WHILE <условие> DO  BEGIN
     <последовательность операторов>
END
```

Обратите внимание, что условия продолжения циклов в операторах **WHILE-DO** и **REPEAT-UNTIL** взаимно противоположны! Первый из них выполняется, пока условие ИСТИННО, а второй — пока оно ЛОЖНО.

С новым оператором «самораспечатка» станет такой.

```
{ P_25_2 - распечатка текста программы }
var   F: text;          { файловая переменная }
      S: string;        { строковая переменная }
begin
      Assign(F, 'P_25_2.pas');          { назначаем собственное имя }
      Reset(F);                         { открываем файл для чтения }
      while not Eof(F) do begin         { пока не конец файла }
            Readln(F, S);               { прочитать строку из файла }
            Writeln(S);                 { вывести строку на экран }
      end;
      Close(F);         { закрываем файл }
      Readln;           { ждем нажатия Enter }
end.
```

В условии цикла **WHILE** видим отрицание **NOT**, значит, цикл будет выполняться, пока НЕ обнаружен конец файла. Проверьте работу этой программы. В следующей главе мы рассмотрим запись данных в текстовый файл и завершим наш шифровальный проект. А сейчас, как обычно, подведем итоги.

Итоги

- Текстовые файлы содержат строки видимых символов, отделенные друг от друга невидимыми на экране управляющими кодами **CR** (возврат каретки) и **LF** (перевод строки).
- К текстовым файлам обращаются через файловые переменные типа **TEXT**.
- Перед чтением файла нужны два шага: 1) связывание файловой переменной с именем файла процедурой **Assign**, и 2) открытие файла для чтения процедурой **Reset**.
- Для чтения отдельных строк вызывают процедуру **Readln**, при этом первым параметром процедуры указывают файловую переменную.

- После открытия файла его чтение начинается с первой строки; каждый вызов процедуры **Readln** смещает позицию чтения в начало следующей строки.
- Чтение файла возможно, пока не будет прочитана последняя строка. Попытка чтения за концом файла вызовет аварию программы.
- Чтобы узнать о достижении конца файла, вызывают функцию **Eof**, которая возвращает **TRUE**, если достигнут конец файла.
- Признак окончания файла исследуют в начале цикла, и для этого лучше подходит оператор цикла **WHILE-DO**.
- По окончании работы с файлом его закрывают процедурой **Close**.

А слабо?

А) Можно ли связать текстовую переменную **F** с файлом оператором присваивания?

```
F := 'c:\autoexec.bat';
```

Б) Напишите программу для вывода на экран файла, имя которого задается с клавиатуры.

В) Напишите три функции для подсчета:

- строк в файле;
- видимых символов в файле;
- всех символов файла (фактический объём файла).

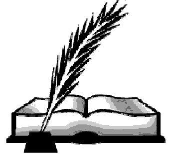
Функции принимают один параметр — ссылку на файловую переменную. Напишите программу, определяющую упомянутые характеристики файла.

Г) Объявите две файловые переменные, свяжите их с одним и тем же файлом, а затем откройте через обе переменные. Вызовет ли это ошибку? Объясните результат, исходя из здравого смысла.

Д) Усовершенствуйте программу «вопрос-ответ» (глава 16) с тем, чтобы ответы хранились не в программе, а в отдельном текстовом файле. Тогда пользователи программы сами смогут сочинять ответы.

Е) Напишите процедуру для вывода на экран **N**-й строки файла, где **N** — параметр процедуры. Воспользовавшись этой процедурой, напишите программу для распечатки строк файла в обратном порядке. Подсказка: предварительно посчитайте количество строк в файле.

Глава 26
Я не читатель, — я писатель!



Наш шпионский проект по шифрованию файла подвигается к финишу. Ступим завершающий шаг: освоим запись в текстовый файл.

Запись в текстовый файл

Порядок записи в текстовый файл схож с его чтением, судите сами:

- для доступа к файлу используют файловую переменную типа **TEXT**;
- файловую переменную надо связать с файлом процедурой **Assign**;
- по окончании записи файл закрывают процедурой **Close**.

Вот схема, где отражены четыре этапа записи в файл (рис. 56).

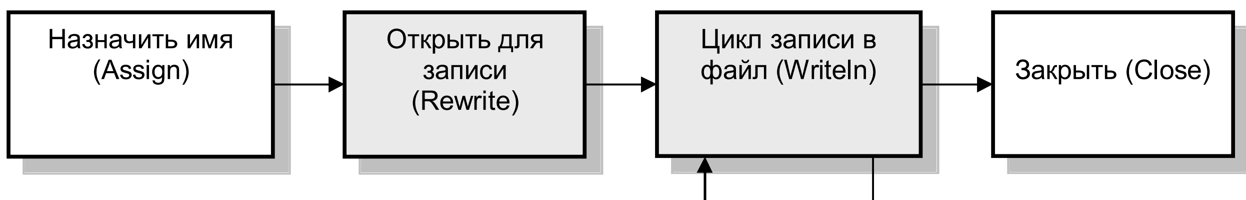


Рис. 56 – Четыре этапа записи в файл

Как видите, запись в файл отличается от чтения вторым и третьим этапами, ими и займемся.

Итак, после привязки файла процедурой **Assign**, файл открывают для записи процедурой **Rewrite** — «перезапись».

```
Rewrite(F);
```

Тут находим первое отличие: если для чтения требуемый файл должен существовать, то для записи этого не нужно. Если файла ещё нет, будет создан новый пустой файл. А при наличии файла он будет очищен, и вся информация в нём сотрется, как мел с доски. Будьте внимательны, иначе лишитесь нужной информации!

В открытый таким образом файл можно записывать нужные нам строки. Как? Вы уже знаете — процедурой **Writeln**. А что указать первым параметром? Правильно, — файловую переменную, вот так:

```
     Writeln(F, S);  { F – переменная типа TEXT, S – строковая }
```

Каждый вызов такой процедуры добавляет в конец файла очередную строку с разделителями. В отличие от чтения, где надо следить за достижением конца файла, при записи вы ограничены лишь объемом винчестера и здравым смыслом (в большей степени последним).

По окончании записи файл закрывают всё той же процедурой **Close**. Как и при чтении, закрытие файла освобождает выделенную для него память и снимает блокировку, позволяя другим программам работать с ним. К тому же закрытие файла гарантирует сохранение записанных данных на диске.

Пример записи в файл

Рассмотрим небольшой пример: заполнение файла числами от 1 до 10.

```
{ P_26_1 }
var   F: text;          { файловая переменная }
      k: integer;
begin
      Assign(F, 'P_26_1.txt');      { назначаем имя файла }
      Rewrite(F);             { открываем файл для записи }
      for k:=1 to 10 do       { записать 10 строк с числами }
            Writeln(F, k);
      Close(F);         { закрываем файл }
end.
```

Есть вопросы? Запустите программу и проверьте, работает ли она. Запустили? Теперь отыщите в папке с программой файл «P_26_1.TXT» и откройте его любым редактором. Уверен, что вы обнаружите в нём столбик из десяти чисел.

Завершение шпионского проекта

Подойдя к финалу нашего проекта, мы научились: 1) шифровать отдельную строку, 2) читать строки из файла и 3) записывать строки в файл. Пора соединить всё это: читая строки исходного файла, будем шифровать их и записывать в другой файл, — так будет работать наша программа.

Прежде всего, договоримся об именах файлов. Назначив зашифрованному файлу постоянное имя, например «CRYPT.TXT», мы избавим себя от ввода его имени с клавиатуры. Вводить мы будем либо имя исходного файла — при зашифровке, либо имя конечного файла — при расшифровке. Обозначим эти неизвестные нам имена файлов как InFile и OutFile, и тогда схема обработки файлов будет такой.

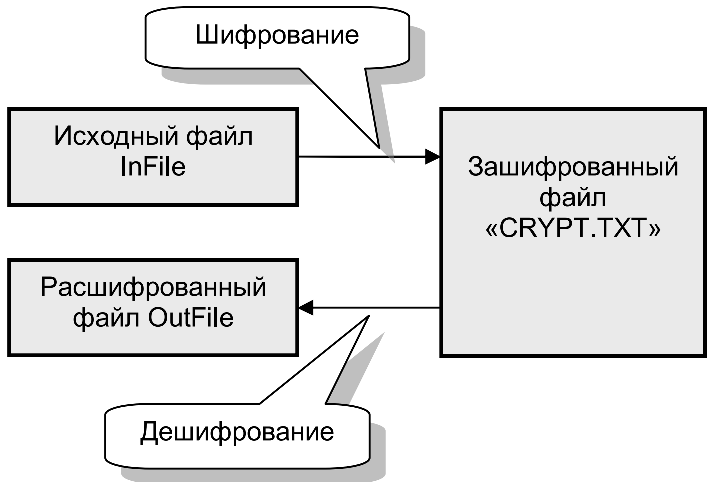


Рис. 57 – Схема именования файлов при шифровании и расшифровки

С учетом этих договоренностей составим блок-схему программы (рис. 58).

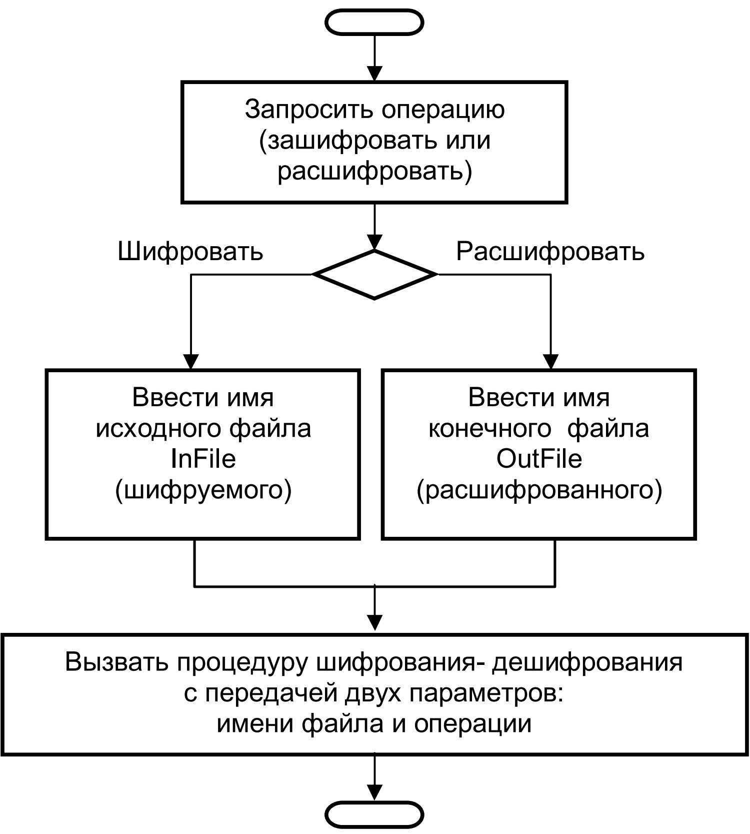


Рис. 58 – Блок-схема программы шифрования/расшифровки

Основную работу поручим процедуре шифрования, блок-схема которой показана на рис. 59. В неё передаём два параметра: имя обрабатываемого файла и код операции (зашифровать или расшифровать).

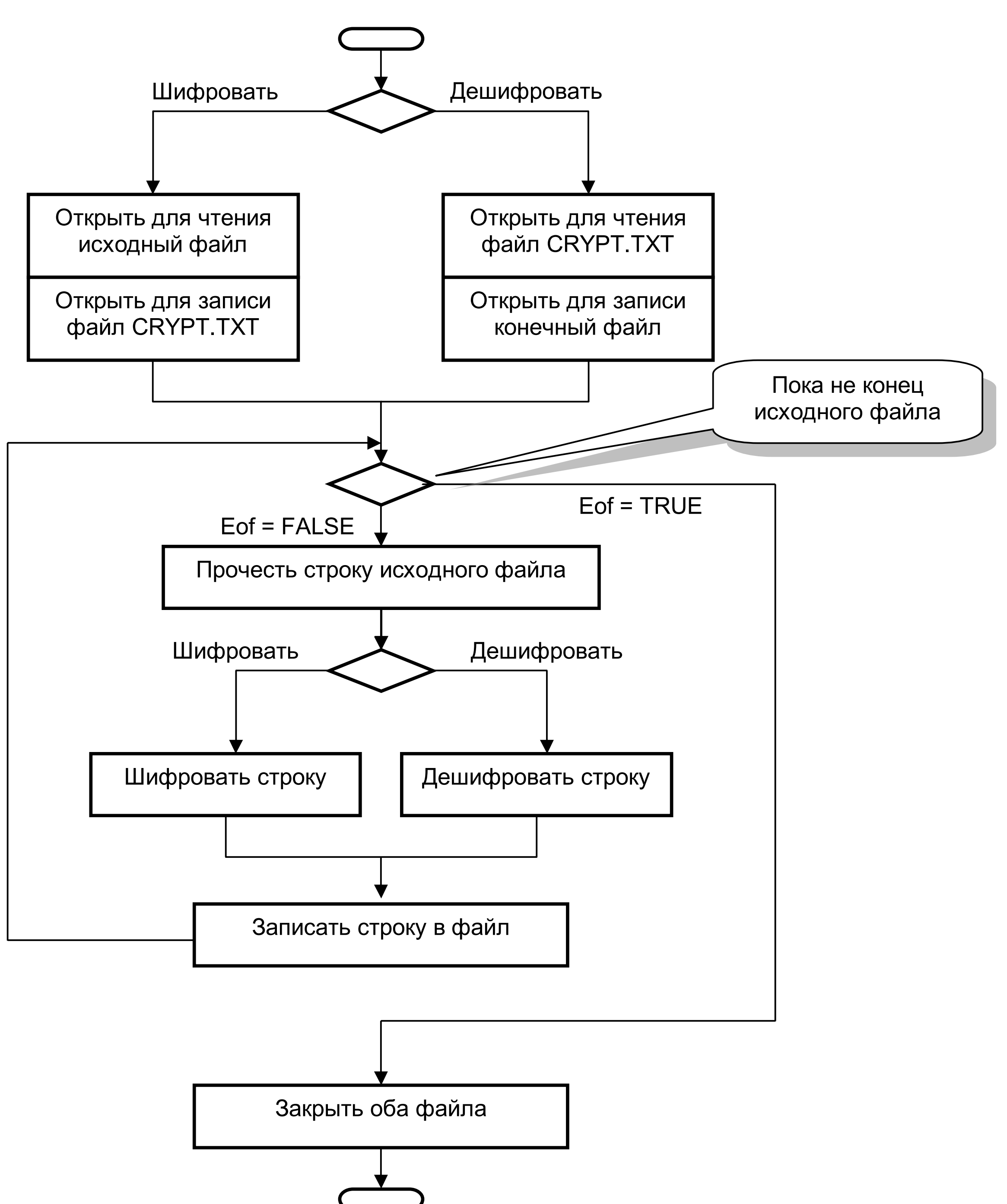


Рис. 59 – Блок-схема процедуры шифрования-расшифровки

Теперь мы готовы смастерить шпионскую программу. Может быть, сами справитесь? По крайней мере, попытайтесь. Функции и процедуры шифрования символов и строк возьмите из программы P_24_1. Написав свой вариант, сравните с представленным ниже.

```
{ P_26_2 - шифрование файлов }

const CKey = 2; { Ключ Цезаря }

            { Зашифровка символа }
function EncryptChar(arg: char): char;
var x: integer;
begin
  EncryptChar:=arg;
  if Ord(arg)>=32 then begin  { управляющие символы не трогаем! }
    x:= Ord(arg)+ CKey;
    if x>255 then x:= x-256+32;
    EncryptChar:= Char(x);
  end;
end;
            { Расшифровка символа }
function DecryptChar(arg: char): char;
var x: integer;
begin
  DecryptChar:=arg;
  if Ord(arg)>=32 then begin  { управляющие символы не трогаем! }
    x:= Ord(arg)- CKey;
    if x<32 then x:= x+256-32;
    DecryptChar:= Char(x);
  end;
end;
            { Зашифровка строки }

procedure EncryptString(var arg: string);
var k: integer;
begin
      for k:=1 to Length(arg) do arg[k]:= EncryptChar(arg[k]);
end;
            { Расшифровка строки }

procedure DecryptString(var arg: string);
var k: integer;
begin
      for k:=1 to Length(arg) do   arg[k]:= DecryptChar(arg[k]);
end;
```

```
      {----- Процедура шифрования файла -----}
procedure  CryptFile(const aFile: string; aOper: boolean);
const CFixName='CRYPT.TXT';   { фиксированное имя файла }
var   FileIn:  text;    { входной файл для чтения }
      FileOut: text;    { выходной файл для записи }
      S:    string;
begin
      if aOper then begin                { если шифровать }
            Assign(FileIn, aFile);
            Assign(FileOut, CFixName);
      end else begin                     { если расшифровать }
            Assign(FileIn, CFixName);
            Assign(FileOut, aFile);
      end;
      Reset(FileIn);           { открыть входной файл для чтения }
      Rewrite(FileOut);        { открыть выходной файл для записи }
      while not Eof(FileIn) do begin
            { пока не закончился входной файл }
            Readln(FileIn, S);      { читать очередную строку из файла }
            if aOper
                  then EncryptString(S)   { зашифровать }
                  else DecryptString(S);  { расшифровать }
            Writeln(FileOut, S);         { записать в выходной файл }
      end;
      { закрыть оба файла }
      Close(FileIn);    Close(FileOut);
end;
      {----- Главная программа -----}
var   S: string;
      Oper: boolean;    { TRUE - шифровать, FALSE - расшифровать }
begin
      Write('Укажите операцию (1 - шифровать, иначе - расшифровать ):');
      Readln(S);
      Oper:= S='1';     { Oper=TRUE если S='1' }
      if Oper
            then Write('Введите имя шифруемого файла: ')
            else Write('Введите имя расшифрованного файла: ');
      Readln(S);
      CryptFile(S, Oper);     { Вызов процедуры шифрования }
      Write('OK, нажмите Enter'); Readln;
end.
```

Пространные пояснения излишни. Признак выполняемой операции формируется в булевой переменной **Oper** в третьей строке главной программы по цифре, введенной в переменную **S**. Значение **Oper=TRUE** влечет зашифровку файла, а **FALSE** — расшифровку. Затем в переменную **S** вводится имя обрабатываемого файла. В конце концов, вызывается процедура **CryptFile** с передачей в неё двух параметров: имени файла и признака выполняемой операции (**aFile** и **aOper**). Приставка «а» в начале имен этих параметров (префикс) помогает при чтении программы отличить параметр от других переменных.

Полюбуйтесь, во что превратила наша программка один из файлов на Паскале (приведен небольшой фрагмент).

```
}"Rtqi2420rcu"□
xct"Ocp"<"uvtkpi=
}///"░гьёднзпкз"▒т░шзжхтэ"///□
rtqegfwtg"Rcwug=
dgikp
"""""Ytkvgnp*)Пвиокфз"Gpvgt<)+=""Tgcfnp=
```

Как говорится, родная мама не узнает! Всё, что попадает в «мясорубку» нашего шифровальщика, обращается в фарш. Однако последующая расшифровка перемолотого файла в точности восстановила его.

Примененный нами метод шифрования не так уж крут, опытный взломщик легко раскроет его. Но фундамент заложен, и когда-нибудь вы придумаете изощренные методы шифрования. Например, ключ шифра можно сделать переменным и зависящим от номера символа в строке или файле. Подумайте над этим. Если же вы намерены заняться криптографией всерьез, изучайте математику! Для программиста это наука номер один.

Итоги

- Для записи в текстовый файл, как и для чтения, требуется файловая переменная типа **TEXT**.
- Перед записью в файл выполняют два действия: связывание переменной с файлом процедурой **Assign** и открытие файла для записи процедурой **Rewrite**.
- Вызов процедуры **Rewrite** либо создаёт новый файл, либо очищает существующий (вся бывшая в нём информация теряется!).
- Запись отдельных строк в файл выполняют процедурой **Writeln**, первым параметром здесь указывают файловую переменную.
- По окончании записи файл закрывают процедурой **Close**, — это гарантирует сохранение данных на диске.

А слабо?

А) Программа создает файл, печатает в него несколько строк с числами, а затем выводит этот файл на экран. Воспользуйтесь одной файловой переменной.

Б) Программа для нумерации строк файла. Строки исходного файла должны копироваться в конечный файл с добавлением перед каждой строкой её номера, например:

Исходный файл:

```
В лесу родилась елочка,
В лесу она росла.
Зимой и летом стройная,
Зеленая была.
```

Конечный файл:

```
1
В лесу родилась елочка,
2
В лесу она росла.
3
Зимой и летом стройная,
4
Зеленая была.
```

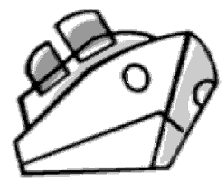
В) Скопировать один файл в другой:

- с перестановкой местами четных и нечетных строк;
- с перестановкой строк в обратном порядке (см. условие задачи «Е» к 25-й главе).

Г) Для передачи по интернету секретного текстового файла разбейте его на два других: в первый запишите нечетные строки исходного файла, а во второй — четные. Напишите для этого программу, или слабо?

Д) Создайте программу для объединения двух файлов (см. условие предыдущей задачи). Из первого составьте нечетные строки конечного файла, а из второго — четные.

Глава 27
Дайте кораблю минутный отдых!



Ой, что мы с вами натворили! Могучая программа шифрования файлов дает нам право если не на медаль, то хотя бы на передышку. Пощадим наши серые клеточки и отправимся на экскурсию по своему кораблю — среде программирования Free Pascal. Ведь мы обошли ещё не все палубы этого лайнера. Сейчас рассмотрим настройку компилятора, а в следующей главе обсудим возможности текстового редактора и справочной системы.

Ошибка ошибке рознь

Где найти безгрешных программистов? Нет таковых! Лихорадочно барабаня по клавишам в попытке изваять очередной проект, мы то и дело ошибаемся. Часть этих ошибок отлавливает компилятор, — он видит синтаксические ошибки — нарушения правил языка. Вы споткнулись на ключевом слове или забыли объявить переменную? — нажмите клавишу *F9*, и компилятор «ткнет носом» в место ошибки. И пока не исправите свои огрехи, не надейтесь получить исполняемый файл. Зато и работают такие программы, «непосильным трудом нажитые», весьма надежно. Восхищенный новичок однажды породил афоризм: «компилируется — значит работает». Увы! если бы так! Некоторые ошибки проявляются лишь во время работы программы, — это ошибки времени исполнения — Runtime errors.

Фатальные ошибки

Да, компилятор Паскаля способен уберечь от многих ошибок, но посмотрите на следующий пример:

```
var   X : integer;
begin
      Readln(X);
      Writeln(100 div X);
end.
```

Программа печатает результат деления числа 100 на переменную **X**. Здесь нет синтаксических ошибок. И всё работает прекрасно, пока пользователь не введет число ноль. Тогда вместо результата деления вы получите неприятное сообщение «Runtime error 200», и программа прервется. Иначе говоря, деление на ноль не позволено никому, даже компьютеру.

Выручит ли здесь компилятор? Ведь это логическая ошибка, то есть ошибка в алгоритме. Нет, тут поможет лишь исправление программы, например, так.

```
var   X : integer;
begin
      Readln(X);
      if X<>0
            then Writeln(100 div X)
            else Writeln('Не делите на ноль, умоляю Вас!');
end.
```

Деление на ноль — это фатальная, то есть неисправимая ошибка, она приводит к аварийной остановке программы. Но случаются и ошибки иного рода.

«Простительные» ошибки

Вот пример по части обработки файлов, — на этой «кухне» мы уже побывали. Попытка открыть для чтения несуществующий файл влечет ошибку вода/вывода (по-английски — «I/O Error»), — это тоже ошибка времени исполнения. Кто виноват? Разумеется, пользователь, — данные надо вводить внимательней. Но программе от этого не легче, — она обязана как-то реагировать. Как? Проще всего аварийно завершиться. Но можно поступить мягче — разобраться в ситуации и подсказать пользователю, где он неправ.

И здесь компилятор поддержит вас, позволив настроить реакцию программы на некоторые ошибки времени исполнения. По сути, способов реагировать только два: прервать программу при появлении ошибки, либо нет. Вариант реакции настраивают через опции компилятора. Рассмотрим выгоды такой настройки на примере ошибок ввода/вывода.

Опции компилятора

Обратимся к настройкам компилятора. Щелкните по пункту меню *Options → Compiler…* (рис. 60), и перед вами появится окно для настройки опций (рис. 61).

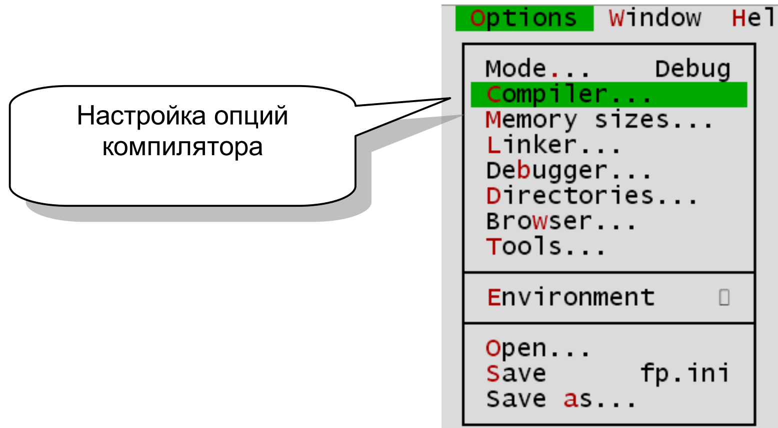


Рис. 60 – Выбор пункта меню для настройки опций компилятора

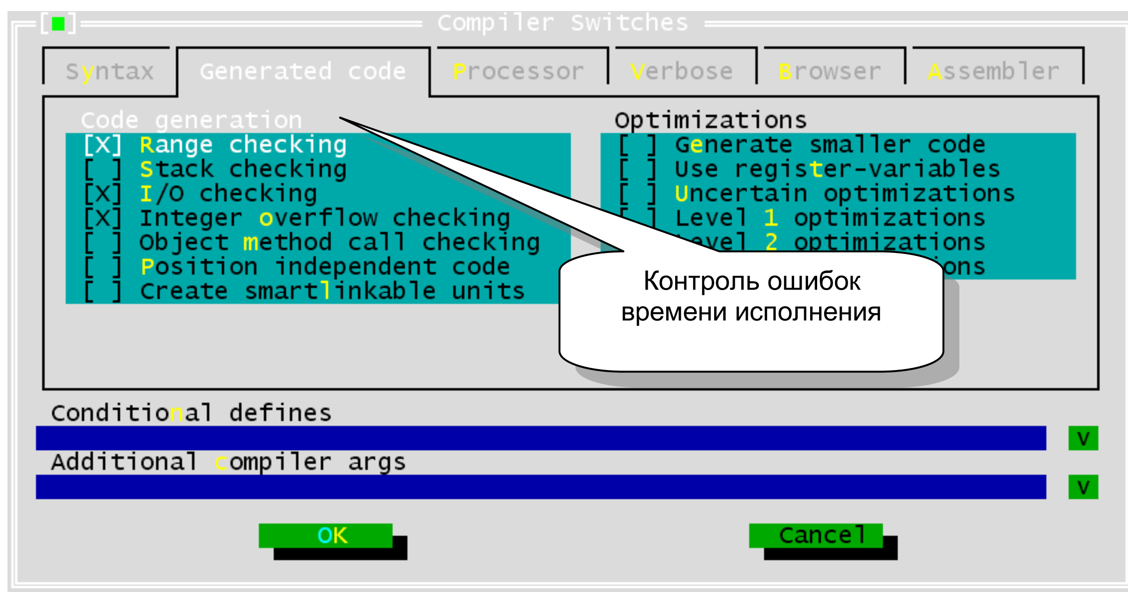


Рис. 61 – Окно настроек опций компилятора

Вкладка «Generated code» содержит нужную нам группу флажков «Code generation». Флажок «I/O checking» заведует реакцией программы на ошибки ввода-вывода («I/O» — это сокращение от Input/Output — «ввод/вывод»). При установленном флажке компилятор будет создавать исполняемые EXE-файлы так, что ошибки ввода-вывода аварийно завершат программу. А если сбросить флажок и снова откомпилировать ту же программу, она поведет себя иначе, — программа не погибнет, однако и работать, как следует, не будет. В чем же смысл настройки?

Обработка ошибок ввода-вывода

Смысл в том, чтобы самому следить за вероятными ошибками. Для слежки используют функцию **IOResult**, которая не имеет параметров и возвращает ноль, если ошибок ввода-вывода не произошло. А если эта неприятность всё же случилась, функция вернет ненулевой код ошибки, по которому легко выяснить её причину. Обратите внимание: функция возвращает состояние последней выполнявшейся операции ввода-вывода. При ошибке дальнейшая работа с файлом автоматически блокируется до его повторного открытия.

Рассмотрим способ безаварийного определения наличия файла на диске (здесь имя файла содержится в переменной **FileName**).

```
Assign(F, FileName);    Reset(F);
if IOResult=0
      then Writeln ('Нашелся файл '+ FileName)
      else Writeln ('Файл '+FileName+' не обнаружен!');
```

Этот фрагмент надёжно сработает только при отключенном флажке «I/O checking», иначе программа может прерваться аварийно. Стало быть, перед компиляцией надо проверять состояние флажка, а это хлопотно и ненадежно.

В настройках опций компилятора через меню есть и другой изъян. Как быть, когда в разных местах программы требуется по-разному реагировать на ошибки: где-то включить этот контроль, а где-то нет? Но флажок действует на всю программу в целом, глобально, — он не допускает выборочной настройки. Что делать?

Директивы компилятора

Выручают директивы, — особые сочетания символов, избирательно настраивающие компилятор. Директивы не являются элементами языка, поэтому вставляются в программу по-хитрому — внутри комментариев. Ведь комментарии, как известно, компилятор должен игнорировать, пропускать мимо ушей. Но компилятор просматривает и комментарии, «процеживая» их в поисках директив (подобно тому, как мы «процеживали» строки в поиске заменяемых символов).

Большинство директив выглядит как сочетание символа доллара «**$**» с латинской буквой и последующего знака «+» или «—». Всё это заключается внутрь комментария. Знак «+» включает действие директивы, а «—» — отключает, что равносильно установке или сбросу флажков на вкладке опций компилятора. Например, директива, управляющая реакцией на ошибки ввода-вывода, записывается так.

```
{ $I+       - включить контроль ввода-вывода }
{ $I-       - отключить контроль ввода-вывода }
```

В один комментарий можно вместить несколько директив. Перечень директив вы найдете в справке по компилятору и в приложении Ж.

Директиву – в студию!

Сейчас мы извлечем первую пользу из директив: сотворим функцию, определяющую наличие файла на диске. Применим для этого директиву **$I**. Наша булева функция будет возвращать **TRUE**, если файл, имя которого передано в параметре, существует. Вот её текст, а заодно и фрагмент тестирующей программы.

```
{ P_27_1 – определение наличия заданного файла }

function FileExists(const aName: string): boolean;
var F: text;
begin
      FileExists:= false;     { предполагаем, что файла нет }
      Assign(F, aName);
      {$I-}  Reset(F);  {$I+} { контроль отключен на время Reset }
      if IOResult=0 then begin      { если файл существует }
            Close(F);               { закрываем файл }
            FileExists:= true;
      end;
end;

begin       {--- главная программа ---}
      Writeln( FileExists('AUTO.BAT') );       { печатает false }
      Writeln( FileExists('C:\AUTOEXEC.BAT') );  { печатает true }
end.
```

В подчеркнутой строке процедура открытия файла **Reset** заключена между парой директив. Первая из них отключает контроль ошибок ввода-вывода, а вторая снова включает его. Это значит, что при выполнении процедуры **Reset** программа не прервется даже при отсутствии открываемого файла. Причем это уже не будет зависеть от состояния флажка в опциях компилятора, поскольку директивы имеют преимущество перед флажками, то есть более высокий приоритет.

Как сработает наша функция? После попытки открыть файл вызовем функцию **IOResult**. Если она вернула ноль, значит, файл существует, и его надо закрыть, поскольку никаких действий с ним внутри функции **FileExists** не намечается. Проверьте работу этой полезной функции, она ещё пригодится вам!

Парад директив

Разбогатев со временем собственными программами, вам, вероятно, захочется поделиться ими. При передаче исходных текстов важно передать и настройки опций компилятора, иначе EXE-файл может быть построен неправильно. Эти настройки лучше передать путём вставки директив компилятора прямо в программу. Но директив много, — запомнить их трудно, а ошибиться легко. Впрочем, есть один волшебный способ…

Откройте опции компилятора (рис. 61) и настройте в нём флажки так, как нужно, не забыв сохранить их кнопкой *OK*. Затем откройте файл с программой и нажмите волшебную комбинацию клавиш *Ctrl+O+O*. То есть, удерживая клавишу

CTRL, дважды нажмите латинскую букву «O». И — о, чудо! — в начале программы появятся строчки с настройками всех директив, например, такие.

```
{$IFDEF NORMAL}
  {$H-,I+,OBJECTCHECKS-,Q-,R-,S-}
{$ENDIF NORMAL}
{$IFDEF DEBUG}
  {$H-,I+,OBJECTCHECKS-,Q+,R+,S-}
{$ENDIF DEBUG}
{$IFDEF RELEASE}
  {$H-,I-,OBJECTCHECKS-,Q-,R-,S-}
{$ENDIF RELEASE}
```

Здесь представлены настройки директив для трех вариантов компиляции. Эти варианты (Normal/Debug/Release) выбираются в пункте меню *Options → Mode…*. Знаки «+» и «—» соответствуют состоянию флажков в окне опций. Директивы вида **$IFDEF** нужны для выбора одного из вариантов компиляции (об условных директивах я расскажу в главе 60). Можно упростить эту конструкцию, оставив, лишь одну строку.

```
  {$H-, I+, OBJECTCHECKS-, Q-, R-, S-}
```

Потребовалось изменить настройки? Пожалуйста! — Удалите эти строчки и повторите «волшебные заклинания». Или расставьте плюсы и минусы вручную.

Итоги

- Программист допускает два рода ошибок: синтаксические и семантические (смысловые).
- Синтаксические ошибки обнаруживает компилятор. Пока вы не исправите все такие ошибки, исполняемый файл не сформируется.
- Смысловые ошибки проявляются в ходе выполнения программы, – это ошибки времени исполнения. Такие ошибки кроются либо в алгоритме программы, либо в неправильных действиях пользователя.
- Реакция программы на некоторые ошибки определяется настройкой опций компилятора. Программа может либо пренебречь ошибкой, либо аварийно завершиться.
- Опции компилятора настраивают двумя способами: в диалоговом окне и вставкой директив непосредственно в программу.
- Директивы в тексте программы имеют преимущество (приоритет) перед настройками опций в диалоговом окне.

А слабо?

А) Выясните код ошибки, возвращаемый функцией **IOResult** при попытке открыть для чтения несуществующий файл. Напишите для этого небольшую программу.

Б) Сделайте то же самое, когда программа пытается открыть для записи файл с установленным атрибутом «только чтение». Для настройки атрибутов файла щелкните по файлу правой кнопкой мыши и выберите пункт «Свойства».

В) Дан файл, строки которого содержат круглые скобки (это может быть программа или математические выкладки — неважно). Ваша программа должна распечатать строки, где скобки расставлены неверно, вот примеры.

```
2+3        – правильно, хотя скобок нет;
(2+3       – ошибка – здесь нет закрывающей скобки;
()2+3()    – это правильно (хоть и лишено смысла);
))2+3((    – ошибка – скобки закрываются до открытия.
```

Рекомендация: для исследования строки напишите булеву функцию **Check**, возвращающую **TRUE**, если скобки расставлены без ошибок.

Г) Дребезг контактов давно уже бесит специалистов по электронике. Дребезг возникает кратковременно при замыкании-размыкании кнопок, тумблеров, реле и других подобных устройств. Сигнал от контактов поступает в микропроцессор с периодичностью, скажем, 100 раз в секунду. Если контакт постоянно разомкнут, микропроцессор принимает «0», а если замкнут — «1». При замыкании-размыкании контакт неустойчив, и процессор получает пачки чередующихся нулей и единиц, — надо отфильтровать эти ложные срабатывания.

Ваша программа будет моделировать поведение микропроцессора. Входной файл содержит последовательность нулей и единиц (по одному символу в строке). Первый символ примите как исходное значение сигнала, а дальше сигнал на выходе программы формируется так: если три подряд идущие значения совпадают, то берется это новое значение, а иначе сохраняется текущее, например:

На входе	На выходе
0	0
1	0
0	0
1	0
1	0
1	1
0	1

В выходной файл запишите в две колонки входной и выходной сигналы.

Глава 28
Редактор и справочная система

Ошибки, ошибки… Мы отбиваемся от них всеми средствами, и компилятор, как вы убедились, — важный рубеж в этой обороне. Важный, но не единственный. Согласитесь, лучше не допускать ошибок, чем устранять их. Хорошо, когда рядом есть мудрый советчик, — вовремя подскажет, объяснит, остережет. К счастью, в IDE есть такие «советчики» — это редактор текста и справочная система, — о них спою в этой главе.

Сегодня никого не удивишь возможностями нынешних IDE: тут и встроенный редактор и справочная система. Но так было не всегда. В начале 90-х годов прошлого века появление замечательных продуктов фирмы Borland было сродни чуду. Подумать только! Куча окон, «умная» раскраска текста, встроенная справочная система и отладчик, — это ли не чудеса?

Небьющиеся окна

Встроенный многооконный редактор — одно из новшеств Borland Pascal, он появился там едва ли не раньше самой MS Windows. Пока мы обходились одним окном редактора, но так будет не всегда, — в сложных проектах приходится одновременно открывать несколько файлов. Впрочем, это нужно уметь и сейчас, например, для копирования частей одной программы в другую.

Окна встроенного редактора схожи с окнами Windows: они открываются и закрываются, меняют размер и положение, допускают перенос кусков текста из одного окна в другое. Для управления окнами IDE служит раздел меню «Window». На рис. 62 рядом с названиями пунктов этого раздела показаны соответствующие им горячие комбинации клавиш.

Рассмотрим самые полезные возможности этого меню.

Команды *Next* и *Previous* переключают окна редактора. То же самое делается нажатием клавиш *F6* и *Sift+F6* или щелчками мыши по окнам. Текущее активное окно выдвигается на передний план и обрамляется особой рамкой с полосами прокрутки (в пассивных окнах этих полос нет).

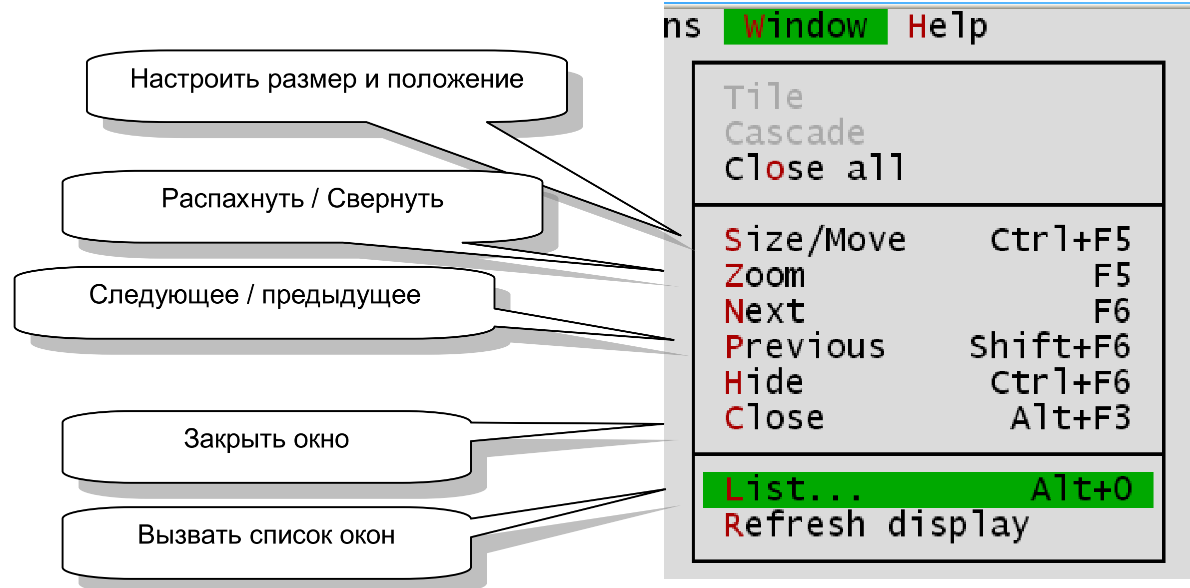


Рис. 62 – Пункт меню «Window»

Касательно активного окна надо сделать важное замечание, связанное с компиляцией. Когда открыто несколько окон, нажатие клавиши *F9* вызовет, вероятней всего, компиляцию файла, открытого в активном окне. Поэтому не забывайте перед компиляцией своей программы переключаться в нужное окно!

В скопище окон не мудрено заблудиться, и тогда выручит команда *Window* → *List* (комбинация *Alt+0*, где «0» — это цифра). Нажав её, вы увидите список всех открытых в данный момент окон (рис. 63).

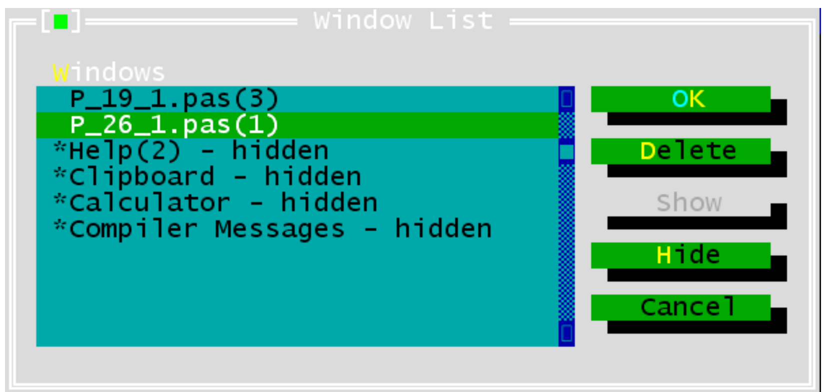


Рис. 63 – Список окон

Для перехода в нужное окно выделите его в списке и щелкните по кнопке *OK* (или сделайте двойной щелчок по строке с именем окна).

Изменить размер и положение окна можно командами *Size/Move* и мышью. Для перетаскивания окна «схватите» его мышью за верхнюю границу рамки, а для изменения размера — за правый нижний угол.

Буфер обмена

Кто сказал, что списывать — плохо? Копирование кусков текста — любимое занятие программистов. В самом деле, разумно ли набивать заново такой же или похожий кусок текста? Куда быстрее и надежней «скопипастить» его (от слов Copy — «копировать» и Paste — «вставить»). И время сбережем, и ошибиться трудней, — а ошибки мы душим всеми средствами, не так ли?

Куски текста копируются и в пределах одного файла, и между разными файлами. При этом копируемый фрагмент временно сохраняется в памяти — так называемом буфере обмена. Фокус с копированием выполняем в четыре счета:

- выделяем кусок текста (об этом чуть позже);
- копируем выделение в буфер обмена (*Ctrl+Insert*);
- помещаем курсор в то место, куда требуется вставить текст (в этом же или другом окне);
- вставляем текст из буфера обмена (*Shift+Insert*).

Теперь о выделении текста. Оно может выполняться и клавиатурой, и мышью. Поместите курсор в начале или в конце нужного куска текста. Затем одной рукой удерживайте клавишу *Shift*, а другой в это время двигайте текстовый курсор (клавишами со стрелками или любым способом, изменяющим положение курсора). Другой прием — «мышиный»: протащите мышь по нужному тексту, удерживая нажатой её левую кнопку.

Когда выделение станет ненужным, снимите его, щелкнув мышью в любом месте текста, либо нажав комбинацию *Ctrl+K+H*.

«Постойте, да ведь в редакторах Windows это делается точно так же» — скажет бывалый читатель. Да, но с одной поправкой: редакторы IDE и Windows используют разные буферы обмена, — у каждого свой. Текст, помещенный в буфер обмена Windows, не вставляется в окно IDE и наоборот. Кстати, содержимое буфера обмена IDE можно открыть через пункт меню *Edit → Show Clipboard*.

Примечание. В IDE Free Pascal под Windows есть пункты меню *Copy to Windows* и *Paste from Windows*, они служат для обмена кусками текста со средой Windows.

Справочная система

Как ни полезны «примочки» редактора текста, они не способны ответить на вопросы забывчивого программиста. Например, о правильном написании ключевого слова или о параметрах некоторой процедуры. С такими вопросами обращаются к руководству по языку, — и шурши страницами! Впрочем, в каком веке мы живем? Всё это есть в компьютере, надо лишь поискать. Справка открывается через пункт меню «Help» или клавишей *F1* — нажал, и вот тебе

счастье! Но если вас интересует что-то конкретное, лучше поступить иначе, — я расскажу о двух таких приемах.

Первый удобен, когда интересующее вас слово уже содержится в тексте. Установите курсор в пределах этого слова (под любой его буквой), например, под словом **IOResult**, и нажмите комбинацию *Ctrl+F1*. Если справочная система найдет «досье» на это слово, то покажет его в окне «Help» (рис. 64).

```
                              Help                                   2
 IOResult

 [Overview][Constants][Types][Classes][Procedures and functions][Variables][I
IOResult
Return result of last file IO operation

Declaration
Source position: systemh.inc line 906

function IOResult: Word;


Description
IOresult contains the result of any input/output call, when the {\$i-}
compiler directive is active, disabling IO checking. When the flag is read,
it is reset to zero. If IOresult is zero, the operation completed
successfully. If  non-zero, an error occurred. The following errors can
occur:
dos errors :

2
    File not found.
3
    Path not found.
4
F1 Help on help  Alt+F1 Previous topic  Shift+F1 Help index  Esc Close help
```

Рис. 64 – Вызов окна справки для слова «IOResult»

А если слово не обнаружится? Тогда справочник покажет список всех своих статей — так называемый индекс — и выделит ближайшее похожее слово. С этого момента вы можете самостоятельно искать нужные слова в индексе. Для поиска просто набирайте слово буква за буквой. Для повторного поиска сдвиньте курсор в любом направлении и снова набирайте слово. Когда искомое слово подсветится, для просмотра статьи просто нажмите *Enter*. Это второй прием получения справки. На рис. 65 показан результат поиска в индексе слова «Close».

Всё, что выделено в справке желтым цветом — это гиперссылки на статьи справочной системы. Щелкая по ним двойным щелчком (либо нажимая *Enter*), вы можете разгуливать по справочной системе, как по Интернету.

С окном справочной системы можно обращаться так же, как с окнами редактора: перемещать, изменять размеры. Вот где пригодится умение работать в многооконной среде! Хотите скопировать что-то из окна справки? Тогда выделите нужный фрагмент буксировкой мыши, а затем перенесите его в свою программу через буфер обмена (*Ctrl+Insert*, затем *Shift+Insert*).

```
 File  Edit  Search  Run  Compile  Debug  Tools  Options  Window  Help
 [■]                              Help                          2 [↕]
  CLONE_CHILD_SETTID                  CLONE_DETACHED
  CLONE_FILES [1]                     CLONE_FILES [2]
  CLONE_FS [1]                        CLONE_FS [2]
  CLONE_NEWNS                         CLONE_PARENT
  CLONE_PARENT_SETTID                 CLONE_PID [1]
  CLONE_PID [2]                       CLONE_PTRACE
  CLONE_SETTLS                        CLONE_SIGHAND [1]
  CLONE_SIGHAND [2]                   CLONE_STOPPED
  CLONE_SYSVSEM                       CLONE_THREAD
  CLONE_UNTRACED                      CLONE_VFORK
  CLONE_VM [1]                        CLONE_VM [2]
  cLong                               clong
  cLong [1]                           cLong [2]
  clongdouble [1]                     clongdouble [2]
  clonglong [1]                       clonglong [2]
  clonglong [3]                       clonglong [4]
  Close [1]                           Close [2]
  Close [3]                           Close [4]
  CloseDir                            CloseDotFill
  CloseExeFile                        CloseFile
  CloseGraph                          CloseSocket
  ClrEol                              ClrScr
  clrviewproc                         cmdline
  cmem                                CMSPAR
  cnAdded                             cnDeleting
  cnExtracting                        CO40
 F1 Help on help  Alt+F1 Previous topic  Shift+F1 Help index  Esc Close help
```

Рис. 65 – Поиск слова «Close» в индексе справочной системы

Напомню, что порядок установки справочной системы для IDE Free Pascal изложен в главе 4. Пользователи Borland Pascal могут найти в Интернете перевод этой справки на русский. Для установки русской справки поместите её файл в ту же папку, где находится исполняемый файл «BP.EXE», и дайте файлу справки стандартное имя «TURBO.TPH». Не забудьте предварительно сохранить или переименовать исходный «английский» файл, — а вдруг ещё пригодится?

Итоги

- Редактором текста можно открыть столько файлов, сколько вам нужно. Каждый файл открывается в отдельном окне; обращаться с окнами так же легко, как с окнами Windows.
- Для переноса кусков текста используйте буфер обмена.
- Буфер обмена IDE и буфер обмена Windows не связаны между собой. Для переноса текста в другие файлы Windows открывайте файлы своих программ в редакторах этой системы, например, в блокноте. Или воспользуйтесь пунктами меню *Copy to Windows* и *Paste from Windows.*
- Ищите ответы на свои вопросы в справочной системе IDE.

Глава 29
Читайте по-новому

Отдохнув на экскурсии, с новой силой набросимся на файлы, — ведь именно там хранятся наши данные. Научимся извлекать из файлов числа.

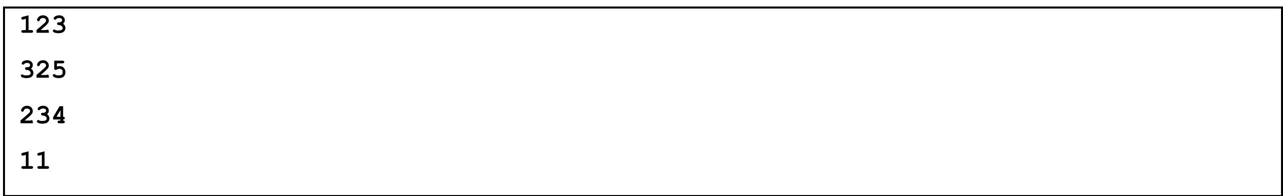
Полицейская база данных, версия 1

Одна из первых наших программ исполняла должность электронного часового, охранявшего секретный объект. Теперь поможем другой силовой структуре — дорожной полиции.

Долг автоинспектора — в числе прочего — поиск угнанных автомобилей. Работа нехитрая: заприметив подозрительный автомобиль, инспектор сверяет его номер со своей картотекой. И, если номер в картотеке найдется, принимает меры к задержанию автомобиля и поимке преступника. В отличие от часового, который помнит всего один пароль, полицейский роется в пухлой картотеке с тысячами номеров. В этом и состояла его главная трудность, пока не явились компьютерные базы данных.

Базы данных (сокращенно БД) — кто не слышал о них? На ум приходят базы данных Пентагона, ЦРУ и налоговой инспекции. Да, эти чудовищные БД впечатляют! Впрочем, чтобы увидеть базу данных, не спешите потрошить Пентагон. Вот расписание поездов, программа телепередач или классный журнал, — всё это простые базы данных. В конце концов, любая БД — это организованное хранилище данных, приспособленное для удобного поиска информации. Нам тоже по силам создать несложную полицейскую базу данных.

Соорудим такую базу с номерами угнанных автомобилей, и воспользуемся для этого текстовым файлом. Допустим, что номера автомобилей — это числа; тогда база данных — это файл с напечатанными в столбик номерами: каждая строка файла содержит один номер. Номера могут следовать в любом порядке, например:



```
123
325
234
11
```

Такой файл можно напечатать любым редактором текста, в том числе и встроенным в IDE. Мы так и поступим: создайте новый файл, впечатайте в него десяток-другой пришедших на ум номеров и сохраните в рабочей папке под именем «Police.txt». Окно с этим файлом пока не закрывайте, оно ещё пригодится.

Всё, база данных готова! Только не предлагайте полицейскому рыться в этом файле, — вместо благодарности вы услышите совсем другие слова. В довершение

доброго дела напишем программу для поиска номера в этой базе. Такая программа должна работать как часовой, запрашивая у полицейского номер автомобиля и сообщая о том, содержится ли этот номер в БД. Признаком выхода из программы будет ввод нулевого номера.

Итак, приступим к программе P_29_1. Схема главной программы ясна из вышесказанного. Выносить заключение об автомобиле будет функция булевого типа, принимающая два параметра: файловую переменную, связанную с нашей БД, и номер искомого автомобиля. Если номер в базе данных обнаружится, функция вернет значение **TRUE**. Ввиду простоты алгоритма не буду рисовать блок-схему. Если чуете в себе силу, напишите программу сами, и, после некоторых мучений, сравните с тем, что показано ниже.

```
{ P_29_1 – Полицейская база данных, версия 1 }

function FindNumber(var aFile: text; aNumber: integer): boolean;
var N: integer;   { текущий номер в БД }
begin
      FindNumber:= false;     { на случай, если файл пуст  }
      Reset(aFile);     { позицию чтения устанавливаем в начало файла }
      N:=0;             { в начале цикла задаем несуществующий номер }
      { читаем номера из файла, пока НЕ конец файла И номер НЕ найден }
      while not Eof(aFile) and (N<>aNumber) do begin
            Readln(aFile, N);
            FindNumber:= (N=aNumber);     { true, если номер нашелся }
      end;
end;
var   F: text;    Num: integer;
begin {----- Главная программа -----}
      Assign(F, 'Police.txt');
      repeat
            Write('Укажите номер автомобиля: ');     Readln(Num);
            if FindNumber(F, Num)
                  then Writeln('Эта машина в розыске, хватайте его!')
                  else Writeln('Пропустите его');
      until Num=0;  { 0 – признак завершения программы}
      Close(F);
end.
```

Поясню некоторые моменты. В начале главной программы файловая переменная **F** связывается с файлом «Police.txt». Далее следует хорошо знакомая конструкция **REPEAT–UNTIL** с проверкой условия в конце цикла.

Самое интересное скрыто внутри функции **FindNumber** (от Find — «искать», Number — «номер»). Туда передаются два параметра, один из которых — файловая переменная. Обратите внимание на способ её передачи: файловая переменная передается по ссылке (в заголовке указано слово **VAR**). И никак иначе файловую переменную не передают! Со временем узнаете причину, а пока просто запомните: файловые переменные передают внутрь процедур и функций только по ссылке! Следовательно, параметр **aFile** ссылается на глобальную переменную **F**.

А к чему здесь приставки «a» перед именами параметров: **aFile**, **aNumber**? Или это тоже правило языка? Нет, друзья, это всего лишь уловка программистов, которую полезно перенять. Чем сложнее будут ваши программы, тем гуще будут заселены разного рода переменными и параметрами. Во избежание путаницы лучше учредить разумную систему обозначений. Большинство программистов используют для систематизации имен так называемые префиксы или приставки. Например, для параметров (аргументов) процедур и функций применяют префикс «a» (от слова «argument»). Помеченные таким образом параметры уже не спутаешь с локальными или глобальными переменными.

Теперь заглянем внутрь функции **FindNumber**. В первой строке результату функции присваивается значение **FALSE**. И это оправдано, поскольку значение функции обязательно должно быть определено, а в случае, если файл БД окажется пустым, этого не случится, поскольку следующий далее цикл **WHILE** не будет выполняться.

Поиск номера должен начинаться с начала файла. Оператор **Reset** внутри функции как раз и возвращает позицию чтения на линию старта. Будьте спокойны — файл от этого не пострадает, открывать его для чтения можно без ограничений!

Теперь взгляните на условие цикла **WHILE**, — оно чуть сложнее тех, к которым мы привыкли.

```
while not Eof(aFile) and (N<>aNumber)
```

Наряду с признаком конца файла проверяется и условие несовпадения искомого номера с номером, прочитанным из файла. Значит, цикл будет продолжаться, пока НЕ достигнут конец файла И НЕ найден искомый номер.

Внутри цикла находим непривычный оператор присваивания.

```
FindNumber:= (N=aNumber);    { true, если номер нашелся }
```

Левая его часть — это идентификатор функции, а правая (в скобках) — оператор сравнения двух чисел. Оператор сравнения дает булев результат, равный **TRUE**, если числа совпадают. Скобки в правой части здесь не нужны, но я поставил их для наглядности. Приведенный выше оператор можно было бы заменить таким.

```
     if N=aNumber
           then FindNumber:= true
           else FindNumber:= false;
```

Но, согласитесь, первый вариант нагляднее и короче.

Итак, прежде чем двинуться дальше, не поленитесь проверить эту программу.

Полицейская база данных, версия 2

Теперь слегка изменим расположение чисел в файле «Police.txt». Вместо одного числа в строке, напечатайте в каждой по нескольку чисел, разделив их одним или несколькими пробелами, например:

```
123 234 325
223 240
845 431 205
```

Подобное расположение данных вполне обычно, взгляните хотя бы в классный журнал, где в одной строке проставлен ряд оценок. Мы, программисты, должны извлекать данные даже из таких файлов.

Переключитесь в окно нашей базы данных «Police.txt» и внесите необходимые изменения. Сохранить файл не забыли? Теперь запустите программу и проверьте её на номерах из этого файла. При должном внимании вы обнаружите, что программа правильно находит только числа, начинающие строку, например 123, 223 и 845. Все последующие номера в строке программа не замечает, хотя и аварийных сообщений не выдает. В чем же дело?

Причина — в процедуре **`Readln`**. До сих пор мы пользовались ею для чтения строк и горя не знали. Но числа — иное дело. Приглашаю вас мысленно проследить за позицией чтения в ходе просмотра нашей БД (рис. 66). Невидимые признаки конца строки обозначены на рисунке условными символами **`Eoln`** (это пара символов с кодами 13 и 10).

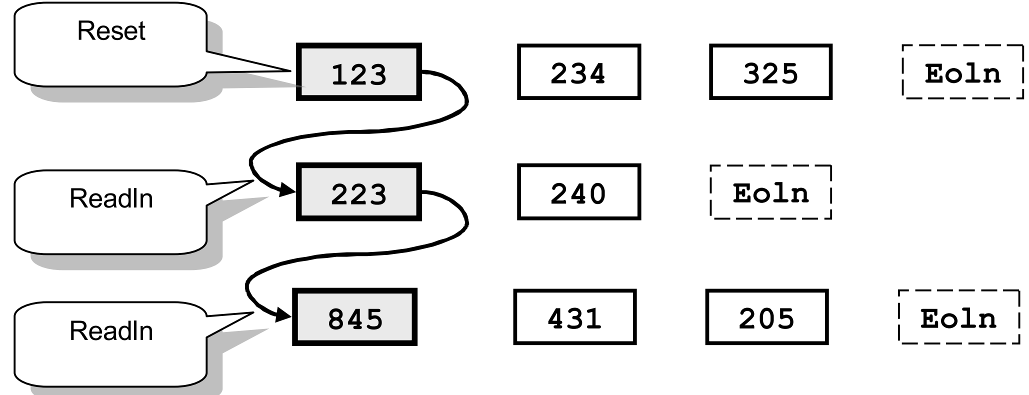


Рис. 66 – Продвижение позиции чтения процедурой Readln

После того, как **Reset** установит позицию чтения в начало файла, процедура **Readln** прочитает первое число. Чтение идет цифра за цифрой, пока не встретится любой символ, отличный от неё, например, пробел или конец строки. Проглотив таким образом первое число, процедура **Readln** продвинет позицию чтения в начало следующей строки, пропуская при этом всё, что расположено до конца текущей. Вот в чем дело! Не зря к названию процедуры прилепился суффикс «LN» (сокращенное от Line — «строка»). Источник проблемы ясен: процедура **Readln** не подходит для чтения нескольких чисел в строке. Где же выход?

Спокойно, друзья, в Паскале заготовлено всё! Познакомьтесь с процедурой **Read** (без суффикса), которая почти не отличается от своей «сестренки» — принимает те же параметры и читает те же данные. Но при этом самовольно не продвигает позицию чтения в начало следующей строки. А нам того и нужно! Продвижение позиции чтения процедурой **Read** показано на рис. 67.

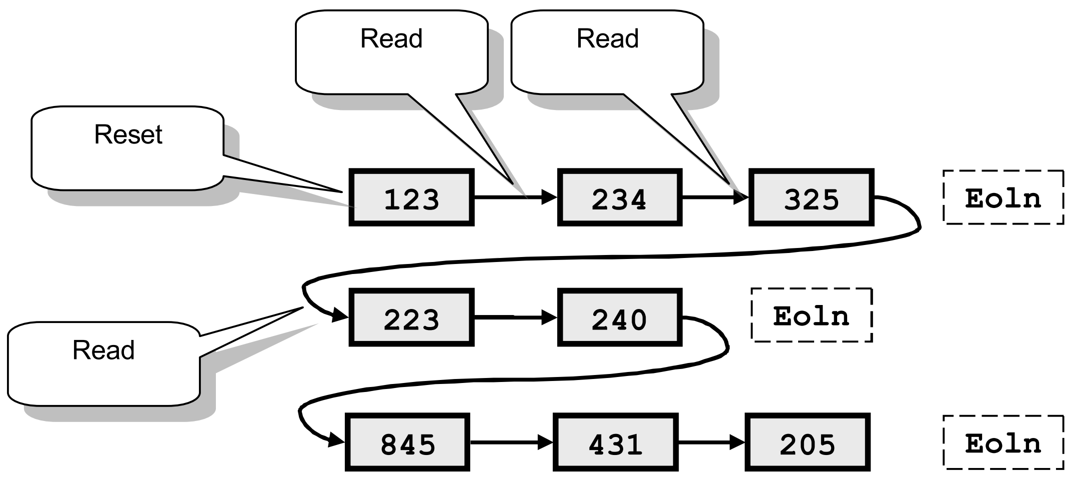


Рис. 67 – Продвижение позиции чтения процедурой Read

После чтения каждого числа позиция продвигается за это число и остаётся там. Следующий вызов процедуры прочитает очередное число в этой же строке и так далее, пока не будет достигнут конец строки. А потом? Потом позиция

сдвинется в начало следующей строки, и чтение продолжится тем же чередом до конца файла.

Итак, решение найдено, и теперь для правильной работы программы надо лишь удалить суффикс в имени процедуры, то есть заменить вызов

```
    Readln(aFile, N);
```

на вызов

```
    Read(aFile, N);
```

Внесите это исправление в программу, сохраните её под именем P_29_2 и проверьте, работает ли она.

Предвижу законный вопрос: к чему в языке две похожие процедуры, нельзя ли обойтись только **Read**? Нет, нельзя, — процедура **Readln** незаменима при построчной обработке файлов, и очень скоро вы в этом убедитесь.

Итоги

- База данных – это организованное хранилище информации, приспособленное для быстрого её поиска. Простейшую базу данных можно создать редактором текста.
- Перед поиском данных в текстовом файле, надо установить позицию чтения в начало файла процедурой **Reset**.
- Процедура **Readln** после чтения затребованных данных продвигает позицию чтения в начало следующей строки. При этом все непрочитанные данные в текущей строке пропускаются.
- Процедура **Read** после чтения затребованных данных продвигает позицию чтения за последний прочитанный элемент. Она подходит для последовательного чтения данных без учета разбивки на строки.

А слабо?

А) Напишите программу для преобразования второго варианта базы данных «Police.txt» (с несколькими числами в строке) в первый вариант (по одному числу в строке). Или слабо?

Б) Можно ли в решении предыдущей задачи назначить одно и то же имя как входному, так и выходному файлам? Испытайте на практике.

Глава 30
Журнальная история

Статистика знает всё?

Давно ли вы заглядывали в классный журнал? Хорошо бы делать это раньше своих родителей. Ах, если бы журнал можно было и править!.. Нет, я не подбиваю вас подтирать оценки! Мы всего лишь подвергнем журнал статистической обработке. Статистика — это наука, находящая закономерности в массе данных. Она как тот холм, взобравшись на который, за деревьями видишь лес. Нужен пример? Пожалуйста.

В некой школе некоторого царства-государства для сравнения учеников и классов учредили рейтинги. Что такое рейтинг? — это вроде места в турнирной таблице. Чем выше рейтинг, тем сильнее спортсмен или команда, то есть ученик или класс. Определять рейтинг условились по средней оценке ученика или всего класса. Так, совокупность многих оценок заменялась одним числом — средним баллом. Когда вместо десятков чисел получаешь одно, — это и есть плод статистической обработки.

Вычисление средних оценок возложили на компьютер, заказав для этого программу. Входные данные для нее извлекались из журнала, который велся в виде текстового файла. Вот как выглядел входной файл, то есть классный журнал.

```
Акулова      3 5 4
Быков        5 5 5 5
Воронов      4 5 5 4
Галкина      3 4 3
Крокодилкин  4 3
```

А вот что получалось после обработки его упомянутой программой.

```
Номер   Фамилия        Количество    Сумма     Средний
                         оценок      баллов     балл
  1     Акулова            3           12        4.0
  2     Быков              4           20        5.0
  3     Волков             4           18        4.5
  4     Галкина            3           10        3.3
  5     Крокодилкин        2            7        3.5
```

Стало быть, средний балл вычислялся как частное от деления суммы баллов на количество оценок, а результат записывался с одним знаком после запятой.

И всё было хорошо, пока вирусная атака не уничтожила бесценную программу. А где распечатка исходника? Увы, к тому времени её погрызли мыши! Друзья, теперь надежда только на вас, выручайте школу!

Строим планы

С первого взгляда на задачу ясно: входной файл надо обрабатывать построчно, выбирая из каждой строки данные двух типов: строковые — фамилии учеников, и числа — их оценки. Это не так просто, как может показаться, а потому решать задачу будем в два счёта. На первом этапе упростим её, оставив во входном файле лишь оценки учеников, а полное решение отложим до следующей главы. Входной файл без фамилий будет теперь таким.

```
3 5 4
5 5 5 5
4 5 5 4
3 4 3
4 3
```

А в результате обработки мы должны получить такой выходной файл.

```
Номер     Количество     Сумма     Средний
            оценок       баллов     балл
  1           3            12        4.0
  2           4            20        5.0
  3           4            18        4.5
  4           3            10        3.3
  5           2             7        3.5
```

Набросаем план предстоящего сражения, то есть блок-схемы алгоритмов. На рис. 68 показан алгоритм главной программы. Он очень похож на тот, что применялся при шифровании текста, и это объяснимо: и там, и здесь выполняется построчная обработка файла.

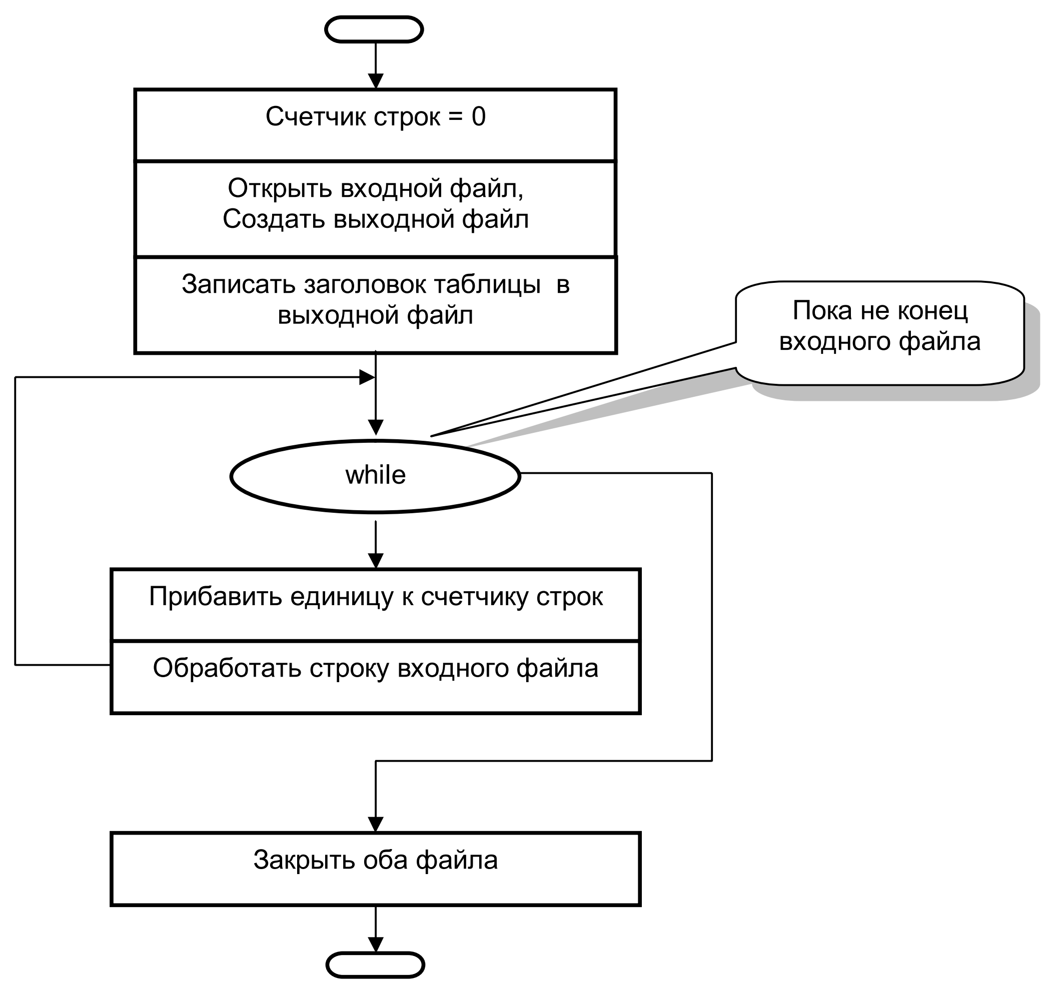


Рис. 68 – Алгоритм главной программы

Рванув со старта, программа открывает входной файл, создает выходной и пишет туда заголовок таблицы — так называемую шапку. По окончании обработки оба файла будут закрыты. В этом алгоритме предусмотрен также и подсчет строк входного файла, необходимый для нумерации учеников в выходном файле.

Разобравшись с главной программой, сосредоточимся на обработке отдельной строки. Здесь заметно сходство со вторым вариантом полицейской базы данных (глава 29). И там, и тут читается ряд чисел, размещенных в одной строке. Но если в полицейской программе нам было безразлично, где кончается строка, то теперь иное дело, — ведь на следующей строке расположены оценки другого ученика! Нужен признак, сообщающий о достижении конца читаемой строки. Где его взять?

Ну, вы же понимаете, — в Паскале предусмотрено всё! Познакомьтесь с функцией булевого типа по имени **EoLn** (от английского End of Line, что значит «конец строки»). Заголовок этой функции выглядит так.

```
      function Eoln(var aFile: text): boolean;
```

Функция принимает параметр — ссылку на текстовый файл — и возвращает **TRUE**, если позиция чтения в этом файле достигла конца строки. Она похожа на функцию **Eof**, проверяющую достижение конца файла. Исследуем функцию следующей программкой.

```
{----- Программа для исследование функции Eoln -----}
var F: text;  N: integer;
begin
      Assign(F, 'Police.txt');   Reset(F);
      while not Eof(F) do begin
            Read(F, N); { чтение числа }
            Writeln(N, ' -- ', Eoln(F)); { печать признака конца строки }
      end;
      Close(F);   Readln;
end.
```

Здесь из файла «Police.txt» читаются все числа, при этом печатаются и сами числа, и признак конца строки. Предположим, файл «Police.txt» содержал такие строки.

```
1 2 3
10 20 30
100 200 300
```

Тогда на экране явится вот что.

```
1 -- FALSE
2 -- FALSE
3 -- TRUE
10 -- FALSE
20 -- FALSE
30 -- TRUE
100 -- FALSE
200 -- FALSE
300 -- TRUE
```

Как видите, после чтения последнего числа в строке признак её окончания равен **TRUE**.

Теперь, когда мы нащупали конец строки, соорудим алгоритм обработки одной строчки входного файла (рис. 69).

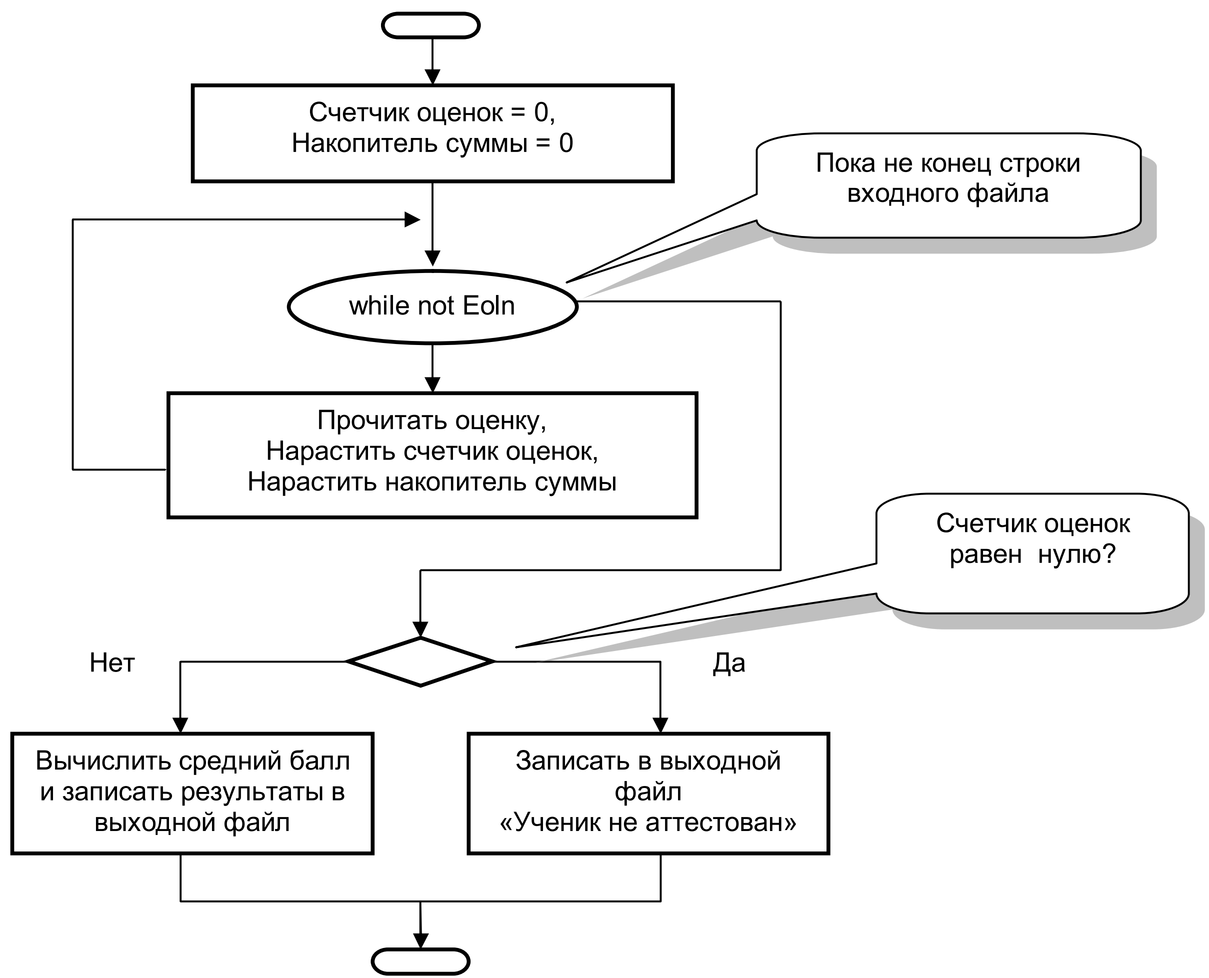


Рис. 69 – Блок-схема обработки одной строки входного файла

Основу алгоритма составляет цикл чтения чисел, в ходе которого наращиваются счетчик оценок и их сумма. На входе в цикл и счетчик, и сумма очищаются, то есть загружаются нулями. Выход из цикла происходит при достижении конца строки, и тогда в бой вступает условный оператор. Он исследует счетчик оценок, и, если оценок в строке не было, выводит в выходной файл сообщение «ученик не аттестован», а иначе печатает средний балл.

Кстати, годится ли для чтения строки оператор цикла **REPEAT-UNTIL**? Правильный ответ — нет. Если текущая строка окажется пустой, следующая оценка прочитается уже на следующей строке, а это не то, что нам нужно! Оператор **WHILE** — единственно правильное решение.

Барабаним по клавишам

Теперь всё готово для сочинения задуманной программы P_30_1, ниже показан её текст. В начале программы объявлены три глобальные переменные: две — для доступа к входному и выходному файлам, и одна — для подсчета читаемых строк. Поскольку эти переменные объявлены перед процедурой обработки строки **HandleString**, то будут видны и в этой процедуре. Поэтому передавать содержащиеся в них данные через параметры здесь не обязательно (но возможно).

Таким образом, мы передаем данные в процедуру через глобальные переменные, — в небольших программах это допустимо. Только не злоупотребляйте этим приемом, иначе в сложных программах запутаетесь.

Заглянем теперь внутрь процедуры **HandleString**. Кстати, её название составлено из двух слов: Handle — «обработка», и String — «строка». Процедура не принимает параметров, поскольку все необходимые данные получает через глобальные переменные. Обратите внимание на вычисление среднего балла:

```
      Rating:= Sum div Cnt;
```

Сумма баллов делится на счетчик оценок; в этой операции участвуют целочисленные переменные, а результат деления тоже получится целым. Стало быть, дробную часть рейтинга мы теряем, — к решению этой проблемы вернемся позже.

А что сказать о главной программе? Она работает по ранее рассмотренному алгоритму. По крайней мере, сейчас нам так кажется.

```
{ P_30_1 – обработка журнала, первый вариант }
      {----- Глобальные переменные -----}
var   InFile, OutFile : text; { входной и выходной файлы }
      Counter: integer;       { счетчик строк входного файла }
      {----- Процедура обработки одной строки -----}
procedure HandleString;
var   N  : integer;     { оценка, прочитанная из файла }
      Cnt: integer;     { количество оценок }
      Sum: integer;     { сумма баллов }
      Rating: integer;  { средний балл }
begin
      Sum:=0; Cnt:=0; { очищаем накопитель и счетчик оценок }
      while not Eoln(InFile) do begin  { пока не конец строки }
            Read(InFile, N);        { читаем оценку в переменную N }
            Sum:= Sum+N;            { накапливаем сумму баллов }
            Cnt:= Cnt+1;            { наращиваем счетчик оценок }
      end;
```

```
      if Cnt>0
            then begin        { если оценки были }
                  Rating:= Sum div Cnt;
                  Writeln(OutFile, Counter, Cnt, Sum, Rating);
            end
            else  { а если оценок не было }
                  Writeln(OutFile, Counter, '  Ученик не аттестован');
end;

      {----- Главная программа -----}
begin
      Counter:= 0;      { обнуляем счетчик строк }
      { открываем входной файл }
      Assign(InFile,'P_30_1.in');   Reset(InFile);
      { создаем выходной файл }
      Assign(OutFile,'_P_30_1.out');      Rewrite(OutFile);
      { выводим шапку таблицы }
      Writeln(OutFile, 'Номер     Количество     Сумма      Средний');
      Writeln(OutFile, 'ученика     оценок       баллов      балл');
      { пока не конец входного файла... }
      while not Eof(InFile) do begin
            Counter:= Counter+1;    { наращиваем счетчик строк }
            HandleString;           { обрабатываем строку }
      end;
      { закрываем оба файла }
      Close(InFile);  Close(OutFile);
end.
```

Скомпилировали программу? Тогда подготовьте входной файл с оценками учеников (без фамилий). Назовите его «P_30_1.in» и сохраните, как обычно, в рабочем каталоге. Можно запускать? В общем-то, да. Но будьте настороже, — ошибки караулят нас на каждом шагу! Возьмите за правило первый прогон своих программ выполнять в пошаговом режиме. Поступим так и в этот раз.

Первый блин

Переключитесь в окно программы и для исполнения одного шага нажмите клавишу *F7*. Продолжайте нажимать её, наблюдая за ходом выполнения операторов. При желании добавьте в окно обзора переменных «Watch» глобальную переменную **`Counter`**.

После нескольких шагов вы попадете внутрь процедуры **HandleString** и благополучно пройдете по ней. На следующем цикле главной программы вновь войдете туда же, но теперь вас ждут «чудеса». Во-первых, цикл

```
while not Eoln(InFile)
```

уже не выполняется ни разу, и в выходной файл пишется «Ученик не аттестован». Ещё загадочней другой фокус: не выполняется условие выхода из цикла в главной программе.

```
while not Eof(InFile)
```

Обработав несколько строк, программа почему-то продолжает фанатично работать и дальше; она, как говорится, зациклилась. Если бы мы запустили её сразу в непрерывном режиме, то захламили бы выходным файлом весь диск! Пришлось бы аварийно снимать программу диспетчером задач. Ясно, что где-то притаилась ошибка, и надо прервать выполнение программы. Сделать это в пошаговом режиме несложно, — нажмите комбинацию *Ctrl+F2*.

Теперь сядем на пенёк и поразмыслим, в чем дело? Ведь первый вход в процедуру отработан верно. Вспомните наше исследование функции **Eoln**: чтение последнего числа в строке устанавливает признак конца строки в **TRUE**. Значит, при повторном входе в процедуру обработки строки цикл **while not Eoln(InFile)** не должен выполняться ни разу, — так оно и происходит. Так вот где собака порылась, — мы застряли в конце первой строки!

Этим же объясняется и зацикливание в главной программе, — не проскочив первой строки, нам не достичь конца файла. Виновник найден? — да, и теперь поищем решение проблемы. После обработки строки нам надо всего лишь перескочить с конца текущей строки в начало следующей, ничего при этом не читая. На ум приходит процедура **Readln**, — ведь она любит это делать. Помните, как подвела она нас во второй версии полицейской базы данных? Зато теперь выручит! Где её вызвать? Сделаем это после выхода из процедуры **HandleString**, то есть в цикле главной программы. Теперь главный цикл будет выглядеть так.

```
while not Eof(InFile) do begin
      Counter:= Counter+1;    { наращиваем счетчик строк }
      HandleString;           { обрабатываем строку }
      Readln(InFile);         { переход на следующую строку }
end;
```

Процедуре **Readln** передана лишь файловая переменная **InFile**, поскольку никаких данных ей читать не надо. Стало быть, для исправления программы добавим лишь один оператор. Сделайте это и запустите программу в пошаговом

режиме. Если пара строк будет обработана правильно, запустите её далее в непрерывном режиме.

Блин второй

Запуская программу, не ждите результатов на экране, — программа отработает молча. Для просмотра результатов откройте выходной файл «P_30_1.out». Сделайте это, не выходя из IDE: просто нажмите *F3* и укажите имя файла. Вам откроется следующая картина.

```
Номер    Количество     Сумма     Средний
           оценок       баллов     балл
13124
24205
34184
43103
5273
```

Что это? Вместо ожидаемых четырех колонок чисел мы видим только одну. Да и числа в ней несуразные! Откуда они взялись? Пробуем разгадать эту головоломку: первая цифра совпадает с порядковым номером строки: 1, 2, 3 и так далее. Вторая равна количеству оценок ученика: 3, 4, 4, 3, 2. Ага, значит, результаты правильные, только «слиплись» в одно число, — между числами нет пробелов. Кто виноват? Нет, не мы с вами, а этот оператор.

```
      Writeln(OutFile, Counter, Cnt, Sum, Rating);
```

Его параметры разделены запятыми, и потому печатаются подряд, без пробелов. Вставить пробелы можно, например, так.

```
      Writeln(OutFile, Counter,'  ',Cnt,'  ',Sum,'  ', Rating);
```

Тут между числами печатаются три строковые константы, состоящие из пробелов, они и будут разделять числа между собой. Количество пробелов можно рассчитать заранее или подобрать опытным путем. Но есть лучшее решение.

Спецификатор ширины поля

Лучшее решение дают спецификаторы ширины поля. Это числовые выражения, задающие количество позиций для печати параметра. Их указывают за печатаемым параметром, причем параметр и спецификатор разделяются двоеточием, например:

```
W := 15;
Writeln(OutFile, Cnt : 10, Sum : W);
```

Здесь значение переменной **Cnt** будет напечатано на десяти символьных позициях, а значение переменной **Sum** — на пятнадцати (соответственно значению **W**).

Спецификаторы — это дополнительные параметры процедуры печати, придающие красоту выводимым результатам. Когда число требует меньше места, чем задано в спецификаторе, лишние позиции заполнятся пробелами, а нам того и надо. К тому же, спецификатор может выравнивать выводимые числа по левому, или правому краю колонки: положительное его значение выравнивает число по правому краю, а отрицательное — по левому. Спецификаторы применяют и к строковым выражениям.

Для нашего случая я подобрал следующие значения спецификаторов.

```
Writeln(OutFile, Counter:3, Cnt:13, Sum:14, Rating:12);
```

Исправьте заодно и этот оператор вывода в файл:

```
Writeln(OutFile, Counter:3, '  Ученик не аттестован');
```

Снова запустите программу и проверьте результат. Кстати, если вы не закрывали окно с выходным файлом «P_30_1.out», то по завершении программы IDE сообщит о том, что файл на диске был изменен, — это сделала ваша программа. Но в открытом окне всё осталось по-прежнему, потому IDE спрашивает разрешение на обновление окна, — дайте положительный ответ кнопкой «Yes» и переключитесь в окно с файлом «P_30_1.out». Теперь вы увидите вот что.

```
Номер     Количество       Сумма       Средний
            оценок         баллов       балл
  1           3              12           4
  2           4              20           5
  3           4              18           4
  4           3              10           3
  5           2               7           3
```

Это почти идеальный результат. Осталась лишь одна шероховатость, — средний балл не содержит дробной части. Займемся этим вопросом.

«Развесные» числа

Обратимся к строке программы, где вычисляется средний балл.

```
Rating:= Sum div Cnt;
```

Здесь одно целое число делится на другое, и результат тоже получается целым. Куда же девается дробная часть? Увы, дробная часть отбрасывается. Куда отбрасывается, не знаю, но она теряется. Поэтому при целочисленном делении получаются, например, такие забавные результаты.

```
7 div 3 = 2
7 div 4 = 1
7 div 5 = 1
```

Целые числа потому и целые, что не дают «отколоть» от себя ни крошки. Они как штучный товар. Но средний балл — это «развесной товар», для него нужны другие числа, и они в Паскале есть.

Я говорю о вещественных числах. В Паскале есть несколько типов для представления таких чисел. Один из них — **REAL** — родной для Паскаля, поскольку существовал в первой версии языка. Другие добавились с появлением в компьютерах математических сопроцессоров. Для хранения среднего балла воспользуемся типом **REAL**; с этой целью изменим объявление переменной **Rating** следующим образом:

```
var Rating: Real;
```

Но этого недостаточно. Дело в том, что, если мы оставим формулу

```
Rating:= Sum div Cnt;
```

без изменений, то и результат не изменится. Всё потому, что операция **DIV** (от Division — «деление») предназначена только для целых чисел, и дробную часть она всё равно отбросит. Для деления вещественных чисел в Паскале есть другая операция, она записывается косой чертой «/». Значит, упомянутый выше оператор мы должны изменить так.

```
Rating:= Sum / Cnt;
```

Вот теперь должно заработать! Запустив новый вариант программы и открыв выходной файл, вы найдете вот что.

```
Номер    Количество      Сумма      Средний
           оценок        баллов       балл
  1          3             12 4.00000000000000E+0000
  2          4             20 5.00000000000000E+0000
  3          4             18 4.50000000000000E+0000
  4          3             10 3.33333333333212E+0000
  5          2              7 3.50000000000000E+0000
```

Что бы это значило? Средний балл считается верно, но печатается очень странными уродливыми числами! Не пугайтесь, перед вами научный формат представления вещественного числа, он удобен для изображения очень маленьких и очень больших чисел. Например, известное физикам и химикам число Авогадро (примерно $6{,}022140 \cdot 10^{23}$) изображается как 6.022140E+0023. Но нам этот формат не подходит, и мы заменим его, задав спецификатор ширины поля.

Для вещественных чисел спецификатор состоит из двух частей, разделяемых двоеточием. Первая часть задает общую ширину поля печати (так же, как и для целых чисел), а вторая — количество цифр после запятой (после точки). Чтобы напечатать переменную **Rating** с одним знаком после точки при общей ширине поля в 12 позиций, нам следует применить такой оператор печати.

```
     Writeln(OutFile, Counter:3, Cnt:13, Sum:14, Rating:12:1);
```

Теперь вновь запустим программу и полюбуемся на результат.

```
Номер    Количество      Сумма      Средний
           оценок        баллов       балл
  1          3             12         4.0
  2          4             20         5.0
  3          4             18         4.5
  4          3             10         3.3
  5          2              7         3.5
```

Прекрасно! Изрядно потрудившись и одолев ряд ошибок, мы достигли цели! Осталось лишь подытожить завоевания этой главы.

Итоги

- Функция **Eoln** следит за признаком конца текущей строки, применяется совместно с оператором **WHILE**.
- Для продвижения позиции чтения в начало следующей строки вызывайте процедуру **Readln**, указывая лишь один параметр – файловую переменную.

- Данные внутрь процедур и функций можно передавать через глобальные переменные; такой прием допустим для несложных программ.
- Целые числа не содержат дробной части. Для действий с дробными числами применяют вещественные типы, например, **`Real`**.
- Для получения дробного результата деления пользуйтесь операцией «/» (косая черта). Операция **`DIV`** при делении отбрасывает дробную часть.
- Для ровной печати чисел применяйте спецификаторы ширины поля.

А слабо?

А) Функция **`Trunc`** выделяет целую часть вещественного числа, например:

```
    Writeln (Trunc( 12.345 ));    { 12 }
```

Исследуйте её и придумайте способ выделения дробной части вещественного числа. Напишите подходящую функцию и программу для её проверки.

Б) Объясните и проверьте, что напечатает следующая программа.

```
var N: integer;
begin      for N:=1 to 20 do Writeln (' ':N, N);    end.
```

В) Сформируйте файл «Numbers.txt», поместив в него 100 случайных чисел в диапазоне от 0 до 999 (некоторые числа могут повторяться). Затем найдите в этом файле: 1) максимальное и минимальное число; 2) сумму всех чисел; 3) среднее арифметическое — напечатайте его с двумя знаками после точки.

Г) Сканирование марсианской поверхности дало файл, содержащий высоту отдельных его точек вдоль одного из направлений, — пусть это будет файл «Numbers.txt» из предыдущей задачи. Найдите точки, где вероятней всего обнаружить марсианскую воду. На следующем ниже рисунке они обозначены буквами W. Программа должна напечатать две колонки: порядковый номер точки относительно начала файла (счет от нуля) и высоту точки (такие точки математики называют локальными минимумами).

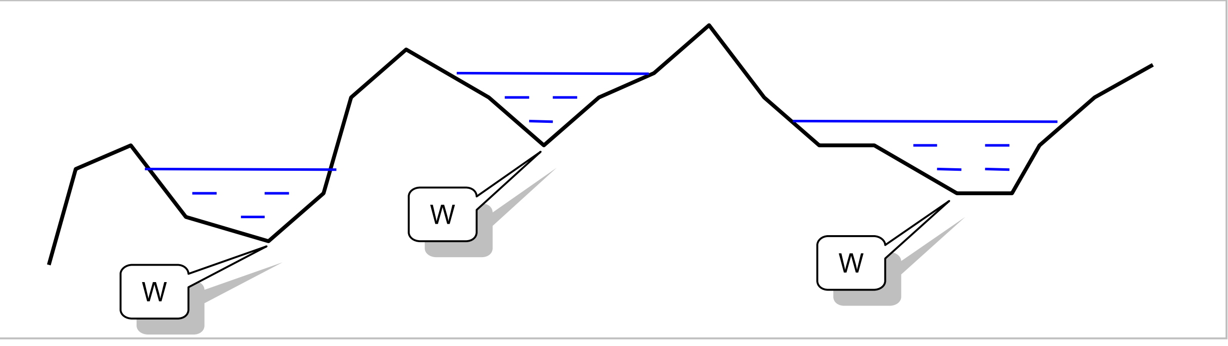


Рис. 70 - Рельеф марсианской поверхности

Глава 31
Финал журнальной истории

В предыдущей главе мы поклялись восстановить съеденную мышами программу и отчасти сдержали клятву. Нами решена упрощенная задача — обработка журнала без фамилий учеников, то есть, мы исполнили вычислительную часть проекта. Теперь завершим его, добившись обработки настоящего классного журнала. Требуется, казалось бы, пустяк — прочесть фамилии учеников. Но воспользоваться процедурой **Readln**, как мы поступили в программе шифрования текста, здесь не получится, — она прочитает всю строку целиком, включая и оценки (которые станут как бы частью фамилии!).

Буква за буквой

Славный литературный герой Остап Бендер по поводу желанного миллиона сказал так: «Я бы взял частями, но мне нужно сразу!». Увы! При чтении фамилий надо проявить терпение. Если не получается сразу, возьмем по частям. Ведь строка фамилии состоит из отдельных букв, — так прочитаем фамилию по буквам! Прочитать букву может всё та же процедура **Read**, например:

```
var sym : char;
. . .
      Read(InFile, sym);      { чтение одного символа }
```

А фамилию **S** склеим из отдельных букв:

```
      S:= S + sym;
```

Разумеется, что здесь нужен цикл, условием выхода из которого будет либо достижение первого пробела, либо достижение конца строки. В этом и состоит основная идея алгоритма, показанного на рис. 71.

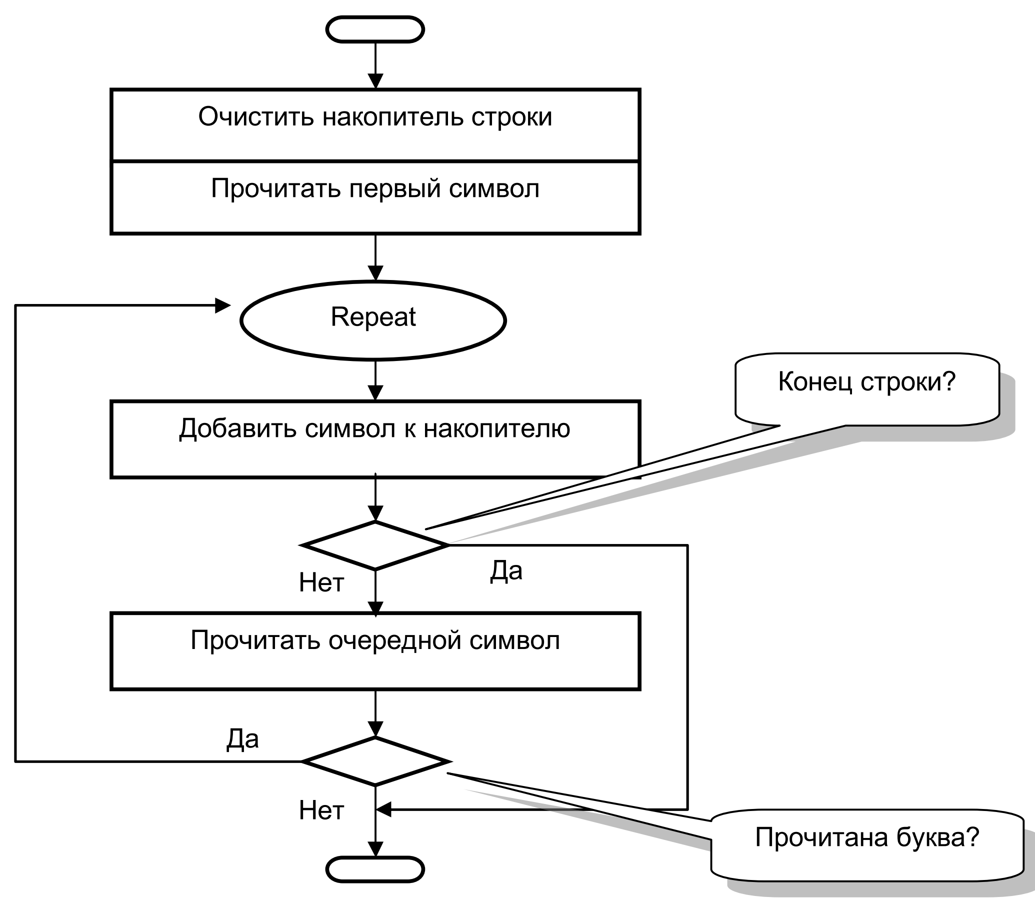


Рис. 71 – Упрощенный алгоритм побуквенного чтения фамилии

Нелишняя предосторожность

Людям свойственно ошибаться, — даже учителям! В строках журнала (а это текстовый файл) могут оказаться лишние пробелы — как между оценками, так и в начале строки, перед фамилией. И что тогда? — проверьте на практике. При чтении чисел процедура **`Read`** «не заметит» лишних пробелов, — она достаточно «умна». Другое дело — показанная выше блок-схема: если перед фамилией обнаружится пробел, то чтение слова завершится досрочно. Стало быть, для правильного чтения фамилии надо пропустить стоящие перед нею пробелы (если они есть). Это улучшение слегка усложнит блок-схему (рис. 72).

Примечание. Лишние пробелы в конце строк (после оценок) тоже приведут к аварии программы, проследите, чтобы во входном файле их не было. Проблема концевых пробелов решается заменой вызовов функций **`Eoln`** и **`Eof`** соответственно вызовами функций **`SeekEoln`** и **`SeekEof`**. Эти функции дают **`TRUE`**, даже если между текущей позицией и концом строки (файла) располагаются несколько пустых символов: пробелов, табуляций или пустых строк (для **`SeekEof`**). Улучшить программу P_31_1 предлагаю вам самостоятельно.

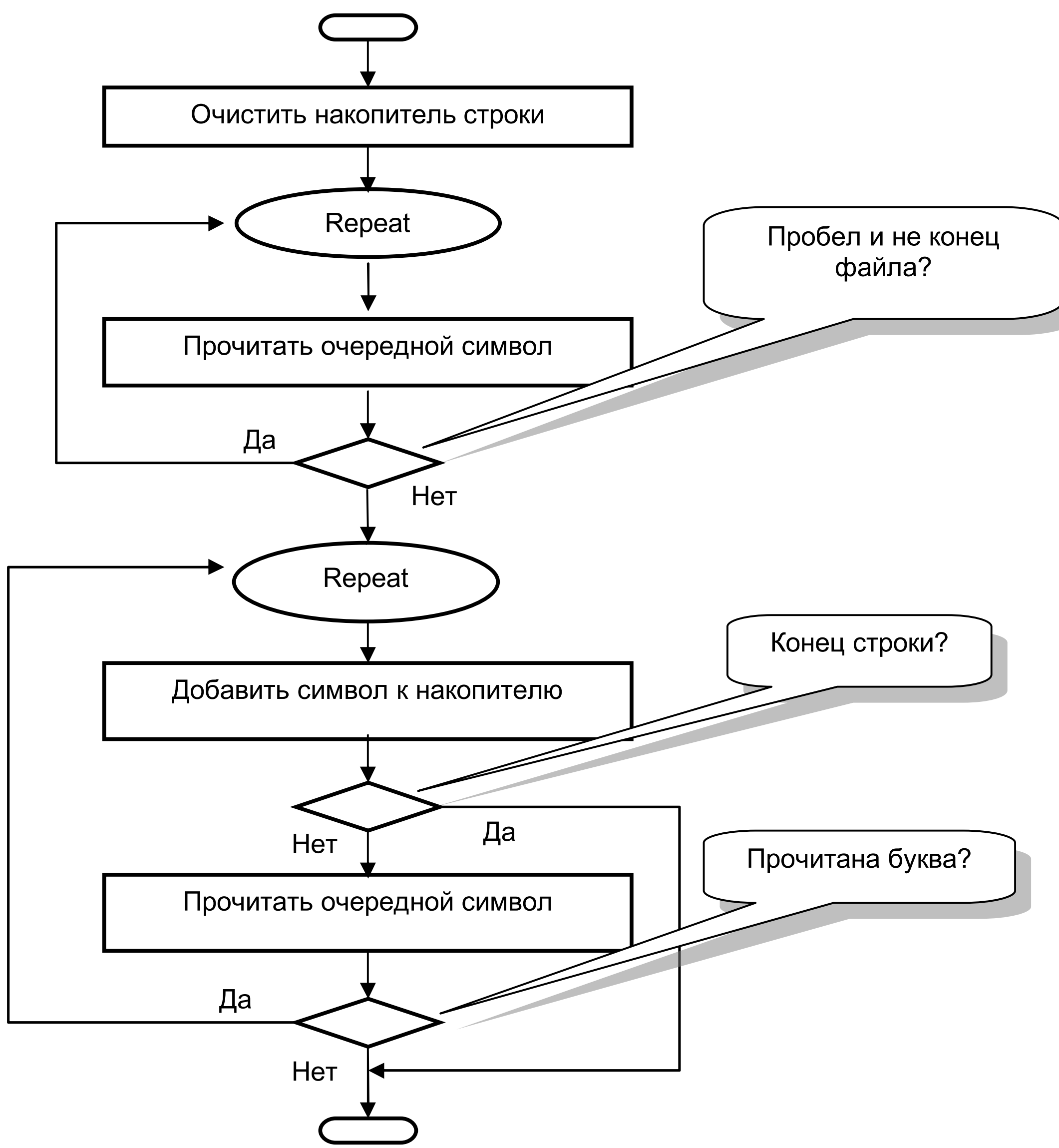


Рис. 72 – Усовершенствованный алгоритм побуквенного чтения фамилии

Достройка программы

В основу новой версии программы P_31_1 положим программу P_30_1. Вам следует, прежде всего, открыть её и сохранить под новым именем. Готово? Тогда приступаем к правке.

Начнем с главной программы, где надо изменить имена входных и выходных файлов (чтобы не путать с похожими файлами предыдущей версии).

```
Assign(InFile,'Journal2.in');       Reset(InFile);
Assign(OutFile,'Journal2.out');     Rewrite(OutFile);
```

Позаботьтесь о том, чтобы файл «Journal2.in» был похож на настоящий классный журнал с фамилиями, как о нём сказано в начале 30-й главы.

Второе изменение внесем в процедуру обработки строки **HandleString**. Здесь объявим ещё одну переменную строкового типа, назовем её **Fam**, она будет вмещать фамилию ученика.

```
Fam:= ReadFam; { читаем фамилию }
```

Разумеется, оператор печати строки тоже будет изменен.

```
Writeln(OutFile, Counter:3, Fam:18, Cnt:8,     Sum:14, Rating:11:1);
```

Осталось выяснить, что такое **ReadFam**? Это функция чтения фамилии, которую мы напишем по рассмотренному чуть выше алгоритму (рис. 72). Мой вариант функции таков.

```
function ReadFam: string;
var sym: char;    { очередной символ }
    S : string;   { накопитель строки }
begin
  S:=''; { очистка накопителя строки }
  { чтение символов до первой буквы }
  repeat Read(InFile, sym) until (Ord(sym)>32) or Eof(InFile);
  { чтение последующих символов }
  repeat
    if Ord(sym) > 32 then s:= s+sym;
    if Eoln(InFile) then Break;
    Read(InFile, sym);
  until Ord(sym) <= 32;
  ReadFam:= S;    { возвращаемый результат }
end;
```

Обратите внимание на сравнение введенного символа с пробелом. Это сравнение можно было бы записать так.

```
sym <> ' '
```

Но пробел в кавычках трудно разглядеть. Лучше сравнивать код символа с кодом пробела (который равен 32), что и сделано внутри функции.

Испытание

Теперь всё готово, запустите программу. Что оказалось в выходном файле «Journal2.out»? Наверное, вот это:

```
Номер     Фамилия        Количество      Сумма       Средний
ученика                    оценок        баллов       балл
  1       Акулова        3             12         4.0
  2       Быков        4             20         5.0
  3       Волков        4             18         4.5
  4       Галкина        3             10         3.3
  5       Крокодилкин        2             7         3.5
```

Если не считать кривых колонок, неплохо. Кривизну даёт разная длина фамилий учеников. Можно выровнять колонки, вычисляя спецификатор ширины в зависимости от длины фамилии. Или поступить иначе, — дополнить фамилии до одинаковой длины пробелами справа, например:

```
      while Length(Fam) < 12 do Fam:= Fam + Char(32);
```

Этот оператор уместен после чтения фамилии. Окончательный вариант программы со всеми дополнениями и уточнениями представлен ниже.

```
      { P_31_1 - Обработка классного журнала, второй этап }

var   InFile, OutFile : text; { входной и выходной файлы }
      Counter: integer;       { счетчик строк в файле }

    {----- Функция чтения фамилии -----}

function ReadFam: string;
var sym: char;
    S : string;
begin
  s:=''; { очистка накопителя строки }
  { пропуск символов до первой буквы }
  repeat Read(InFile, sym) until (Ord(sym)>32) or Eof(InFile);
  { чтение последующих символов фамилии }
  repeat
    if Ord(sym) > 32 then s:= s+sym;
    if Eoln(InFile) then Break;
    Read(InFile, sym);
  until Ord(sym) <= 32;
  ReadFam:= S;
end;
```

```
    {----- Процедура обработки строки -----}

procedure HandleString;
var
  N  : integer;     { оценка, прочитанная из файла }
  Cnt: integer;     { количество оценок }
  Sum: integer;     { сумма баллов }
  Rating: Real;     { средний балл }
  Fam: string;      { фамилия }
begin
  Fam:= ReadFam; { читаем фамилию }
  if Length(Fam)>0 then begin  { если фамилия не пуста, обрабатываем }
    { для выравнивания столбцов добавляем пробелы }
    while Length(Fam) < 12 do Fam:= Fam + ' ';
    Sum:=0; Cnt:=0;  { очищаем накопитель и счетчик оценок }
    While not Eoln(InFile) do begin  { пока не конец строки }
      Read(InFile, N);  { читаем оценку в переменную N }
      Sum:= Sum+N;      { накапливаем сумму баллов }
      Cnt:= Cnt+1;      { наращиваем счетчик оценок }
    end;
    if Cnt>0
      then begin             { если оценки в четверти были }
        Rating:= Sum / Cnt;  { вычисляем и печатаем ср. балл }
        Writeln(OutFile, Counter:3, Fam:18, Cnt:8, Sum:14, Rating:11:1);
      end
      else    { а если оценок не было }
        Writeln(OutFile, Counter:3, Fam:18,' : Ученик не аттестован');
  end
end;

begin    {--- Главная программа ---}
  Counter:= 0;  { обнуляем счетчик строк }
  { открываем входной файл }
  Assign(InFile,'Journal2.in');  Reset(InFile);
  { создаем выходной файл }
  Assign(OutFile,'Journal2.out');  Rewrite(OutFile);
  { выводим шапку таблицы }
  Writeln(OutFile, 'Номер  Фамилия  Количество   Сумма   Средний');
  Writeln(OutFile, '                  оценок       баллов  балл');
```

```
{ пока не конец входного файла... }
  while not Eof(InFile) do begin
    Counter:= Counter+1;  { наращиваем счетчик строк }
    HandleString;         { обрабатываем строку }
    { переход на следующую строку }
    if not Eof(InFile) then Readln(InFile);
  end;
  { закрываем оба файла }
  Close(InFile);  Close(OutFile);
end.
```

Итоги

- Для чтения отдельного слова в строке не годятся ни процедура **`Readln`** (она прочитает всю строку), ни процедура **`Read`**, которая не видит конца строки. Слово читается посимвольно процедурой **`Read`** с отслеживанием признака окончания строки и других условий.
- Строку выходного файла можно формировать порциями, применяя несколько вызовов процедуры **`Write`**. Каждый такой вызов формирует часть строки и продвигает позицию записи, оставляя её в текущей строке. Для перехода к следующей строке вызывается процедура **`Writeln`**.

А слабо?

А) Напишите программу для преобразования первого варианта базы данных «Police.txt» (которая содержит по одному числу в строке) во второй вариант (будет содержать по три числа в строке).

Б) Файл с физическими данными старшеклассников содержит три колонки: фамилия, рост и вес ученика. Создайте программы для решения следующих задач:

- отбор кандидатов для занятий баскетболом, – рост кандидата должен составлять не менее 175 см;
- поиск учеников с избыточным весом, для которых разница между ростом ученика (см) и его весом (кг) составляет менее 100.

Ваши программы должны сформировать соответствующие файлы с фамилиями и данными учеников.

Глава 32
Порядковые типы данных

Вот поле битвы, где там и сям мелькают спины бегущего противника. Разгоряченные боем, наши полки готовы гнать его хоть на край света. Но что это? Зачем полководец прекращает атаку и велит трубить сбор? Поверьте, он знает свое дело: выигрыш битвы — ещё не победа в войне. Предстоят новые сражения, и надо укрепить армию: дать отдых бойцам, накормить, подлечить и вновь построить в боевые порядки.

Освоенные нами элементы Паскаля (считайте их нашим войском) разбили в пух и прах все поставленные задачи. Но, то ли ещё будет! Впереди сильнейший противник. Так соберем свою армию в кулак, соединим всё, что нам известно о Паскале. В этой и двух последующих главах мы детально рассмотрим уже освоенные элементы языка, и в первую очередь — типы данных.

Типы данных: простые и сложные

Кто из вас видел «предметы вообще»? Также не бывает и «данных вообще», — они обязательно принадлежат к тому либо иному типу. Учреждая переменную, параметр или функцию, потрудитесь сообщить их тип компилятору, иначе он не уяснит, сколько памяти отвести для этих данных и что позволено совершать с ними.

На рис. 73 представлены почти все типы данных, встроенные в язык Паскаль. Их принято делить на три категории: простые, сложные и указатели. К настоящему моменту вы знакомы со многими простыми типами данных и одним сложным — строковым типом **`String`**.

Чем разнятся сложные типы от простых? Тем ли, что сложные труднее изучать? Отчасти так, но суть не в этом. Сложные типы обладают внутренней структурой, в которой выделяются отдельные элементы. Так, например, можно выделить отдельные символы в строке. Простые же типы данных не «раскалываются» на мелкие детали.

А указатели? Их применяют для доступа к данным других типов. Указатель чем-то похож на адрес электронной почты или на гиперссылку, в свое время я расскажу об указателях всё.

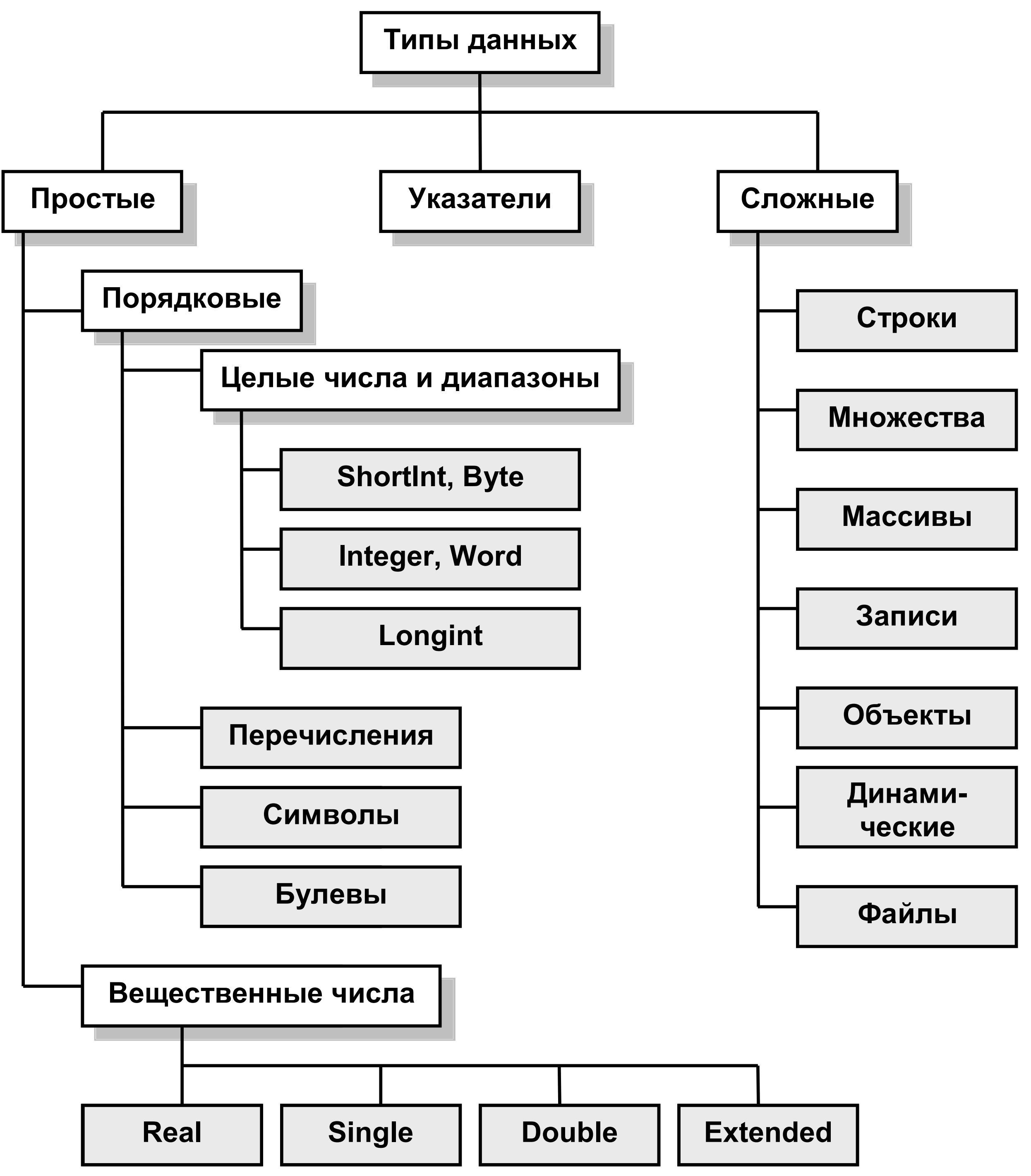


Рис. 73 – Типы данных языка Паскаль

Сейчас направим внимание на простые типы данных с тем, чтобы снять гроздь, висящую на рис. 73 слева. Разобравшись с простыми типами, мы укрепим свой тыл и подготовим атаку на сложные типы данных. В этой главе ознакомимся с общими свойствами порядковых типов данных, а к вещественным обратимся в следующей главе.

Но прежде, чем дать общую характеристику порядковым типам, рассмотрим их по отдельности.

Целое братство

Целые числа образуют дружную «семью» из нескольких братьев. Вам пока знаком лишь один из них — это тип **Integer**, а где остальные? Начнем с двух младших братьев: типов **Byte** и **ShortInt**.

Изучая символы, мы узнали, что они кодируются целыми числами. Основной набор составляют 256 символов с кодами от 0 до 255 включительно. Этот диапазон значений может храниться в ячейке памяти, называемой байтом (BYTE). Байт — это наименьшая порция данных, адресуемая в памяти компьютера. Вы знаете, что байтом названа и единица измерения информации. Мог ли язык Паскаль пренебречь этим фактом? Нет, и ещё раз нет! В Паскале существует тип данных, который так и называется — **Byte**. Переменные типа **Byte** объявляются так.

```
var A, B, C : byte;
```

Байтовые переменные вмещают числа от 0 до 255. Это немного, но в некоторых случаях достаточно, и тогда применение байтовых переменных значительно экономит память.

Перейдем к брату-близнецу байта — коротышке **ShortInt**. Его имя расшифровывается как Short Integer — «короткое целое». Почему близнец? А потому, что он тоже занимает один байт памяти, но вмещает другой диапазон чисел — от минус 128 до плюс 127. Вместе с нулем получаются те же 256 значений. Этот тип данных тоже введен для экономии памяти. В самом деле, к чему тратить лишнюю память на переменные, хранящие маленькие числа? Отведем им один байт, только кодировать будем так, чтобы половина значений стала отрицательной. Так родился близнец-коротышка **ShortInt**.

Понятно, что ёмкости младших братьев хватает далеко не всегда, — даже количество дней в году не помещается в байтовой переменной. Для действий с более крупными числами программист обращается к средним братьям: типам **Integer** и **Word**. Каждый из них занимает по два байта памяти. Но если тип **Integer** вмещает диапазон чисел от минус 32768 до плюс 32767 (справедливо для Borland Pascal), то для типа **Word** диапазон сдвинут в положительную область и составляет от 0 до 65535. Кстати, название этого типа чисел — **Word** — переводится как «слово». Оно восходит ко временам 16-разрядных мини-ЭВМ, где длина так называемого машинного слова составляла два байта.

Когда не хватает емкости средних братьев, программисты зовут старшего — четырехбайтовый тип **LongInt** (Long Integer — «длинное целое»). Этот тип данных вмещает числа, превышающие два миллиарда. В табл. 2 представлены целочисленные типы данных языка Паскаль.

Примечание. Размеры и ёмкость целочисленных типов зависят от компилятора и его настроек. Так, в 32-разрядном компиляторе Delphi типы **Integer** и **LongInt** совпадают и представлены 4-мя байтами, а также имеется 8-байтовый тип **Int64**.

Табл. 2 – Целочисленные типы данных

Тип данных	Размер в байтах	Диапазон возможных значений	
		От	До
Byte	1	0	255
Shortint	1	-128	127
Word	2	0	65535
Integer (BP, FP)	2	-32768	32767
Integer (Delphi)	4	-2147483648	2147483647
Longint	4	-2147483648	2147483647

Хорошо, ну а если и ёмкости **LongInt** недостаточно? Неужели это предел? Конечно, нет. Но рассмотренных целочисленных типов хватает в большинстве случаев, где требуется точный подсчет. Подчеркиваю ещё раз — точный. Вещественные числа, о которых я расскажу в следующей главе, вмещают огромные значения, но представляют их приближенно. Для точного представления громадных чисел используют сложные типы данных.

Капля, переполняющая чашу

Конечно, вы догадались, что размер числового типа определяет его емкость, то есть диапазон возможных значений. А что случится при попытке выйти за этот диапазон? На ум приходит доверху наполненная чаша: очевидно, что лишняя капля стечет по стенке, и в чаше ничего не изменится. Так ли будет с числовой переменной? Вопрос не праздный, и, для ответа на него, проведем эксперимент.

```
{$R+  - включить проверку диапазонов }
var   N : byte;
begin
      N:= 255;    { 255 – максимальное значение для байта }
      N:= N+1;
      Writeln(N); Readln;
end.
```

Введите и откомпилируйте эту программу. В первой её строке вставлена директива, разрешающая компилятору следить за диапазонами числовых переменных. Эта директива соответствует флажку «Range checking» в окне опций компилятора (рис. 74).

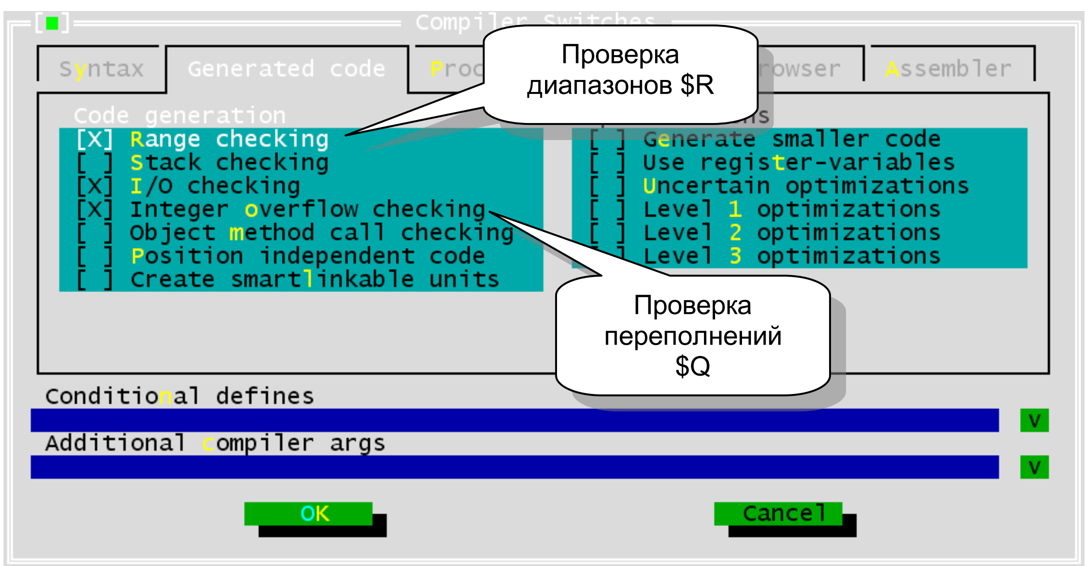


Рис. 74 – Окно опций компилятора

Запуск программы приведет к сообщению об ошибке «Runtime Error 201». Это значит, что попытка превысить диапазон для байтовой переменной, вызвала аварию программы.

Теперь измените директиву в первой строке, отключив проверку диапазонов (замените знак «+» знаком «—»).

```
{$R-   - отключить проверку диапазонов }
```

Этот вариант программы не выдаст сообщений об ошибке, но результат ошеломит вас — это будет ноль! Вот так чаша! Числовая переменная оказалась необычной посудой, — лишняя капля полностью опустошила её! И теперь можно вновь заполнять пустую чашу. Убедитесь в этом, поменяв единицу на другое слагаемое, например 5, — в результате сложения получится 4. Открытое нами явление называют **переполнением** (по-английски — OVERFLOW).

В следующем опыте запустим такую программу.

```
{$R-   - отключить проверку диапазонов }
var   N : byte;
begin
      N:= 0;      { 0 – минимальное значение для байта }
      N:= N-1;
      Writeln(N); Readln;
end.
```

Результат ещё удивительней: теперь программа напечатает число 255! То есть, удалив из пустой чаши несуществующую каплю, мы наполнили её доверху! Этот фокус называют **антипереполнением**, то есть переполнением наоборот.

Проделав опыты с переменными других числовых типов, вы убедитесь, что переполнение и антипереполнение может постигнуть любую из них. Так, добавление единицы к положительному числу 32767 в переменной типа **INTEGER** дает отрицательный результат -32768. Отсюда следует общее правило: добавление единицы к максимальному значению для числового типа дает минимальное значение. И наоборот: вычитание единицы из минимального значения дает максимальное. Рис. 75 наглядно показывает это.

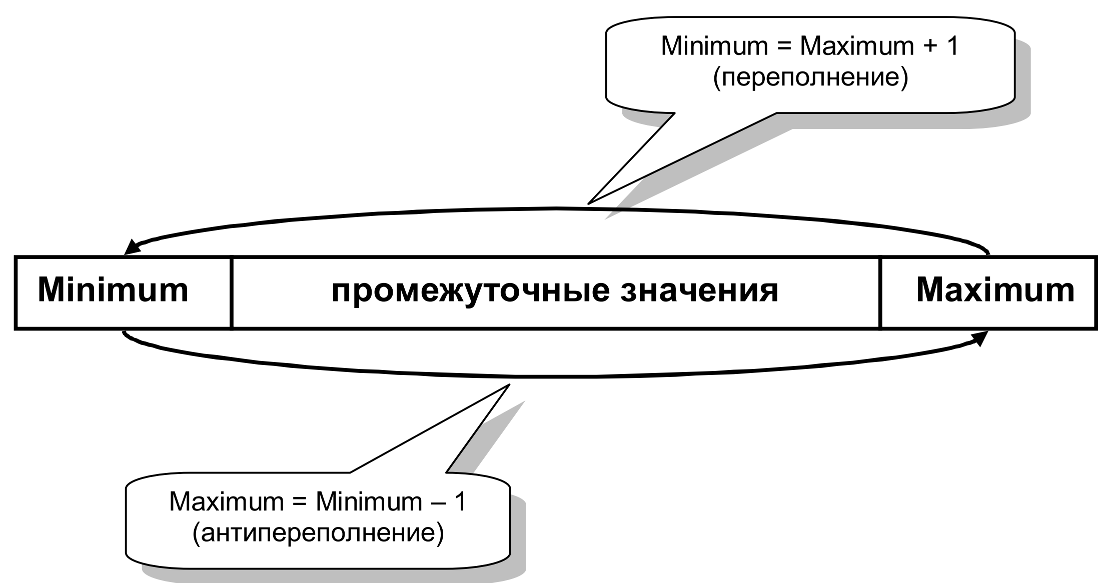


Рис. 75 – Изменение числовых переменных при переполнении и антипереполнении

Такая вот чудная арифметика! Причина переполнений и антипереполнений кроется в устройстве регистров процессора, — в свое время мы узнаем о них больше при изучении двоичной системы счисления. Или вспомните одометр — прибор для подсчёта пробега автомобиля: по достижении предельного количества километров (99999) одометр сбрасывается в ноль.

Сейчас важно понять, что присвоение переменной некоторого выражения не гарантирует правильного результата, — он будет верным лишь при отсутствии переполнений и антипереполнений. Когда в вычислении участвуют переменные разных типов, оно выполняется в самом емком формате, то есть в **Longint**, а затем результат «обрубается» в соответствии с типом принимающей переменной, например:

```
{ $R- }
var   B: Byte;    S: ShortInt;      W: Word;    N: Integer;
      . . .
      N:= B + S + W;
```

Здесь даже при положительных значениях всех суммируемых операндов, результат в переменной **N** может оказаться отрицательным! Если вам не по нраву такое поведение программы, включайте директиву проверки диапазонов **$R+**.

Инкремент и декремент

Угадайте, что чаще всего делают с целыми переменными? — прибавляют и вычитают единицу. Потому в процессорах стараются ускорить эти операции. Паскаль не обошел вниманием эту особенность программ, и предлагает вам две процедуры, объявленные так:

```
procedure Inc (var N : longint);    { прибавление единицы к переменной N }
procedure Dec (var N : longint);    { вычитание единицы из переменной N }
```

Хотя параметр **N** в процедурах объявлен как **LONGINT**, в действительности здесь может стоять переменная любого порядкового типа: **INTEGER**, **WORD**, **BYTE**, **CHAR** и даже **BOOLEAN**.

```
var   B: byte; N: integer; C: char;
      . . .
      Inc(B);      { B:= B+1 }
      Dec(N);      { N:= N-1 }
      C:= 'A';    Inc(C);     { 'B'}
```

Процедуры инкремента и декремента — так их называют — выполняются быстрее операторов присваивания **N:=N+1** и **N:=N-1**.

Работающим в IDE Borland Pascal следует учесть, что здесь процедуры инкремента и декремента не подвластны директиве **$R+** (в отличие от сложения и вычитания). То есть, переполнения и антипереполнения не вызывают аварий.

Диапазоны

Контроль переполнений директивой **$R+** повышает надежность программ. Но порой нужны более сильные ограничения. Предположим, некая переменная **M** по смыслу является порядковым номером месяца в году. Стало быть, её значения должны быть ограничены диапазоном от 1 до 12. Программист может указать это компилятору, объявив переменную как диапазон, и явно задав допустимые пределы её изменения:

```
var   M : 1..12;
```

Диапазон выражается двумя константами: минимальным и максимальным значениями, разделенными двумя точками. Теперь, при включенной директиве **$R+**, будет выдано сообщение об ошибке при попытке присвоить этой переменной любое значение за пределами 1…12. Во всем прочем диапазон — это обычный целочисленный тип (в данном случае — однобайтовый).

Перечисления

Рассмотрим ещё пример:

```
var   M : 1..12;          { месяцы }
      D : 1..7;           { дни недели }
      …
      M:= D;      { здесь возможна смысловая ошибка }
```

Здесь объявлены две переменные: **M** — номер месяца в году, и **D** — номер дня недели. Это сделано через диапазоны, что гарантирует соблюдение границ. Но ничто не мешает нам присвоить месяцу значение дня, — ведь это не нарушит установленных пределов. Другое дело — смысл. Есть ли смысл в таком присваивании, или налицо ошибка программиста? Вероятней всего — последнее. Выявить ошибки такого рода помогает ещё один тип данных — перечисление.

Перечислением программист дает имена всем возможным значениям переменных, эти имена перечисляются внутри круглых скобок. Например, переменные **M1** и **M2** могут быть объявлены через сокращенные названия месяцев, а переменные **D1** и **D2** — через сокращенные названия дней недели.

```
var   M1, M2 : (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
      D1, D2 : (Mond, Tues, Wedn, Thur, Frid, Satu, Sund);
```

Теперь компилятор разрешит присваивать переменным только объявленные значения, например:

```
      M1:= Apr;           { допустимо }
      M1:= M2;            { допустимо }
      M1:= 3;             { ошибка }
      M1:= Jan+2;         { ошибка }
      D2:= M1;            { ошибка }
```

Кстати, один из перечислимых типов вам знаком — это булев тип. Объявление булевой переменной равнозначно объявлению перечисления.

```
var   B : ( FALSE, TRUE );          { равнозначно B : Boolean; }
```

Имена в перечислениях — это не строковые константы. Поэтому имя **Jan** и строка «**Jan**» совсем не одно и то же. Иначе говоря, оператор **Write(M1)** не напечатает вам название месяца, который содержится в переменной **M1**. Вы спросите, а как же печать булевых данных? Ведь они печатаются как «TRUE» и «FALSE». Да, но это единственное исключение.

Порядковые типы

Итак, вы познакомились с пятью числовыми типами данных, диапазонами и перечислениями. Вместе с булевым и символьным типами они составляют семейство порядковых типов данных, а значит, имеют общие свойства и области применения. Рассмотрим их.

Определение порядкового номера

Название «порядковый» говорит о том, что значения этих типов данных упорядочены относительно друг друга. С числами всё ясно, — здесь порядок очевиден. А символы? Если вспомнить алфавит и таблицу кодировки символов, вопрос отпадет.

Хорошо, а как насчет перечислений и булевого типа? Оказывается, в памяти компьютера они тоже хранятся как числа. Например, упомянутое выше перечисление месяцев в памяти компьютера кодируется числами 0, 1, 2 и так далее, то есть как числовой диапазон 0..11. Таким образом, значение **Jan** соответствует нулю, **Feb** — единице и так далее. Подобным образом кодируются и булевы данные: **FALSE** — нулем, а **TRUE** — единицей.

В Паскале есть функция, определяющая числовой код данных любого порядкового типа. Она называется **Ord** (от Order — «порядок»), вот примеры её применения (в комментариях указаны результаты).

```
Writeln ( Ord(5) );          { 5 }
Writeln ( Ord('F') );        { 70 – по таблице кодировки}
Writeln ( Ord(Mar) );        { 2 – смотри перечисление месяцев }
Writeln ( Ord(False) );      { 0 }
Writeln ( Ord(True) );       { 1 }
```

Для числа функция возвращает само число, для символа — код по таблице кодировки, а для перечислений — порядковый номер в перечислении, считая с нуля.

Сравнение

Из того, что данные порядковых типов кодируются числами, следует возможность их сравнения. Например, для перечислений месяцев и дней недели можно записать.

```
if M2 > M1 then … { если второй месяц больше первого }
if D1 = D2 then … { если дни совпадают }
```

Нельзя сравнивать данные разных перечислимых типов.

```
if M2 > D1 then …        { месяц и день – недопустимо }
if 'W' > 20 then …       { символ и число – недопустимо }
```

Но любые типы можно сравнить, приведя их к числовому типу.

```
if Ord(M2) = Ord(D1) then …   { сравниваются числовые коды }
if Ord('W') > 20 then …       { сравнивается код символа с числом }
```

Прыг-скок

Итак, числа, символы, булевы данные, диапазоны и перечисления принадлежат к порядковым типам. В общем случае наращивать и уменьшать порядковые переменные путём сложения и вычитания нельзя (можно лишь числа и диапазоны). Но рассмотренные ранее процедуры инкремента (**INC**) и декремента (**DEC**) умеют это делать, они были введены в Паскаль фирмой Borland. Другим таким средством являются функции **SUCC** и **PRED**, которые существовали ещё в исходной «виртовской» версии языка.

Функция **SUCC** (от слова SUCCESS — «ряд», «последовательность») принимает значение порядкового типа и возвращает следующее значение того же самого типа, например:

```
      Writeln ( Succ(20) );         { 21 }
      Writeln ( Succ('D') );        { 'E' }
      Writeln ( Succ(False) );      { True }
      m:= Succ(Feb);                { переменной m присвоено Mar }
```

Функция **PRED** (от PREDECESSOR — «предшественник») возвращает предыдущее значение порядкового типа:

```
      Writeln ( Pred(20) );         { 19 }
      Writeln ( Pred('D') );        { 'C' }
      Writeln ( Pred(True) );       { False }
      m:= Pred(Feb);                { переменной m присвоено Jan }
```

Функции **SUCC** и **PRED** подчиняются директиве контроля диапазонов **$R+**. Например, следующие операторы вызовут аварийное прекращение программы:

```
{ $R+ }
      m:= Succ(Dec);    { превышение верхнего предела }
      m:= Pred(Jan);    { выход за нижний предел }
```

В Borland Pascal есть одна тонкость: директива **$R+** не действует, если функции **SUCC** и **PRED** вызываются для чисел, например:

```
{ $R+ }
var   B : byte;
      . . .
      B:=255;     B:= Succ(B);      { нет реакции на переполнение }
      B:=0;       B:= Pred(B);      { нет реакции на антипереполнение }
```

В таких случаях в Borland Pascal имеет силу директива проверки переполнения **$Q+**, которая соответствует флажку «Overflow Checking» в окне опций компилятора (рис. 74). Директивы **$R+** и **$Q+** можно применять совместно, например:

```
{ $R+, Q+ }
var   B : byte;         { допустимые значения для байта от 0 до 255 }
      C : 'a'..'z';     { это ограниченный диапазон символов }
      . . .
      C:='z';     C:= Succ(C);      { сработает R+ }
      B:=255;     B:= Succ(B);      { сработает Q+ }
```

Счетчики циклов

В операторе **FOR-TO-DO** для счетчика цикла мы применяли числовые переменные. Теперь разнообразим меню: ведь для этого годятся переменные любого порядкового типа, например:

```
var   m : (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
      . . .
      for m:= Jan to Dec do . . .
```

А вот так вычисляется сумма кодов для символов от «a» до «z», здесь счетчиком цикла является символьная переменная:

```
var   Sum : word;  Chr : char;
      . . .
      Sum:=0;
      for Chr:= 'a' to 'z'    do Sum:= Sum + Ord(Chr);
```

Метки в операторе выбора

Вот ещё одно следствие числового кодирования: любой порядковый тип может служить меткой в операторе **CASE-OF-ELSE-END**:

```
var   c : char;
      . . .
      Case c of
            '0'..'9': Writeln('Цифра');
            'a'..'z': Writeln('Латинская строчная');
            'A'..'Z': Writeln('Латинская прописная');
      end;
```

А вот ещё пример:

```
type  TMonth = (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
var   m : TMonth;  { здесь хранится один из месяцев }
      . . .
      Case m of
            Jan, Feb, Dec     : Writeln('Зима');
            Mar..May          : Writeln('Весна');
            Jul..Aug          : Writeln('Лето');
            Sep..Nov          : Writeln('Осень');
      end;
```

Как видите, метки можно группировать, перечисляя их через запятую или объединяя в диапазон.

Разумный контроль

Директивы **$R+** и **$Q+** лучше использовать при отладке программы. В хорошо отлаженной программе таких ошибок возникать не должно, — за это отвечает программист. При компиляции окончательной версии эти директивы лучше отключить, чтобы не увеличивать размер программы и не замедлять её работу.

Итоги

- Существуют три категории типов данных: простые, сложные и указатели.
- Простые типы данных делятся на порядковые и вещественные.
- К порядковым типам относятся целые числа, символы, перечисления и булевы данные.
- Целые числа представлены пятью типами, которые отличаются размерами и диапазонами.
- Присвоение переменной порядкового типа значения, выходящего за допустимый диапазон, влечет ошибку нарушения диапазона.
- При включенной директиве **$R+** нарушение диапазона приводит к аварии программы, а при отключенной – к переполнению или антипереполнению.

- Функцией **ORD** можно определить код любого значения порядкового типа.
- Переход к следующему или предыдущему значению порядкового типа выполняется функциями **SUCC** и **PRED**.
- Для быстрого прибавления и вычитания единицы предпочтительней применять процедуры **INC** и **DEC**.
- Порядковые типы данных обладают рядом общих свойств, что позволяет применять их в счетчиках циклов и в метках оператора выбора.

А слабо?

А) Напомню, что функция **SizeOf** возвращает объем памяти, занимаемый переменной, например:

```
      Writeln( SizeOf( LongInt ) );      { 4 }
      Writeln( SizeOf( M1  ) );          { 1 }
```

Воспользуйтесь ею для распечатки размеров всех известных вам порядковых типов данных.

Б) Перечислимые типы и диапазоны строятся на базе других типов данных (**Byte**, **ShortInt** и так далее). Какие типы данных, по вашему мнению, будут положены в основу следующих диапазонов:

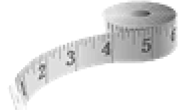
```
var   N : -10..10;
      M : -200..200;
      R : 0..40000;
      L : 0..400000;
      S : ’0’..’9’;
```

В) Процедура печати **Writeln** не способна распечатать название месяца, представленного в перечислении. Напишите для этого свою собственную процедуру (объявите тип **TMonth** и воспользуйтесь оператором **CASE**).

Г) «Не думай о секундах свысока…». Штирлицу подарили секундомер, который показывал секунды, прошедшие с начала суток. Пусть ваша программа переведет это число в привычные часы, минуты и секунды.

Подсказки: во-первых, примените операции **DIV** и **MOD**. Во-вторых, переменную для секунд объявите как **LONGINT** (а не **INTEGER**), поскольку количество секунд в сутках (86400) не поместится в типе **INTEGER**.

Глава 33
Вещественные числа



Почему так несовершенны все людские поделки? Даже компьютер и язык Паскаль! Эх, был бы числовой тип, пригодный на все случаи жизни, но…

Пять целочисленных типов не покрывают всех потребностей в вычислениях. Во-первых, диапазон их значений не так уж велик. Скажем, население Земли — около шести миллиардов — не поместится в переменной типа **LongInt**. А что сказать о комарином «населении»? Это, во-первых. А во-вторых, такими числами нельзя выразить дробные значения.

Выручают вещественные числа. Откуда такое чудное название? — Этими числами можно выразить количество сыпучих и жидких веществ. Если так, то целые числа следовало назвать штучными. У вещественных чисел есть и другое название — действительные. Которое из двух предпочтете? — дело вкуса.

Изображение вещественных чисел

Вещественные числа отнесены к простым типам, но устроены сложнее целых. Рассмотрим способы изображения таких чисел. Представить их можно двояко: либо в привычной для нас форме с фиксированной точкой (в Паскале точку используют вместо запятой), либо в так называемом научном (логарифмическом) формате. Увидев где-либо такое число, не пугайтесь, — здесь применен научный формат изображения вещественного числа.

```
3.33333343267441E-0002
```

Мы видим десятичную дробь, на хвосте которой болтается буква «E» и число **-0002**. Вот разгадка этой записи: дробь, что расположена до буквы «E», называется мантиссой, а число за этой буквой — порядком. Порядок показывает, на сколько позиций надо передвинуть десятичную точку в мантиссе для получения числа в привычном виде. Здесь порядок отрицательный, поэтому точка сдвигается на две позиции влево, а значит перед нами число **0.0333333343267441**. Для положительного порядка точку двигают вправо, стало быть, число

```
3.33333343267441E+0003
```

в форме с фиксированной точкой запишется так: **3333.33343267441**.

Разумеется, что при нулевом порядке точку не трогают. Вот и вся премудрость научного формата, который называют ещё форматом с плавающей точкой. Если научная форма кажется вам причудливой и неудобной, изобразите иначе следующие числа:

```
 9.1093829140E-0031 = 9.1093829140•10^-31 – масса электрона, кг
 1.9889200000E+0030 = 1.9889200000•10^30  – масса солнца, кг
```

Вывод вещественных чисел

Паскаль может избавить вас от мысленных передвижений десятичной точки: при печати вещественных чисел допустимы спецификаторы ширины поля. Напомню, что для вещественных чисел спецификатор состоит из двух частей, разделенных двоеточием. Первая часть задает общую ширину поля печати, а вторая — количество знаков после точки. Рассмотрим несколько вариантов вывода одного и того же числа со спецификаторами и без них. Подопытным будет число **10/3**, что соответствует бесконечному ряду троек: **3.333...** и т.д. Вот программа для этого опыта.

```
{ Программа для исследования форматов вывода вещественных чисел }
begin
      Writeln(F, 10/3);             { без спецификаторов }
      Writeln(F, 10/3 : 12);        { указывается только ширина поля }
      Writeln(F, 10/3 : 15:0);      { только целая часть }
      Writeln(F, 10/3 : 15:2);      { два знака после точки }
      Writeln(F, 10/3 : 15:3);      { три знака после точки }
end.
```

Результат её работы таков.

```
3.3333333333333E+0000
 3.333E+0000
              3
           3.33
          3.333
```

Как говорится, лучше раз увидеть... Вывод ясен: если не указать спецификатор поля или его вторую часть, то число выводится в научном формате с плавающей точкой, а иначе — с фиксированной.

Типы вещественных чисел

Подобно целым, вещественные числа представлены несколькими типами, которые разнятся размерами и диапазонами значений. Причина разнообразия всё та же — стремление сэкономить память. В табл. 3 показаны четыре типа вещественных чисел языка Паскаль.

Табл. 3 – Вещественные типы

Тип данных	Точность	Диапазон возможных значений		Количество значащих цифр (точность)	Размер в байтах
		От	До		
Real	Стандартная	2.9×10^{-39}	1.7×10^{38}	11-12	6
Single	Одинарная	1.5×10^{-45}	3.4×10^{38}	7-8	4
Double	Двойная	5.0×10^{-324}	1.7×10^{308}	15-16	8
Extended	Повышенная	3.6×10^{-4951}	1.1×10^{4932}	19-20	10

Но почему в колонке минимальных значений я указал не нули, а очень маленькие числа? Да, ноль допустим, но для оценки точности вычислений важно знать именно этот предел. Разумеется, что указанные диапазоны распространяются и на отрицательные числа.

Теперь исследуем точность представления чисел разными типами данных.

```
{ Программа для исследования точности вещественных типов }
var   F0 : Real;  F1 : single;  F2 : double;  F3 : extended;
begin
      F0:= 1/3;  F1:= 1/3;  F2:= 1/3;  F3:= 1/3;
      Writeln('Single   = ',  F1:23:18);
      Writeln('Real     = ',  F0:23:18);
      Writeln('Double   = ',  F2:23:18);
      Writeln('Extended= ',  F3:23:18);
end.
```

Десятичное представление дроби 1/3 нам известно, — это бесконечная последовательность троек, а результат вычислений по этой программе перед вами (для Borland Pascal, в других компиляторах результаты могут немного отличаться):

```
Single   =    0.333333343267440796
Real     =    0.333333333333484916
Double   =    0.333333333333333315
Extended=    0.333333333333333333
```

Как и следовало ожидать, тип **Extended** дает самую высокую точность: после десятичной точки следуют одни тройки. Другие типы менее точны. Если так, зачем они нужны? Обратимся к истории.

Первые версии Паскаля ещё не застали персональных компьютеров. Тогда в языке существовал только один тип вещественных чисел — **Real.** Его считают стандартным типом Паскаля, и для обработки таких чисел годится любой процессор (но вычисления будут медленными).

Но вот появились компьютеры с математическими сопроцессорами, многократно ускоряющими счет. Эти сопроцессоры оперируют с форматами, отличными от **Real**. Для совместимости с новой техникой в язык были введены ещё три типа чисел, указанные в табл. 3. Тип **Extended** дает наивысшую точность и самый широкий диапазон представления чисел. И это понятно, — ведь его размер больше, чем у других, и составляет 10 байтов. Но почему он выигрывает и в скорости? А потому, что для сопроцессора тип **Extended** — родной, применяйте его для вычислений. А что же **Single** и **Double**? Поскольку они занимают меньше места в памяти, то лучше подходят для хранения больших объемов данных.

Сравнение вещественных чисел

Вещественные числа часто сравнивают между собой. Однако проверка их на точное равенство таит неприятный сюрприз, — такие сравнения ненадежны! Всё потому, что вещественные числа — приближенные; они могут быть очень близки, и всё же чуточку не совпадать друг с другом. Точное совпадение — это удача, а не закономерность. Правильней сравнивать числа с некоторой точностью, как в показанном ниже примере.

```
var   A, B : Extended;
. . .
if A = B    then  …           { это ненадежное сравнение }
if Abs(A-B) < 0.001 then …    { надежное сравнение с точностью 0.001}
```

Во втором сравнении переменные **A** и **B** считаются одинаковыми, если отличаются менее чем на одну тысячную. Как видите, знаком равенства тут и не пахнет. К слову, функция **Abs** возвращает абсолютное значение аргумента, — ведь здесь надо получить положительное значение разности. Выражение **Abs(A-B)** в математике пишется так: **|A-B|**.

Типы данных пользователя

Богатый арсенал типов данных, запасенный в Паскале, кажется достаточным на все случаи жизни. Скоро вы убедитесь, что это не так. Но Паскаль разрешает программисту создавать свои собственные типы данных, заточенные под определенную задачу. Их называют пользовательскими типами, и строят на основе всё тех же базовых типов Паскаля. Рассмотрим пример.

Предположим, ваша программа содержит много переменных типа **Integer**. Но, по ходу работы над проектом, вы решили заменить этот тип чисел на какой-то

иной, например на **Longint** или даже **Extended**. Такую замену можно сделать редактором текста, тщательно порывшись в программе и найдя все места, где объявлены переменные. Но можно сделать лучше — объявить собственный тип данных.

Для объявления пользовательских типов в Паскале служит секция описания типов, которая начинается с ключевого слова **TYPE** — «тип». Внутри секции можно объявить один или несколько типов данных пользователя. Так, например, вы можете объявить свой тип на базе встроенного типа **Integer**.

```
Type  TValue = Integer;
```

Здесь объявлен тип данных по имени **TValue** (Value — «значение»), он равнозначен типу **Integer**. Как видите, объявление типа схоже с объявлением константы. Только справа от знака равенства следует не значение, а описание типа.

Имя пользовательского типа придумывают по общим правилам для имен. В свое время мы учредили префиксы для констант и аргументов процедур, — префиксы делают программу понятней. Для констант мы договорились применять префикс «C», для аргументов процедур и функций — префикс «a». Для типов возьмем префикс «T» (от слова Type). Повторяю: префиксы — это всего лишь добровольное соглашение программистов, а не требование языка.

Теперь, когда тип **TValue** объявлен, его можно использовать для объявления переменных, например:

```
Type  TValue = Integer;
Var   A, B, C : TValue;
      . . .
      Readln(A, B);
      C:= A+B;
```

Если со временем потребуется изменить типы переменных **A**, **B** и **C** на тип **Longint**, то мы исправим лишь объявление пользовательского типа.

```
Type  TValue = Longint;
```

И после компиляции все переменные типа **TValue** примут тип **Longint**.

На первый взгляд, пользовательские типы дают лишь косметические удобства. Но скоро — при освоении сложных типов данных — вы и шагу не ступите без них. Впрочем, пользовательские типы пригодятся нам гораздо раньше — для преобразования типов данных.

Совместимость и преобразование типов

Поздравляю! Теперь вам знакомы все простые типы данных — солдаты нашей армии. Сражаясь в едином строю, они будут передавать при необходимости данные друг другу. Такая передача данных должна подчиняться определенным правилам. Они просты, но заставляют программиста всякий раз задуматься: то ли я делаю? А это придает программам надежность.

Далее рассмотрим правила, по которым данные одного типа обращаются в другой: вещественное число — в символ, или целое — в булево. Эти волшебства возможны благодаря числовому кодированию всех типов данных. Поэтому целые числа будут центральным звеном всех преобразований, с них и начнем.

Целое и целое

Все целые типы совместимы между собой, а значит позволено взаимное присваивание их значений. Но не забывайте о возможном нарушении диапазонов. Вот общее правило: переменные более ёмких старших типов всегда примут данные из младших типов своего братства. Обратное не возбраняется, но может вызвать нарушение диапазонов, например:

```
Var   B: Byte; I: Integer; W: Word; L: Longint;
      . . .
      { Эти операторы не вызовут нарушения границ диапазонов }
      I:= B;      W:=B;       L:= W;      L:=I;
      { Эти операторы могут повлечь нарушения диапазонов }
      B:=I;       I:=W;       W:=L;
```

Когда число **N** не помещается в переменной, то в неё попадает лишь младшая часть числа **N** (т.е. остаток от деления **N mod 256** или **N mod 65536**).

Целое и символ

Взаимно превращать эти типы данных мы научились при шифровании символа; напомню об этом. Функция **Ord** возвращает числовой код любого порядкового типа, в том числе и символа. А функция **Char** делает обратный фокус, превращая число в символ, вот примеры:

```
      Writeln ( Ord('D') );        { 68 }
      Writeln ( Char(68) );        { D }
```

Как видите, число в символ преобразуется через имя типа **Char**. Это общий прием, волшебная палочка для обращений типов данных. Например, булевых.

Целое и булево

Превратить булево в целое можно всё той же функцией **Ord**.

```
      Writeln ( Ord(False) );       { 0 }
      Writeln ( Ord(True) );        { 1 }
```

А для обратного превращения воспользоваться именем типа **Boolean**.

```
      Writeln ( Boolean(0) );       { False }
      Writeln ( Boolean(1) );       { True }
```

Целое и перечисление

Вернемся к перечислению месяцев, вымышленному нами в предыдущей главе, где переменная была объявлена так.

```
var   m : (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
```

Превратить значение такой переменной в число можно функцией **Ord**. А наоборот? Пока нам этого не удавалось. Но после объявления пользовательского типа данных задача решается очень просто.

```
Type  TMonth = (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
Var   m : TMonth;
      . . .
      m:= 3;             { это ошибка }
      m:= TMonth(3);     { это равнозначно m:= Apr (счет идет от нуля) }
```

Здесь объявлен пользовательский перечислимый тип **TMonth** (Month — «месяц»), далее вы вольны применять его и для объявления переменных, и для преобразования типов. Вот где проявляется сила пользовательского типа!

Вещественное и вещественное

Подобно целому братству, вещественное братство дружно между собой. Переменной одного вещественного типа можно присвоить значение другого типа, при условии, что это значение поместится в новое «жилище», то есть, не приведет к нарушению диапазона.

Целое и вещественное

Вещественным переменным можно присваивать целые значения, не задумываясь о последствиях, — преобразование происходит автоматически. Но обратная операция не так проста, поскольку здесь надо решить судьбу дробной

части вещественного числа, которая не попадет в целочисленную переменную. Есть две возможности: либо отбросить дробную часть, либо округлить вещественное число до ближайшего целого. Для этого Паскаль предлагает две функции, возвращающие целочисленные значения: **Trunc** — усечение, и **Round** — округление. Вот примеры их вызова.

```
      Writeln ( Trunc(3.75) );      { 3 }
      Writeln ( Round(3.75) );      { 4 }
      Writeln ( Round(3.25) );      { 3 }
```

Напоследок рассмотрите рис. 76, где показана общая картина совместимости и преобразования простых типов данных.

Строгий контроль типов в Паскале задуман для пущей надежности программам. В старых языках программирования (Си, Фортран) такого контроля либо не было, либо он был очень слаб. Программист перебрасывал данные как ему вздумается, не слишком заботясь о последствиях. Подобные вольности порождали массу ошибок. Теперь же Паскаль предлагает программисту следующее: пожалуйста, переноси данные, куда угодно, но при этом укажи явным образом, что ты делаешь.

Размеры переменных и типов данных

Иногда требуется знать не только содержимое переменных, но и объем занимаемой ими памяти. Разумеется, вы можете узнать это из таблиц, приведенных в справке или руководстве по языку. Размеры сложных типов данных тоже поддаются расчету. И всё же лучший способ определить размер переменной некоторого типа — вызов псевдофункции **SizeOf**. В качестве параметра она принимает либо имя переменной, либо имя типа, а возвращает целое число — объем занимаемой памяти в байтах. Вот примеры.

```
Type  TMonth = (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
Var   m : TMonth;
      . . .
      Writeln ( SizeOf(m) );              { 1 }
      Writeln ( SizeOf(TMonth) );         { 1 }
      Writeln ( SizeOf(Integer) );        { 2 }
      Writeln ( SizeOf(Extended) );       { 10 }
```

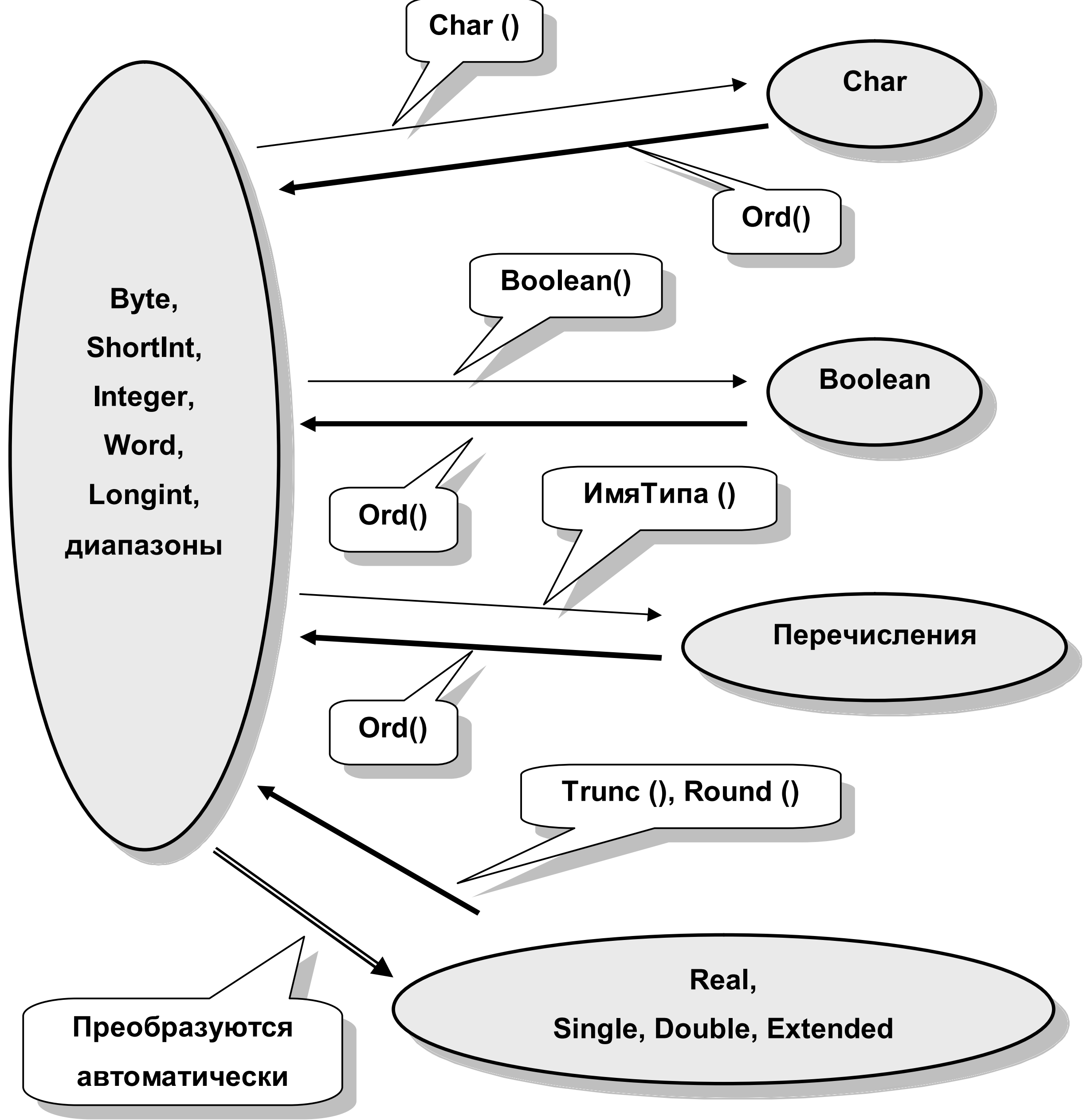


Рис. 76 – Совместимость и преобразования типов

Я обозвал **SizeOf** псевдофункцией за то, что никаких вычислений она не делает, — результат вычисляется при компиляции программы. Ведь компилятор сам «знает» объем памяти, занимаемой любым типом данных, и подставляет в программу уже готовое число.

Псевдофункция **SizeOf** удобна, и вдобавок улучшает переносимость программ. Например, в разных режимах компиляции Free Pascal и в разных компиляторах размер типа **Integer** может отличаться (2 или 4 байта). Применяя функцию **SizeOf**, вам не придется задумываться об этом и менять вручную одни числа на другие, — компилятор сделает это за вас.

Итоги

- Для представления дробных, а также очень больших и очень маленьких чисел используют вещественные типы данных. Разные вещественные числа различаются размером, диапазоном хранимых значений и точностью их представления.
- Тип **`Extended`** предпочтительней использовать для вычислений, тип **`Single`** – для хранения больших объемов данных в памяти и на диске.
- Вещественные числа печатаются либо в форме с плавающей точкой, либо в форме с фиксированной точкой.
- Вещественные числа, в отличие от целых, – приближенные. Сравнивать их между собой можно лишь с некоторой точностью.
- Вещественным переменным разрешено присваивать целые значения, при этом преобразование типов происходит автоматически.
- Целым переменным нельзя присвоить вещественные значения непосредственно, для этого используют либо функцию отсечения дробной части **`Trunc`**, либо функцию округления **`Round`**.
- Порядковые типы данных допускают взаимное преобразование посредством псевдофункций, имена которых совпадают с именами типов данных.
- Пользовательские типы данных объявляют внутри секции **`TYPE`**, такие типы данных делают программу гибче и надежней.
- Размер памяти, занимаемый переменной любого типа, определяют псевдофункцией **`SizeOf`**. Её применение снижает зависимость программы от особенностей компиляторов и компьютерных платформ.

А слабо?

А) Напишите две функции, округляющие вещественное число:

- до большего значения (например: 3.1 → 4; 3.9 → 4);
- до меньшего значения (например: 3.1 → 3; 3.9 → 3).

Б) Ваша процедура принимает строковую переменную, вычисляет среднее арифметическое кодов её символов и печатает его с двумя цифрами после точки.

В) Напечатайте с тремя знаками после точки 20 случайных вещественных чисел в диапазоне от 0 до 10. **Подсказка**: для формирования дробных чисел можно делить случайное число на другое число, например, **`Random(10000)/1000`**.

Г) Напечатайте с тремя знаками после точки 20 случайных чисел в диапазоне от 0 до 10 так, чтобы числа следовали по возрастанию. Подсказка: сравнивайте очередное число с предыдущим.

Д) Программа для подсчета стоимости покупок. Для каждой покупки пользователь вводит два действительных числа: вес покупки и цену за 1 кг в рублях. Признак завершения ввода данных — нулевой вес. Программа должна напечатать общую стоимость с точностью до копейки (два знака после точки) с округлением в большую сторону. Проверьте результат на калькуляторе.

Е) Квадратный корень. Квадрат — это равносторонний прямоугольник, его площадь вычисляется по формуле **S=D•D**, где **D** — сторона квадрата. А когда площадь **S** известна, и надо определить сторону **D**? Тогда из **S** извлекают квадратный корень (обозначается символом **√)**. Так, если **S=9**, то **D=√9=3**.

Для извлечения корня в Паскале есть функция **SQRT**. Напишите собственную функцию **MySQRT**, прибегнув к методу последовательных приближений. В грубом, нулевом приближении примем $D_0=1$. Последующее, более точные значения **D** будем вычислять по формуле

$$D_{i+1} = (D_i + S/D_i)/2$$

Так, при **S=9** получим $D_1=(1+9/1)/2=$ <u>5</u>, $D_2=(5+9/5)/2=$ <u>3.4</u> и так далее, пока абсолютная разность между двумя последовательными значениями **D** станет пренебрежимо мала. Функция **MySQRT** должна принять число и вычислить его корень с точностью **0.0001**. Внутри функции напечатайте промежуточные значения **D**. Подсказка: для D_i и D_{i+1} вам потребуются лишь две локальные переменные.

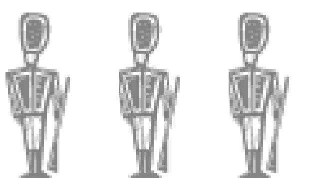
Ж) В тесто кладут четырех главных ингредиента: муку, сахар, яичный порошок и молоко. Всё это смешивается в пропорции, заданной рецептом. Например, рецепт **100:5:7:500** означает, что на 100 граммов муки кладут 5 граммов сахара, 7 граммов яичного порошка и 500 граммов молока. У пекаря есть некоторое количество всех ингредиентов, и он хочет замесить из них максимально возможное количество теста, соблюдая рецепт. Ваша программа должна ввести:

- Рецепт – это 4 целых числа.
- Исходное количество ингредиентов – это 4 действительных числа.

Программа должна напечатать:

- Общее количество полученного теста с точностью два знака после точки.
- Остатки ингредиентов – 4 числа с точностью два знака после точки.

Глава 34
Структура программы



В этой главе мы рассмотрим структуру программы, и завершим тем самым боевое построение нашего войска, начатое в 32-й главе.

Управляющие структуры

Управляющие структуры составляют основу языков программирования. Ключевых структур всего три:

- линейная последовательность – это естественный порядок выполнения операторов друг за другом, то есть слева направо и сверху вниз;
- альтернатива – выбор одного из двух или нескольких направлений исполнения операторов;
- цикл – повторное исполнение операторов до соблюдения некоторого условия.

Альтернатива и цикл представлены в Паскале несколькими операторами, из которых программист выбирает тот, что лучше подходит к решаемой задаче (рис. 77).

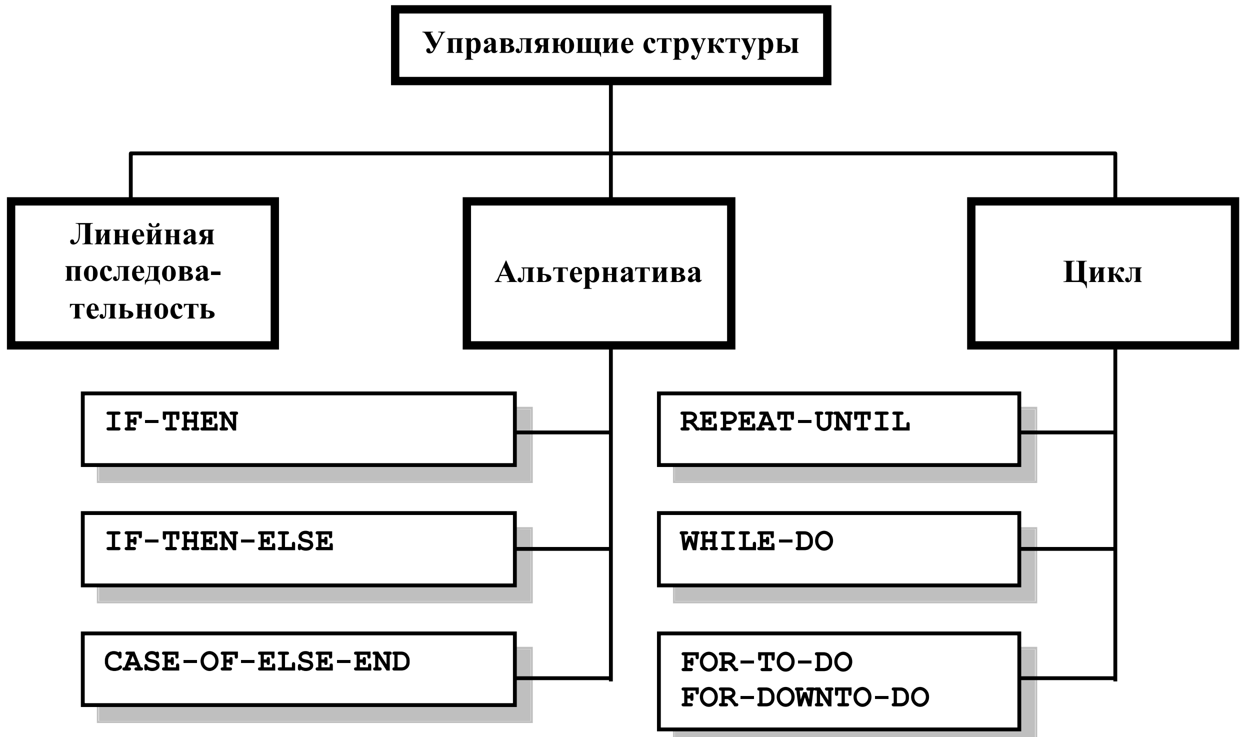


Рис. 77 – Управляющие структуры языка Паскаль

Итак, для организации альтернативы может быть использован один из трех операторов:

- неполный условный оператор **IF-THEN**;
- полный условный оператор **IF-THEN-ELSE**;
- оператор выбора **CASE-OF-ELSE-END**.

Для организации циклов программист также применяет три оператора:

- цикл с проверкой условия в конце **REPEAT-UNTIL**;
- цикл с проверкой условия в начале **WHILE-DO**;
- цикл со счетчиком **FOR-TO-DO** и **FOR-DOWNTO-DO**.

Обратите внимание на условия продолжения циклов **WHILE-DO** и **REPEAT-UNTIL**, — они взаимно противоположны! Первый из них выполняется, пока условие истинно, а второй — пока оно ложно.

Странно, что из этих немногих структур лепятся столь сложные программы!

Структура программы

Программа на Паскале состоит из ряда секций (Section — «часть», «раздел»). Под структурой программы будем понимать взаимное положение этих секций. На рис. 78 показана упрощенная структура программы.

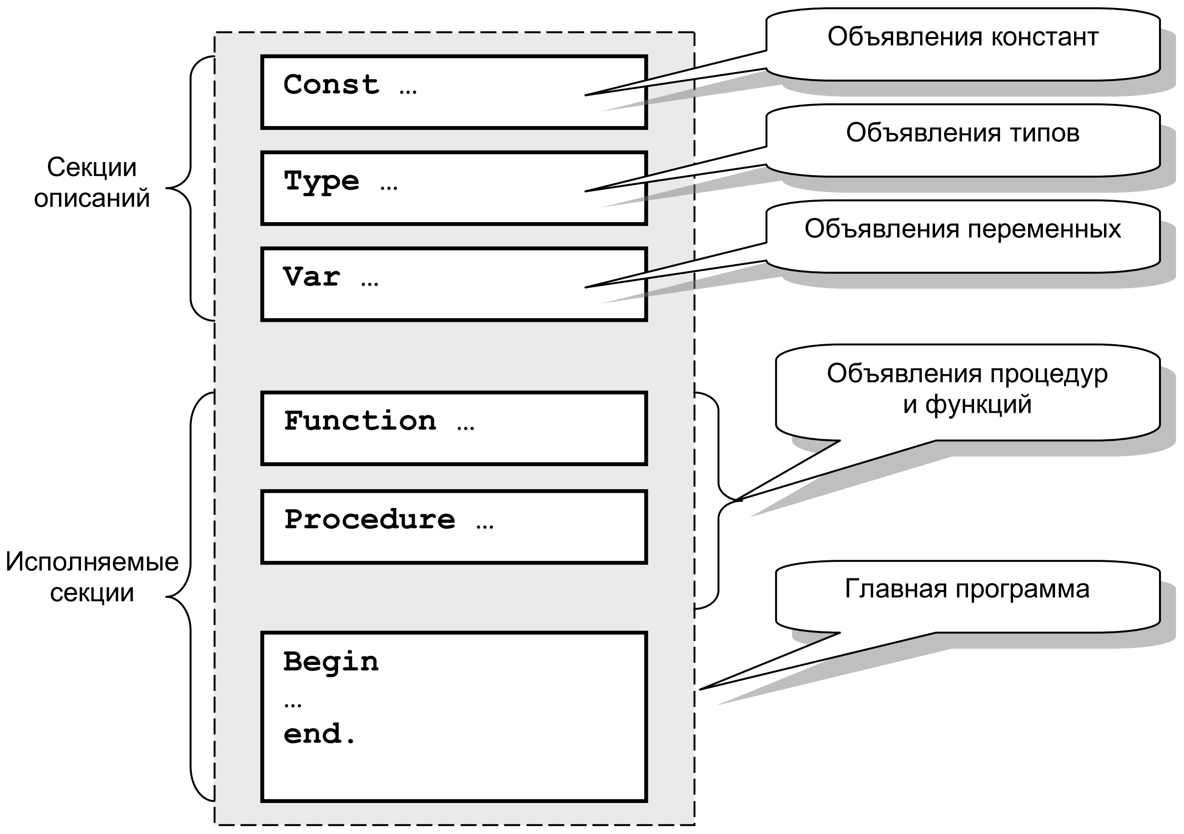


Рис. 78 – Структура программы на языке Паскаль

Каждую секцию открывает своё ключевое слово. Три секции: **Const**, **Type** и **Var** — образуют описательную часть программы. Здесь компилятор черпает информацию о размещении данных в памяти. Секции с описаниями процедур и функций и главная программа формируют исполнительную часть, — здесь содержатся исполняемые операторы (секция кода). Все секции, кроме главной программы, необязательны. Но, при необходимости, секции могут повторяться и чередоваться в любом порядке, соблюдая два простых правила:

- любой объект программы – будь то константа, тип, переменная или процедура – объявляется до своего применения;
- главная программа располагается в тексте последней (хотя исполнение начинается именно с нее!).

Два слова о точке с запятой (;). В описательной и в исполнительной частях программы её назначение слегка различается. Если в объявлениях точка с запятой завершает оператор и обязательна, то в секции кода она разделяет операторы и не нужна за последним оператором блока.

Структура процедур и функций

Процедуры и функции — основные строительные блоки программ, в крупных проектах их сотни. Главная программа обычно содержит несколько операторов, а основная работа отдается процедурам и функциям. Такой подход не только упрощает разработку, отладку и понимание программ, но и существенно уменьшает их размер (объем занимаемой памяти). Всё, что требует алгоритм, достигается вызовом одних процедур и функций из тела других, — то есть применением вложенных вызовов. Глубина вложения таких «матрешек» практически не ограничена. Опытный программист обычно разбивает большую программу на ряд мелких и простых процедур и функций.

Внутренняя структура процедур и функций схожа со структурой программы. Это своего рода программы в программе, потому их и называют подпрограммами. На рис. 79 показана упрощенная структура процедуры с условным именем **ABC**.

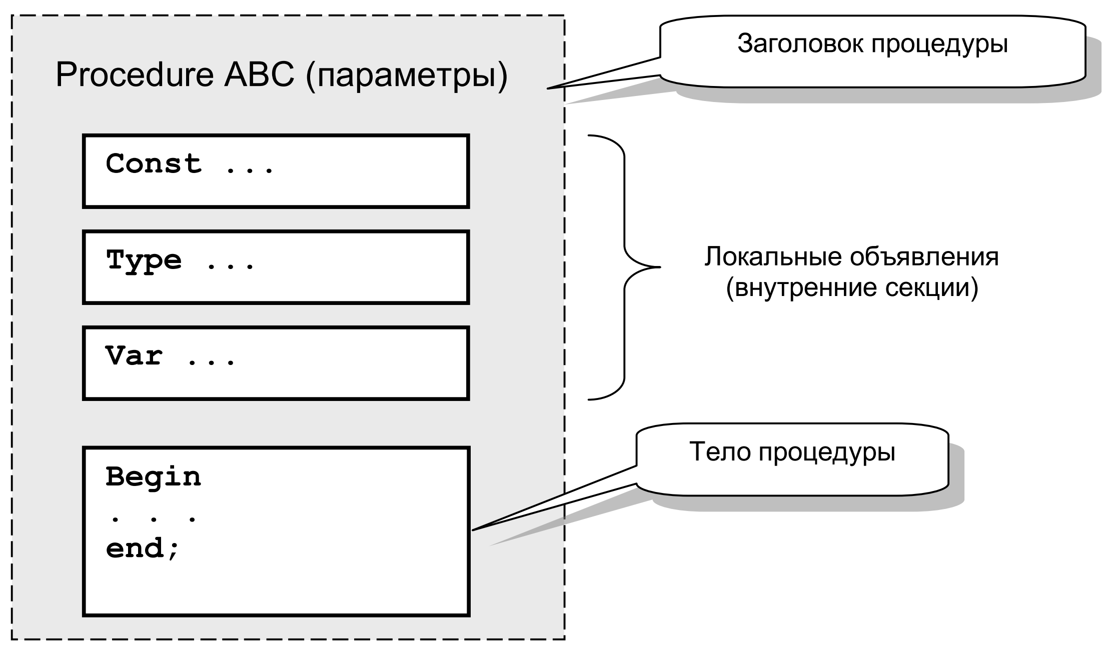


Рис. 79 – Структура процедуры

Такой же структурой обладают и функции, которые, в отличие от процедур, возвращают значение некоторого типа. Правила чередования секций внутри подпрограмм — локальных секций — точно такие же, как и для секций программы в целом, а именно:

- любой объект объявляется до своего применения;
- тело процедуры или функции обязательно и размещается последним.

Объявленные внутри подпрограммы константы, типы и переменные — локальные объекты — видны лишь внутри этой подпрограммы. При совпадении их имен с глобальными объектами, локальные имеют преимущество, то есть закрывают собою внешние объекты.

Обмен данными с подпрограммами

Вызов процедур и функций обычно сопровождается передачей данных между вызываемой подпрограммой с одной стороны и вызывающим её фрагментом с другой. Иначе говоря, данные либо передают внутрь подпрограммы, либо получают от нее. Иногда делают и то, и другое. Существует три способа такого обмена:

- через глобальные переменные;
- через параметры процедур и функций;
- возвратом результата через имя функции.

Передача данных через **глобальные** переменные кажется самой простой, — ведь эти переменные видны из многих частей программы. Но этот способ оправдан лишь в небольших проектах. С ростом размера и сложности программы всё труднее отслеживать взаимные влияния её частей через глобальные переменные. Это запутывает программу и снижает её надежность.

Для обмена данными разумнее использовать **параметры** процедур и функций, а также **имена функций**. В табл. 4 показаны три способа передачи данных через параметры.

Табл. 4 – Три способа передачи данных через параметры

Способ передачи данных	Пример заголовка процедуры	Пример вызова
По значению: в процедуру передается значение параметра.	`Procedure ABC (arg:integer);`	`ABC(10);` `ABC(X+3);`
По ссылке CONST: В процедуру передается ссылка на константу или переменную, содержащую данные.	`Procedure ABC (const arg:integer);`	`ABC(10);` `ABC(X);`
По ссылке VAR: В процедуру передается ссылка на переменную, содержащую данные.	`Procedure ABC (var arg:integer);`	`ABC(X)`

Опытного программиста отличает умение эффективно передавать данные; табл. 5 поможет вам выбрать наиболее удачный способ такой передачи.

Табл. 5 – Рекомендуемые способы передачи данных

Куда передавать данные	Рекомендуемый способ
Только в процедуру или функцию	1) По значению (простые типы) 2) По ссылке **CONST** (сложные типы)
Только из процедуры и функции	1) Через имя функции (одно значение) 2) По ссылке **VAR** (несколько значений)
В обоих направлениях	По ссылке **VAR** (любые данные)

В каждом случае предпочтительный способ указан первым. Данные простых типов лучше передавать внутрь подпрограмм по значению. По ссылке **CONST** передают строки и другие сложные типы данных (скоро мы изучим их). Через имя функции возвращают лишь один результат. А если надо вернуть несколько результатов, или вернуть сложный тип данных, используют ссылки **VAR**.

Встроенные процедуры и функции

Программа, сработанная профессионалом, состоит почти из одних только процедур и функций, разработка которых отнимает львиную долю времени. Но не всегда программисты пишут их сами. В Паскале запасено немало готовых подпрограмм — это встроенные в язык и в библиотеки процедуры и функции. С ними можно ознакомиться в руководстве по языку и во встроенной справке. Некоторые из них вам известны, и применялись нами.

Что дальше?

Мы изучили фундамент языка Паскаль, который составляют простые типы данных и управляющие структуры. Впереди интересные и серьезные проекты, в основе которых лежат сложные типы данных. Вы осилите их, если пройденный материал надежно закрепился в вашей голове. Вы чувствуете это? Нет? Тогда без ложного стыда вернитесь к началу книги, ведь повторение — мать учения!

Итоги

- Основу программ составляют три базовые управляющие структуры: линейная последовательность, альтернатива и цикл.
- Альтернатива организуется условными операторами и оператором выбора.
- Для циклов в Паскале предусмотрено три оператора: 1) цикл с проверкой в начале, 2) цикл с проверкой в конце и 3) цикл со счетчиком.
- Программа состоит из ряда секций. Секции описания констант, типов и переменных нужны для размещения данных. Исполняемые секции содержат процедуры, функции и главную программу.
- Обязательной является лишь секция главной программы, прочие секции включают в программу по мере необходимости.
- Секции могут чередоваться произвольно. Но любой объект программы должен быть объявлен до того, как будет использован.
- Основная нагрузка по обработке данных возлагается на процедуры и функции – подпрограммы. Из тела одних подпрограмм вызывают другие подпрограммы, – такие вызовы называют вложенными.
- Передачу данных между подпрограммами предпочтительней выполнять через параметры и имена функций.

А слабо?

А) Найдите две ошибки в следующей программе.

```
var X : TNum;

type TNum = integer;

const A = 10;

begin

      X:= A+B;

end.
```

Б) Напишите булеву функцию **Test** и программу для её демонстрации. Функция должна проверять, делится ли без остатка первое число на второе, например:

```
Writeln( Test(20, 4) );       { true }
Writeln( Test(21, 5) );       { false }
```

В) Напишите целочисленную функцию **Division** для деления первого числа на второе без применения операции **DIV**. Вот примеры вызовов:

```
Writeln( Division(20, 4) );        { 5 }
Writeln( Division(21, 5) );        { 4 }
```

Подсказка: внутри функции вычитайте второе число из первого. Предотвратите деление на ноль (как результат возвращайте ноль). Сделайте два варианта: 1) деление положительных чисел, 2) деление чисел с учетом знака.

Г) Пусть ваша программа распечатает все множители (кроме единицы) введенного пользователем целого положительного числа, например:

```
Введите число: 60
2  2  3  5
```

Д) Напишите функцию для ввода целого числа. Она принимает строку-приглашение и возвращает введенное число, например:

```
      X:= GetNumber('Введите стоимость покупки=');
```

Глава 35
Множества

С малых лет я завидовал обладателям волшебных палочек, ковров-самолетов и прочих волшебных штучек! Смел ли я мечтать о таких игрушках? И вот познакомился с Паскалем... Мы приступаем к мощнейшим средствам этого языка — сложным типам данных. Овладейте ими, и мудреные задачи разрешатся сказочно просто!

В директорском кабинете

Редкий смельчак сунется в директорский кабинет. Но чтобы вникнуть в предстоящую задачу, нам надо тайно проникнуть к директору школы. Вот вам шапка-невидимка (ещё одна волшебная штуковина), вдохните глубже и ступайте на цыпочках за мной.

Мы находим усталого Семена Семеновича перед кипой исчерканных листков с фамилиями учеников. Чем озабочен директор? Сейчас объясню. В начале учебного года Семен Семенович распорядился, чтобы все ученики вступили в какой-либо кружок или спортивную секцию — по желанию. А теперь, спустя пару месяцев, он проверяет исполнение приказа. Директор намерен наказать тех, кто не исполнил распоряжения, и поощрить состоящих в нескольких кружках или секциях. Но, промучившись неделю со списками кружков, он готов уж отказаться от своей затеи, — задача не поместилась в директорской голове. Судите сами: ведь в школе двести пятьдесят учеников! Спасайте Семена Семеновича!

Первым делом, первым делом – оцифровка

Директорскую задачу поручим компьютеру, а тому сподручней орудовать с числами. Заменим фамилии учеников числами, назначив каждому ученику уникальный, несовпадающий с другими, номер. Переход от фамилий к номерам и обратно — простая задачка, её мы оставим Семену Семеновичу. Таким образом, наш входной файл со списками учеников будет содержать по одной строке для каждого кружка, где перечисляются через пробел номера учеников, состоящих в этом кружке. Вот пример входного файла для трех кружков.

```
2 11 4 13
9 17 12 11 3 5 18
14 2 13 15 20
```

Здесь в первый кружок записались 4 школьника, во второй — 7, а в третий — 5 учеников. Как видите, их номера перечислены в произвольном порядке, что затрудняет ручную обработку таких списков. От компьютера требуется выявить номера учеников (от 1 до 250), которых нет в таком файле. Хочется найти простое решение, а оно возможно лишь с применением нового для нас типа данных — множества.

Множества глазами математика

Слово «множество» намекает на большое количество чего-либо. Чего именно? А всё равно! Множества придумали математики, а им безразлично, что считать. Так подать сюда математика, и пусть ответит за всех! Скоро явился математик, взял два кружочка — черный и белый — и, протерев свои толстые очки, стал объяснять. Вот суть его речи.

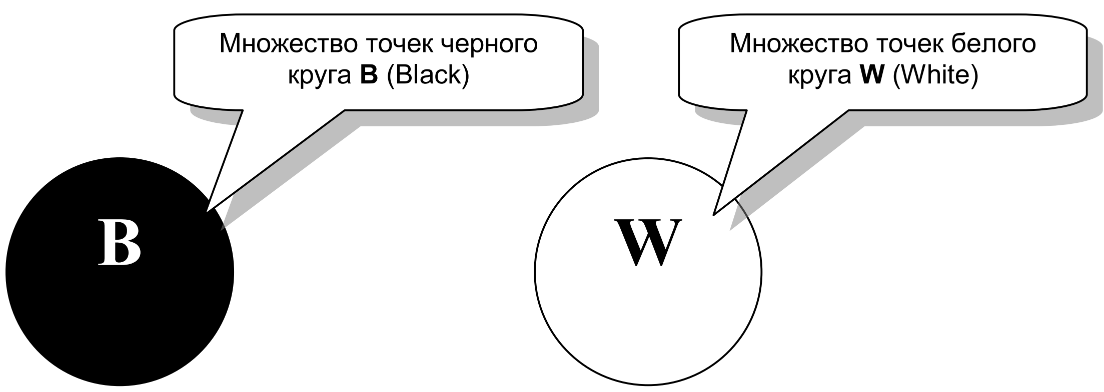


Рис. 80 – Множества точек черного (B) и белого (W) кругов

Вы полагаете, что это кружочки? Нет, друзья, это два множества точек, — одно принадлежит черному кругу, другое — белому. Обозначим первое из них латинской буквой B (от Black — «черный»), а второе буквой W (от White — «белый»). Итак, черные и белые точки этих кружков назовём элементами множеств. Сколько там этих точек? Доказано, что бесконечно много, но к свойствам множеств это не имеет отношения. Что же это за свойства?

Добавление к множеству существующих элементов

Покройте черный круг таким же или меньшим черным кругом, или почеркайте его углем, — заметите разницу? Если на белый круг наложить такой же, или почеркать его мелом, — тоже не увидите изменений. Значит, множество не изменится при добавлении к нему элементов, уже входящих в это множество. На языке математики это свойство выразится так:

B + B = B

или так:

W + W + W = W

Не правда ли, странная арифметика?

Объединение множеств

Продолжим наши мысленные опыты и перекрасим оба круга в серый цвет. Будем считать их теперь одной фигурой, разорванной на части.

Рис. 81 – Объединение непересекающихся множеств G = B + W

Так мы получили новое множество, представляющее сумму или объединение двух предыдущих. Обозначим это новое множество буквой G (от Gray — «серый») и выразим то, что сделали, формулой:

G = B + W

Очевидно, что число точек во вновь образованном множестве равно их сумме в двух исходных. Пока в этом нет ничего интересного, — ведь исходные множества B и W, как говорят математики, не пересекаются. Сблизим круги так, чтобы добиться их частичного перекрытия (рис. 82).

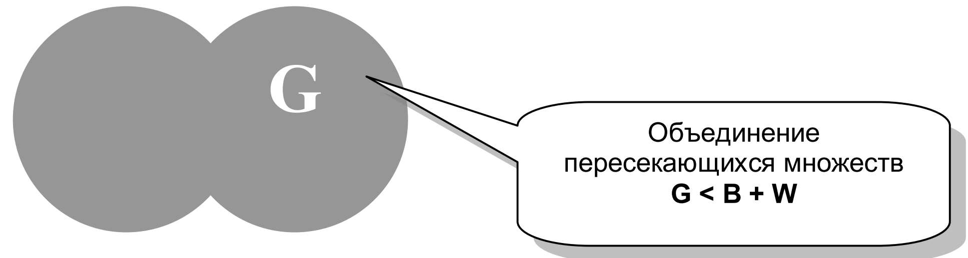


Рис. 82 – Объединение пересекающихся множеств G < B + W

Теперь количество точек в объединенном множестве будет меньше, чем в двух исходных по отдельности:

G < B + W

В общем случае при объединении множеств (как пересекающихся, так и не пересекающихся) соблюдается правило:

G ≤ B + W

Пересечение множеств

Иногда математиков (и не только их) интересует область пересечения множеств, отметим её серым цветом (рис. 83).

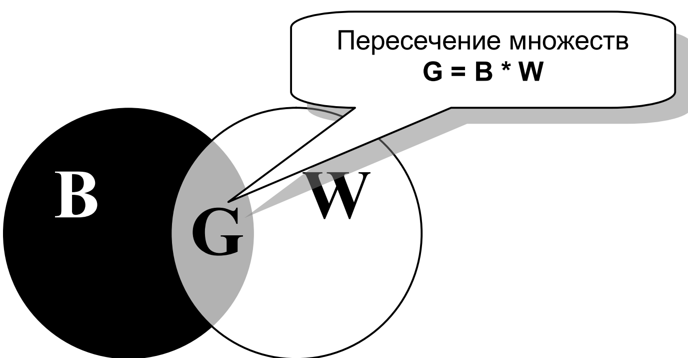


Рис. 83 – Пересечение множеств G = B * W

Операцию пересечения множеств обозначают знаком умножения:

$G = B \bullet W$

Количество точек в пересечении, как понимаете, не может быть больше, чем в любом из исходных множеств B и W. Для этого случая справедливо утверждение: пересечение множеств не больше любого из них:

$B \bullet W \leq B$ и $B \bullet W \leq W$

Вычитание множеств

О солнечных и лунных затмениях слышали все, а кто-то и наблюдал их. Для математика это зримые примеры вычитания множеств; взгляните на рис. 84 — чем не затмения? Серую область можно трактовать как результат вычитания одного круга из другого. На левом рисунке белый круг «отгрыз» часть черного, превратив его в серую область, а на правом — наоборот. Подобающие этим случаям формулы будут таковы:

$G = B — W$ или $G = W — B$

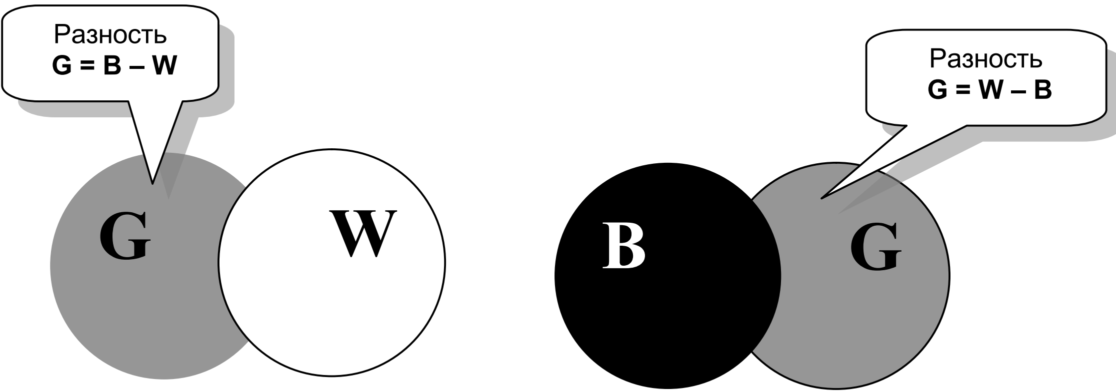


Рис. 84 – Вычитание множеств

А если вычитаемый круг окажется больше того, из которого вычитают, и полностью поглотит его? В алгебре разность получится отрицательной, а здесь? Ничего подобного! При вычитании большего множества из меньшего или равного

ему получается пустое множество, оно обозначается символом Ø. Из пустого множества тоже можно вычитать, и результатом опять будет пустое множество:

(B — B) — B = Ø

(Ø — W) — B = Ø

Вот такими интересными свойствами обладают множества!

Подмножества и надмножества

На рис. 85 белый круг полностью поглощен черным. Тогда говорят, что множество точек белого круга составляет подмножество точек черного. Или так: множество точек черного круга является надмножеством точек белого. Математик выразит это формулой:

B > W

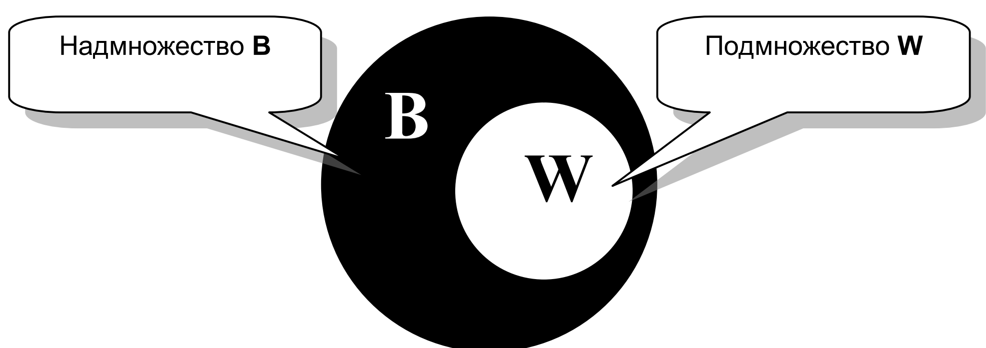


Рис. 85 Надмножество (B) и подмножество (W)

А если круги совпадают и полностью перекрывают друг друга? Тогда говорят, что множества равны, и любое из них является и подмножеством, и надмножеством другого. В общем случае:

если B ≥ W , то B является надмножеством W;

если B ≤ W , то B является подмножеством W.

Числовые множества

Мы рассмотрели несметные множества бесконечно маленьких точек. Но компьютеры ещё не умеют работать с бесконечностями. Так умерим свой аппетит и перейдем к множествам с конечным числом элементов. Поступим так: вместо раскраски кругов расставим на них ряд жирных точек и пронумеруем их числами от 1 до 9 (рис. 86). В ходе последующих опытов нас будут интересовать лишь эти избранные точки (то есть, числа).

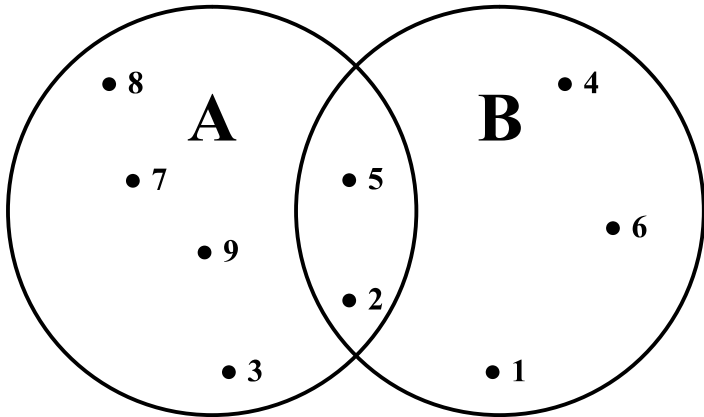


Рис. 86 – Множества чисел

Так мы получили два конечных множества чисел. Одно из них, обозначенное буквой A, содержит числа 8, 7, 9, 3, 5, 2. Другое обозначено буквой B и включает числа 5, 4, 6, 1, 2. Эти множества математики записали бы так:

A = { 8, 7, 9, 3, 5, 2 }

B = { 5, 4, 6, 1, 2 }

Для записи множеств они используют фигурные скобки. Обратите внимание: числа в скобках следуют в произвольном порядке. Это значит, что порядок перечисления элементов множества не важен. Учтите также, что числа 2 и 5 входят в оба множества.

Подобно точкам на круге, каждый элемент числового множества уникален, иными словами, может входить в множество лишь единожды. Вспомните нашу попытку покрасить углем черный круг, — добавление к множеству существующих в нём элементов не изменяет его. Этим же свойством обладают и числовые множества. Например, для нашего случая справедливо следующее:

A + { 8, 7 } = A

Множество A после объединения с множеством {8,7} не изменилось, поскольку уже содержало эти числа.

С числовыми множествами поступают так же, как и с бесконечными: объединяют, пересекают, вычитают и сравнивают. Вот примеры этих операций для нашего случая.

Объединение множеств содержит все числа исходных множеств, при этом повторения (дубликаты) отбрасывают:

G = A + B = { 8, 7, 9, 3, 5, 2 } + { 5, 4, 6, 1, 2 } = { 8, 7, 9, 3, 5, 2, 4, 6, 1 }

Хотя числа 2 и 5 входили в оба исходных множества, в объединении они встречаются по разу.

Пересечение множеств содержит только числа, входящие в оба множества:

A * B = { 8, 7, 9, 3, 5, 2 } * { 5, 4, 6, 1, 2 } = { 5, 2 }

Разность множеств A—B содержит числа, состоящие в множестве A, но отсутствующие в множестве B:

A — B = { 8, 7, 9, 3, 5, 2 } — { 5, 4, 6, 1, 2 } = { 8, 7, 9, 3 }

Разность множеств B—A содержит числа, состоящие в множестве B, но отсутствующие в множестве A:

B — A = { 5, 4, 6, 1, 2 } — { 8, 7, 9, 3, 5, 2 } = { 4, 6, 1 }

Всё это легко проверить по рис. 86.

Мощность множества, полные и неполные множества

Мощность множества — это наибольшее количество элементов, которое может содержаться в нём. В нашем числовом примере мощность множества равна девяти.

Множество, содержащее все возможные свои элементы, называют полным. В нашем случае полным является объединение множеств A+B.

Множество, содержащее не все возможные элементы, является неполным. Так, множества A и B по отдельности — неполные.

Всё это рассказал нам математик. А что же Семен Семенович, или мы забыли о директоре? Нет, конечно, но к директорской задаче мы вернемся после ознакомления с паскалевскими множествами.

Итоги

- Множество – это совокупность различимых объектов (точек, чисел, предметов), которую мы воспринимаем как нечто целое. Отдельные объекты множества называют его элементами.
- К множествам применим ряд операций: объединение, пересечение, вычитание, сравнение.
- Объединение двух множеств содержит по одному элементу из каждого исходного множества.
- Пересечение двух множеств содержит только общие их элементы. Если таких элементов нет, пересечение будет пустым.

- **Разность** множеств содержит элементы уменьшаемого множества за исключением элементов вычитаемого множества.
- Первое множество является **подмножеством** второго, если все элементы первого принадлежат второму. И тогда второе множество будет **надмножеством** первого. Множества **совпадают**, если содержат одни и те же элементы.

А слабо?

А) Полицейская база данных некоторого государства содержит номера всех автомобилей, сгруппированные в ряд множеств. Три множества составлены по типам автомобилей: легковые, грузовые, автобусы. Шесть множеств образованы по цвету автомобилей: множества белых, черных, желтых, красных, синих и зеленых.

- Пересекается ли множество легковых автомобилей с множеством грузовых? А множество желтых автомобилей с множеством черных?
- Может ли быть непустым пересечение множества желтых автомобилей с множеством автобусов?
- Свидетель дорожно-транспортного происшествия сообщил, что с места преступления скрылся **грузовой** автомобиль **синего** цвета. Как вычислить группу подозреваемых автомобилей?
- На улице висит знак: грузовым проезд запрещен. Как определить множество автомобилей, въезд которым разрешен?

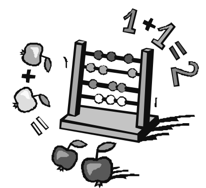
Б) Два государства, назовем их **A** и **B**, спорят о некой территории, — каждое считает ее своей. Нарисуйте на листочке предполагаемую карту, заштрихуйте спорную область, а затем объясните:

- Как вычислить спорную область государств?
- Как вычислить бесспорную область, включая оба государства?
- Заштрихуйте область, отвечающую формуле `G = (A–B) + (B–A)`.
- Заштрихуйте область, отвечающую формуле `G = A+B – A•B`. Совпадает ли она с той, что вычислена по предыдущей формуле?

В) Дайте ответы на следующие вопросы.

- Является ли множество ваших одноклассников подмножеством учеников вашей школы?
- Пересекается ли множество ваших друзей с множеством ваших одноклассников?
- Является ли множество ваших друзей подмножеством ваших одноклассников?

Глава 36
Множества в Паскале



Зная силу математических множеств, Никлаус Вирт — «отец» языка Паскаль — ввел в язык тип данных множество и предусмотрел операции с ним.

Элементами множеств здесь могут быть числа, символы и булевы данные — то есть порядковые типы данных размером в один байт. Стало быть, мощность множеств в Паскале не превышает 256.

Объявление множеств

Множества объявляются конструкцией вида:

SET OF <диапазон или тип>

Вот примеры объявления переменных типа множество:

```
      { объявление множества }      { возможные элементы множества }
var   SN1 : set of 10..100;         { числа от 10 до 100 }
      SN2 : set of byte;            { числа от 0 до 255 }
      SC1 : set of 'a'..'z';        { только малые латинские буквы }
      SC2 : set of Char;            { все символы }
```

Поскольку мощность множеств в Паскале не превышает 256, множества **SET OF BYTE** и **SET OF CHAR** представляют множества предельной мощности.

Присвоение значений множествам

Переменным типа множество присваивают значения выражений того же типа, вот примеры таких операторов:

```
      SN1:= [10, 20, 50];           { содержит три элемента }
      SN2:= [11..20, 51..60];       { содержит 20 элементов }
      SN2:= [0..255];               { содержит 256 элементов от 0 до 255 }
      SN2:= SN1;                    { копия другого множества }
      SC1:= ['z', 'y', 'x'];        { содержит три элемента }
      SC2:= ['0'..'9'];             { содержит 10 элементов }
```

Как видите, для записи множеств в Паскале используют квадратные скобки, а не фигурные. Что позволено в записи множеств, и что запрещено?

Подряд идущие элементы можно заменять диапазоном с указанием крайних значений. Допустимо перечислять элементы в произвольном порядке и даже

вставлять дубликаты, — они всё равно будут отброшены. Вот примеры трех совершенно одинаковых по результату операторов:

```
SN1:= [5..8];          { множество задано диапазоном }
SN1:= [8, 7, 6, 5];    { то же множество, но в другом порядке }
SN1:= [5..8, 6, 6];    { трижды указано число 6, дубликаты будут отброшены }
```

Множеству любого типа можно присвоить пустое значение, например:

```
SB1:= [];         SN1:= [];         SC1:= [];
```

Пустое множество изображается парой квадратных скобок, между которыми ничего нет. Нельзя считать пустым множество [0], поскольку оно содержит один элемент — число ноль.

Элементами множеств могут быть только значения переменных и выражений соответствующего типа:

```
var   k, n : byte;      c: char;
      . . .
      k:= 10;     n:= 20;
      SN1:= [1..k, n+5];          { 1..10, 25 }
      c:= 'm';
      SC1:= [c, 'a', 'b'];        { 'm', 'a', 'b' }
```

Компилятор не позволит включать в множество элементы, не относящиеся к нему, а также смешивать элементы разных типов, вот примеры таких ошибок:

```
SN1:= [5..200];        { в объявлении SN1 указан диапазон от 10 до 100 }
SC1:= ['a', 'b', 5];   { вместо символа '5' указано число 5 }
```

Операции с множествами

В Паскале предусмотрены три известные вам вычислительные операции с множествами, а также сравнение множеств и проверка на вхождение элемента в множество.

Вычислительные операции — объединение, пересечение и вычитание — записывают на Паскале так:

```
SN2:= [3, 7] + [5, 2];         { объединение = [2, 3, 5, 7] }
SN2:= [2..10] * [8..20];       { пересечение = [8, 9, 10] }
SN2:= [2..10] – [8..20];       { разность = [2..7] }
```

Множества, объединенные знаками операций и круглыми скобками, образуют выражение, например:

```
SN2:= (SN1 + [0..15]) * SN2;
```

Выражения, составленные из множеств, очень похожи на выражения из чисел, но вычисляются по другим правилам. Это обманчивое сходство может спровоцировать ошибку — смешение в одном выражении чисел и множеств. Предположим, вы хотите добавить к множеству число, содержащееся в переменной **K**. Следующее выражение будет неверным:

```
SN1:= SN1 + K;          { сложение множества с числом – ошибка }
```

Правильно будет так:

```
SN1:= SN1 + [ K ];      { добавляется множество из одного элемента }
```

Разумеется, за ошибками такого рода присматривает компилятор, проверьте его реакцию на практике.

Сравнение множеств

Множества можно сравнивать между собой, получая в результате булево значение — **TRUE** или **FALSE**.

Два множества равны, если содержат одни и те же элементы:

```
if SN1 = SN2      then  …     else   …
```

Множества неравны, если одно из них содержит, хотя бы один элемент, которого нет в другом:

```
if SN1 <> [15, 17, 19]  then  …     else  …
```

Проверка на подмножество (<=) отвечает на вопрос: все ли элементы первого множества входят во второе?

```
if SN1 <= SN2     then  …     else   …
```

Проверкой на надмножество (=>) выясняют, все ли элементы второго множества входят в первое:

```
if SN1 => SN2     then  …     else   …
```

Проверка на вхождение элемента в множество (операция IN)

Входит ли некоторый элемент в множество? Это можно выяснить так:

```
var N : byte;     S : set of byte;
      . . .
      if ([N] * S) <> [] then { N входит в S } else { не входит }
```

Понятно, что, если число **N** входит в множество **S**, то пересечение **[N]*S** не будет пустым. Но проще выяснить это операцией **IN** — она введена специально для этого. Операция дает **TRUE**, если значение перечислимого типа входит в данное множество, например:

```
      if N in S   then { N входит в S } else { не входит }
      if 20 in S  then { 20 входит в S } else { не входит }
```

Решение директорской задачи

Вернемся к временно покинутому директору Семену Семеновичу. Напомню стоящую перед нами задачу: есть текстовый файл, каждая строка которого содержит список номеров учеников, состоящих в некотором кружке:

```
2 11 4 13
9 17 12 11 3 5 18
14 2 13 15 20
```

Надо составить список нигде не числящихся разгильдяев.

Можно ли воспринимать эти списки как множества? Вероятно, да, судите сами:

- каждый список содержит номер ученика не более одного раза (ошибочные повторные записи всё равно отбросят);
- порядок следования в списке не важен;
- список может быть пустым (если никто не записался в этот кружок).

Хорошо, а будет ли множеством список всех учеников школы? Конечно. Такое множество будет полным, поскольку содержит все возможные элементы. А раз так, директорскую задачку решим через множества.

Множество тех, кто записался хотя бы в один кружок, найдем объединением отдельных множеств-кружков **(S1 + S2 + S3)**. Вычтя это объединение из

полного множества учеников, получим множество уклонившихся. Вот и всё решение! На Паскале это запишется так:

```
var   R, S1, S2, S3 : set of 1..250;
begin
      S1:= [ 2, 11, 4, 13 ];              { 1-й кружок }
      S2:= [ 9, 17, 12, 11, 3, 5, 18 ];   { 2-й кружок }
      S3:= [ 14, 2, 13, 15, 20 ];         { 3-й кружок }
      R:= [1..250] - (S1 + S2 + S3);      { R - множество уклонившихся }
end.
```

Подчеркнутое выражение в скобках — это множество учеников, состоящих хотя бы в одном кружке. Итак, решение задачи вместилось в одну строчку! Нет, не зря мы терпели математика и корпели над множествами!

Показанное выше решение — это работающая программа, которую можно запустить в пошаговом режиме, и через отладчик увидеть результат. Сделайте это. А как быть с вводом и выводом множеств? Ведь исходные данные хранятся в файле, а результат — переменную **R** — тоже надо вывести в файл или на экран. Вот этим мы и займемся в следующей главе.

Итоги

- Множества – это инструмент, взятый в Паскаль из математики.
- В Паскале применяют конечные множества, элементами которых могут быть числа, символы и булевы значения. Мощность множеств в Паскале не превышает 256.
- В Паскале предусмотрен ряд операций с множествами: объединение, пересечение, вычитание, сравнение, а также проверка на вхождение элемента в множество.
- Сравнение двух множеств дает булев результат, который используют в условных и циклических операторах.
- Операция **IN** – удобное средство для проверки вхождения одного элемента в множество, она тоже дает булев результат.

А слабо?

А) Найдите ошибки в следующих операторах.

```
type TNumbers = set of 1..300;
      TChars = set of char;
      TBytes = set of byte;

var c1, c2 : TChars;
      b1, b2 : TBytes;
begin
      c1:= [1..9];
      c2:= ['1'..'9'];
      c2:= c2 + ’0’;
      c2:= c2 + [0];
      b1:= c1;
      b2:= b1 + [1,7,3];
      Writeln(b1=b2);
      Writeln(1 in b2);
      Writeln([1] in b2);
      Writeln(b1 in b2);
end.
```

Б) Напечатайте 20 случайных чисел в диапазоне от 1 до 50 так, чтобы каждое число встретилось в распечатке лишь по разу. Подсказка: после генерации числа функцией **Random** проверьте его на вхождение в множество уже напечатанных чисел.

В) Введите программу решения директорской задачи (см. предыдущую страницу), а затем запустите её в пошаговом режиме (клавишей *F7*). Перед запуском вставьте все переменные в окно обзора переменных «Watch» и проследите за их изменением. Напомню, что о средствах отладки рассказано в главе 21.

Глава 37
Ввод и вывод множеств

Мы узнали о множествах и приспособили их к директорской задаче. Чтобы покончить с нею, сделаем ещё пару пустяков: организуем ввод и вывод множеств. Для ввода-вывода строк и простых типов данных годятся процедуры **Read[ln]** и **Write[ln]**. Но сейчас всё не так просто, — эти процедуры не способны работать ни с множествами, ни с другими сложными типами данных. Однако ж «нормальные герои всегда идут в обход», — пойдем так и на этот раз.

Вывод множества в текстовый файл

Начнем с вывода числового множества на экран (или в файл, что одно и то же). Так мы получим средство для последующей проверки вводимых множеств.

Раз уж процедура **Writeln** не печатает множество одним махом, выведем каждый его элемент по отдельности — ведь это обычные числа или символы. Проверяя все возможные элементы множества, будем печатать лишь те, что входят в него — в этом основная идея. Напомню, что для такой проверки подходит операция **IN**. Дополнив её циклом со счетчиком, соорудим несложную процедуру распечатки числового множества. Вот она вместе с программой для её проверки.

```
{ P_37_1 – вывод множества в файл }

type  TSet = set of 1..255;         { объявление типа «множество» }

      {----- Процедура вывода множества в файл -----}
procedure WriteSet(var aFile: text; const aSet : TSet);
var k : integer;
begin
      for k:=1 to 255 do           { цикл по всем элементам множества}
            if k in aSet           { если K входит в множество }
                  then Write(aFile, k:4);      { печатаем в строке }
      Writeln(aFile);   { по окончании – переход на следующую строку }
end;
```

```
      {----- Программа для проверки процедуры WriteSet -----}
var   S1 : TSet;        F: text;
begin
      Assign(F, ''); Rewrite(F);    { связываем файл с экраном! }
      S1:= [3, 10, 25];             { значение множества }
      WriteSet(F, S1);              { печатаем }
      Readln;
      Close(F);
end.
```

В первой строке объявлен тип данных **TSet**, он может содержать целые числа от 1 до 255. Процедура распечатки **WriteSet** принимает по ссылке два параметра: файловую переменную и множество, которое надо распечатать. Внутри процедуры работает цикл **FOR**, перебирающий все возможные элементы множества. Те из них, что содержатся в нём, печатаются в текущей строке. По завершении цикла оператор **Writeln** переводит позицию записи на следующую строку файла.

Обратите внимание: множество передано в процедуру по ссылке **CONST**. Передача в процедуры множеств, строк и других сложных типов данных по ссылкам **CONST** и **VAR** — это обычная практика. Так повышается скорость работы программ и уменьшается объём памяти, занимаемый параметрами.

Теперь взгляните на оператор **Assign(F,''),** который назначает файловой переменной пустое имя файла. Так файловая переменная связывается с экраном дисплея (при выводе данных), либо с клавиатурой (при вводе). А когда вам потребуется вывести результаты в дисковый файл, достаточно будет задать нужное имя файла, не меняя процедуры **WriteSet** (этот прием — подстановка пустого имени — не работает в Pascal ABCNet).

Примечание. В современные версии Паскаля (Delphi) для обработки множеств введён вариант цикла **FOR-IN-DO**. С ним вывод множества станет ещё проще:

```
      for k in aSet do Write(aFile, k:4);
```

Ввод множества из текстового файла.

Разобравшись с распечаткой множества, перейдем к вводу его из файла. Есть соображения на этот счет? Здесь пригодится опыт чтения чисел из строки текстового файла, — вспомните обработку классного журнала. Добавить число к множеству мы тоже умеем: для этого надо объединить его с множеством, состоящим из добавляемого числа. На этих идеях построена процедура ввода, показанная ниже вместе с тестирующей её программой.

```
{ P_37_2 – ввод и вывод числового множества }
type  TSet = set of 1..255;   { объявление типа «множество» }
      {----- Процедура чтения множества из файла -----}
procedure ReadSet(var aFile: text; var aSet : TSet);
var   k : integer;
begin
      While not Eoln(aFile) do begin     { пока не конец строки }
            Read(aFile, K);              { читаем очередное число }
            aSet:= aSet+[K];             { и добавляем к множеству }
      end;
      Readln (aFile);                    { переход на следующую строку }
end;
      {----- Процедура распечатки множества в файл -----}
procedure WriteSet(var aFile: text; const aSet : TSet);
var k : integer;
begin
      for k:=1 to 255 do           { цикл по всем элементам множества}
            if k in aSet           { если входит в множество }
                  then Write(aFile, k:4); { печатаем в строке }
      Writeln(aFile);        { по окончании переход на следующую строку }
end;

      {----- Программа для проверки процедуры ввода -----}
var   S1 : TSet;        F, D: text;
begin
      Assign(F, ''); Rewrite(F);   { вывод на экран }
      Assign(D, ''); Reset(D);     { ввод с клавиатуры }
      S1:= [];                     { перед вводом опустошаем множество }
      ReadSet(D, S1);              { вводим множество из файла }
      WriteSet(F, S1);  Readln;    { распечатаем для проверки }
      Close(F);   Close(D);
end.
```

Полагаю, что комментарии поясняют всё. Обязательно проверьте работу этой программы. Учтите, что вводить данные вы будете с клавиатуры: напечатайте в одной строке несколько чисел, разделяя их пробелами, а затем нажмите *Enter*.

Директорская задача, первый вариант

Освоив ввод и вывод множеств, мы вплотную подошли к полному решению директорской задачи. Напомню, что суть решения заключается всего в одном операторе:

```
      R:= [1..250] - (S1 + S2 + S3);
```

Теперь добавим ввод и вывод множеств. Чтобы не занимать место повторами показанных ранее процедур, я представлю решение в целом.

```
{ P_37_3 - решение директорской задачи, вариант 1 }

const CMax = 20;                    { мощность множества, реально 250 }
type  TSet = set of 1..CMax;  { объявление типа «множество» }

procedure WriteSet(var aFile: text; const aSet : TSet);
{ взять из P_37_2 }

procedure ReadSet(var aFile: text; var aSet : TSet);
{ взять из P_37_2 }

var   R, S1, S2, S3 : TSet;
      FileIn, FileOut: text;

begin {----- Главная программа -----}
      { Открытие входного файла }
      Assign(FileIn, 'P_37_3.in'); Reset(FileIn);
      { Создание выходного файла }
      Assign(FileOut, 'P_37_3.out'); Rewrite(FileOut);
      { Ввод множеств из входного файла }
      S1:=[];     ReadSet(FileIn, S1);
      S2:=[];     ReadSet(FileIn, S2);
      S3:=[];     ReadSet(FileIn, S3);
      R:= [1..CMax] - (S1+S2+S3);   { Решение }
      WriteSet(FileOut, R);         { Вывод решения в выходной файл }
      Close(FileIn);    Close(FileOut);
end.
```

Для ввода и вывода множеств используем дисковые файлы, поэтому оператор **Readln** в конце программы не нужен. Для облегчения проверки я уменьшил число учеников — константу **CMax** — с 250 до 20. При тестировании программы входной файл содержал следующие строки:

```
2 11 4 13
9 17 12 11 3 5 18
14 2 13 15 20
```

А в выходной файл попали следующие числа:

```
   1   6   7   8  10  16  19
```

Легко убедиться в том, что никто из этих учеников не состоит в кружках.

Директорская задача, второй вариант

Итак, задача решена, но директор не вполне доволен. Сейчас возможности программы ограничены тремя кружками и двадцатью учениками. При изменении этих данных надо менять и программу, — мы избавимся от этого недостатка.

Во-первых, слегка изменим входной файл. Пусть первая его строка содержит количество учеников в школе; и тогда файл станет таким:

```
20
2 11 4 13
9 17 12 11 3 5 18
14 2 13 15 20
```

Во-вторых, отведем для участников кружков не три, а лишь одну переменную типа множество. Затем, по мере чтения строк файла, будем накапливать в этой переменной всех, кто состоит в кружках. Цикл чтения завершится по достижении конца входного файла. Вот и все изменения, посмотрите на второй вариант (процедуры ввода и вывода множеств только обозначены).

```
{ P_37_4 - решение директорской задачи, вариант 2 }

type  TSet = set of byte;     { объявление типа «множество» }

{ Здесь надо поместить процедуры ввода и вывода множеств }
procedure WriteSet(var aFile: text; const aSet : TSet);
{ взять из P_37_2 }

procedure ReadSet(var aFile: text; var aSet : TSet);
{ взять из P_37_2 }
```

```
var   R, S : TSet;
      FileIn, FileOut: text;
      N: integer; { общее число учеников }
begin
      Assign(FileIn, ' P_37_4.in'); Reset(FileIn);
      Assign(FileOut, ' P_37_4,out'); Rewrite(FileOut);
      Readln(FileIn, N);      { читаем общее число учеников }
      S:= [];     { очищаем перед вводом }
      { пока не конец файла, объединяем участников всех кружков }
      while not Eof (FileIn) do ReadSet(FileIn, S);
      R:= [1..N] - S;         { Решение }
      WriteSet(FileOut, R);
      Close(FileIn); Close(FileOut);
end.
```

Согласитесь, программа стала и гибче, и проще. Однако к первому её варианту мы ещё вернемся.

Итоги

- Стандартные процедуры ввода и вывода не способны вводить и выводить множества, для этого создают специальные процедуры.
- Вывод (распечатка) множества выполняется циклом со счетчиком, внутри которого проверяется вхождение каждого элемента в множество.
- Ввод множества из текстового файла основан на операции объединения по отдельности прочитанных элементов.

А слабо?

А) Напишите процедуры для ввода и вывода множества символов. Можно ли здесь для счетчика цикла применить символьную переменную?

Б) Напишите функцию, принимающую числовое множество и возвращающую количество содержащихся в нём элементов.

В) На основе первого варианта директорской программы придумайте способ поиска учеников, записавшихся более чем в один кружок. Или слабо?

Г) Напишите две функции, принимающие строку и возвращающие:

- строку, в которой символы исходной строки встречаются лишь по разу и следуют в алфавитном порядке, например «PASCAL» → «ACLPS»;
- то же, но порядок следования символов такой же, как в исходной строке, например «PASCAL» → «PASCL».

Глава 38
Множества «в бою»

Множества, множества... — заполучив столь острое оружие, удержимся ли не пустить его в ход? Вот ещё несколько задач, — мы изрубим их в капусту!

Активисты, шаг вперед!

Прежде всего, отдадим долги Семену Семеновичу. Мы обещали директору выявить разгильдяев, что отлынивают от кружков, и сдержали слово. Теперь найдем активистов, состоящих в нескольких кружках. Откуда подступиться к этой задаче?

Положим для простоты, что в школе лишь три кружка, их списки представлены множествами **S1**, **S2** и **S3**. Выявить тех, кто состоит одновременно в кружках **S1** и **S2** легко, — достаточно найти пересечение **S1*S2**. Точно так же поступим с другими парами: **S1** и **S3**, **S2** и **S3**. Объединив все три пересечения, мы выявим интересующих нас школяров. Итак, решение задачи выразится формулой.

```
R := S1*S2 + S1*S3 + S2*S3;
```

Попадут ли в это множество ученики, состоящие во всех трех кружках? Если да, то, как их отделить от прочих? Придумайте, как выявить тех, кто состоит:

- в трех кружках:
- в двух кружках и не более;
- только в одном из кружков.

Надеюсь, что с этим проектом, назовем его P_38_1, вы справитесь сами, желаю успеха!

Подвиг контрразведчика

Контрразведка некоторого государства обнаружила утечку информации из лабораторий секретного учреждения. Для поимки шпиона позвали сыщика Шерлока Ивановича Холмского. Первым делом, он попросил списки сотрудников лабораторий. Лаборатории именовались латинскими буквами: A, B, C и так далее, причем некоторые сотрудники допускались в несколько лабораторий. Шерлок Иванович оцифровал списки, заменив фамилии сотрудников их табельными номерами, то есть, уникальными числами. Затем сгруппировал эти числа по лабораториям и составил табл. 6.

Табл. 6 – Исходные данные для «вычисления» завербованного сотрудника

Лабо—ратория	Номера сотрудников, допущенных в лабораторию
A	1 2 4 5 9 11 13 15 22 23 24 25 27 30 31 37 41 42 43 44 45 46 48 50 51 56 64 70 72 73 74 75 76 77 82 84 86 87 89 92 95 97 98 101 102 103 104 105 106 107 108 111 113 116 117 118 124 125 127 130 132 133 134 138 143 144 145 147 149 150
B	16 21 22 23 24 25 26 27 28 29 31 33 35 37 39 41 44 47 49 50 51 52 54 55 56 57 59 61 62 65 66 69 70 71 72 77 78 79 81 83 84 85 91 92 93 94 95 96 98 100 101 103 107 108 109 112 113 115 117 118 119 121 122 124 129
C	1 3 5 9 12 19 22 25 33 34 41 42 46 50 52 55 56 57 58 59 61 66 69 72 80 81 82 84 87 88 94 97 99 100 101 102 112 119 121 123 125 129 134 137 138 139 149 152 153 154 155 157 158 165 166 168 171 172 180 184 185 190 193 194 198 199 205 213 216 220
D	5 6 7 8 9 10 11 12 13 14 16 18 21 22 23 24 27 28 29 30 31 32 34 35 38 40 41 42 43 44 45 46 47 48 51 52 53 54 55 57 58 59 60 61 62 63 64 65 66 67 70 71 73 74 75 76 78 79 80 81 82 84 85 86 88 89 91 92 93 94 95 96 97 98 99 100 104 105 106 107 108 111 112 113 115 116 117 118 119 120
E	10 15 16 26 33 40 42 44 50 53 65 67 74 79 82 83 85 87 90 91 93 99 106 108 110 120 121 124 125 132 135 146 148 149 151 156 157 158 163 166 168 169 171 175 183 184 189 195 197 205 206 207 216 220 221 225 226 227 241 244
F	8 12 21 25 26 29 30 31 34 48 49 50 52 55 59 60 62 70 71 73 83 85 90 91 92 93 94 96 97 99 100 102 103 104 105 106 108 119 121 122 124 127 128 130 132 141 142 144 156 160 165 166 169 171 173 176 179 191 192 195 199 200 207 209 220 221 222 224 226 229 233 234 236 239 240
G	23 24 26 27 29 30 35 36 41 42 44 45 46 49 52 55 56 58 60 61 63 64 65 68 72 74 76 77 81 82 86 87 88 90 93 94 95 96 97 98 100 101 102 107 108 109 112 113 114 115 117 120 123 127 132 133 135 137 138 143 145 146 147 150 152 155 156 159 161 162 163 164 165 168 170 172 177 178 179 180
H	15 17 19 20 21 22 23 26 28 29 30 32 33 34 36 38 41 42 44 45 46 48 49 52 57 60 62 65 66 68 73 74 77 78 83 84 85 88 89 90 91 92 95 96 97 98 99 100 101 102 103 104 107 108 115 116 118 127 128 129 130 131 134 135 136 137 139 145 146 150 151 152 154 157 160 161 164 166 167 172 173 177 178 179 180 182 185 188 189 190 193 195 197 204 207

Дальнейшее разбирательство показало, что секреты просачивались только из лабораторий A, D, G и H (в таблице они выделены серым). При этом секреты

остальных лабораторий (B, C, E и F) остались нетронутыми. Это направило дедуктивную мысль в правильное русло.

«Очевидно, — рассуждал Шерлок Иванович, — шпионить может тот, кто допущен в «дырявые» лаборатории. Из этого круга исключим тех, кто работает в нетронутых лабораториях, иначе их секреты тоже стали бы известны». Рассудив так, Шерлок Иванович достал ноутбук, и через 30 минут агент был вычислен, — подозреваемым оказался сотрудник с номером 45. Установленная за ним слежка подтвердила подозрение, и шпион был задержан.

Слабо ли вам повторить подвиг контрразведчика? Воспроизведите программу, написанную Шерлоком Ивановичем, я подскажу вам только её первую строку:

```
{ P_38_2 - подвиг контрразведчика }
```

В тридевятом царстве

Это случилось на затерянном в океане материке, что носил на себе несколько царств-государств. Жители материка — те ещё скряги — тратили для названий своих стран всего по одной букве: A, B, C и так далее. И мы будем их так называть. Границами стран служили каналы, специально для того прорытые; каналы были пронумерованы. Некоторые страны выходили к океану, берега которого тоже были пронумерованы и служили границами.

Самым могущественным было царство A. Однако, ввиду его обширности и частых политических перемен, тамошний государь никак не мог уяснить точные границы своей страны. Он толком не знал даже ближайших соседей, — сведения были самыми разноречивыми. Когда терпение монарха лопнуло, он повелел своим инженерам запустить спутник, который бы исследовал границы и внес ясность в этот вопрос.

Слово царя — закон, и вскоре спутник кружился на орбите. С высоты ясно наблюдались берега океана и каналы, составлявшие границы царств. Рис. 87 показывает то, что «увидел» спутник. Буквами обозначены названия стран, а числами — участки границ. В центре континента темным цветом выделено обширное царство A. К нему примыкают несколько стран, отмеченные серым, — это его соседи. Страны, примыкающие к царству A уголками своих границ, соседями не считаются. Они и все прочие «не соседи» отмечены белым цветом, а вокруг — океан.

Увы, примитивная техника тех лет не смогла отправить на землю эту фотографию. Спутник передал лишь номера границ каждого государства в виде текстового файла, содержащего строчки чисел.

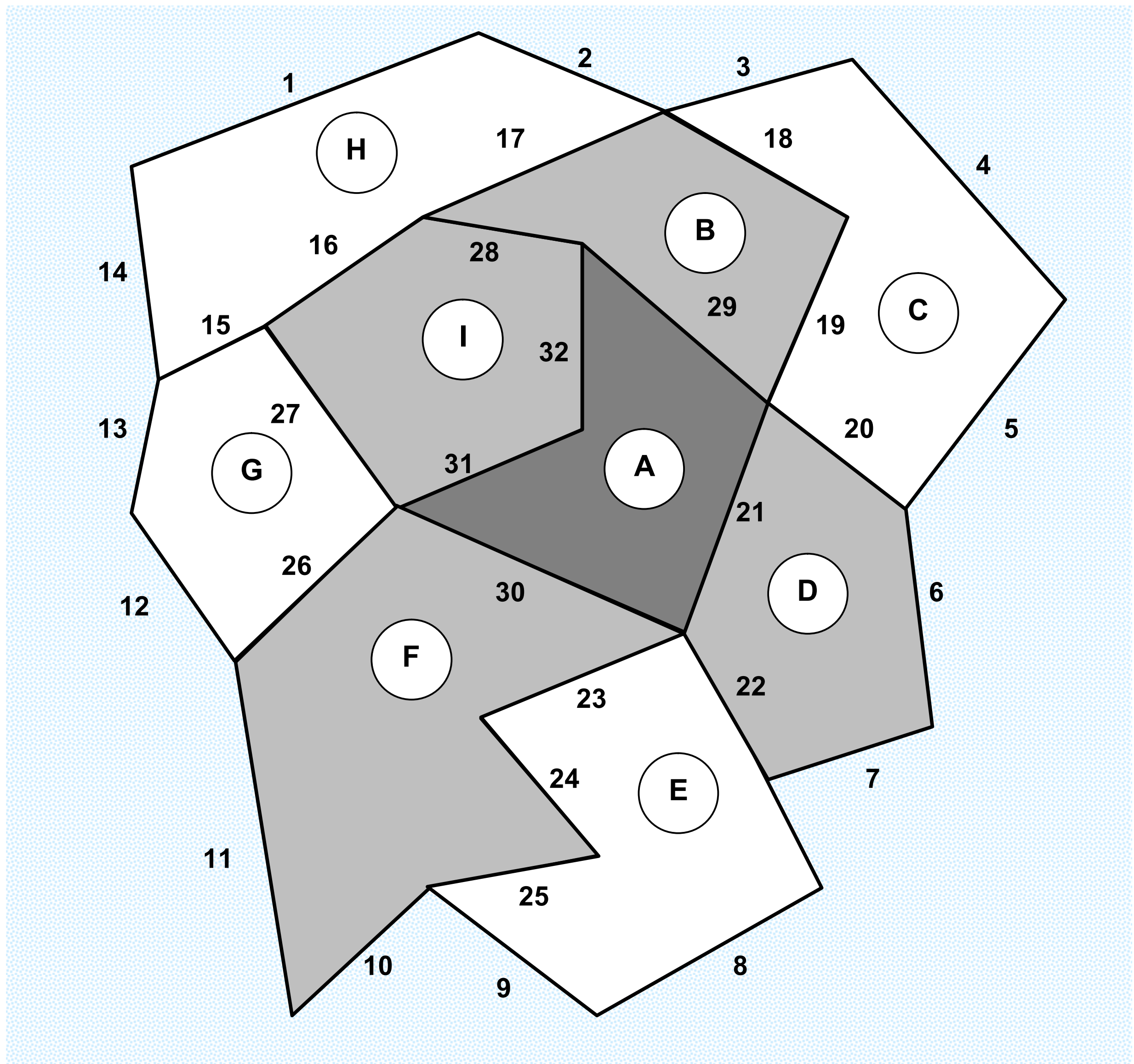


Рис. 87 – Вид на материк из космоса

Выдернув из принтера ещё теплую распечатку файла, первый министр примчался во дворец, протянул листок монарху и покорно припал к подножию трона. Царь встрепенулся, стал разглядывать бумажку, вертеть её так и сяк, и даже на зуб попробовал. Наконец терпение государя иссякло: «Болван, — обратился он к министру, — покажи тут наших соседей. Что? Не можешь? Так проваливай с глаз долой!». И смятая распечатка угодила в лицо министра. «А ведь хотел, как лучше…» — стучало в башке убегающего премьера. «А получилось, как всегда!» — догнал его вопль взбешённого монарха.

Куда податься бедолаге? Разумеется, к самому умному — к придворному программисту. «Выручай, браток, я тебе премию выпишу!». Инженеры, создавшие спутник, тоже не остались в стороне и растолковали программисту суть проблемы. Расправив скомканную царской рукой бумагу, Ник — так звали придворного программиста — увидел вот что.

```
29 21 30 31 32
17 18 19 29 28
3 4 5 20 19 18
6 7 22 21 20
8 9 25 24 23 22
10 11 26 30 23 24 25
12 13 15 27 26
14 1 2 17 16 15
16 28 32 31 27
```

Каждая строка этого файла, — объяснили инженеры, — перечисляет границы некоторого царства: первая строка — царства A, вторая — царства B и так далее. Имена стран в файле не указаны, но подразумевается их алфавитный порядок. Надо составить список стран, которые соседствуют с нашей страной A — первой в этом списке.

Друзья, отложите книгу и попытайтесь решить эту интересную задачу. В случае успеха, я похлопочу за вас при дворе!

А пока вы раздумываете, я исполню свой долг перед историей и покажу решение заморского коллеги. Ник сразу понял, что имеет дело с двумя видами множеств: множеством границ, обозначенных числами, и множеством стран, обозначенных буквами (вы помните, что страны именовались буквами?). Парень смекнул, что две страны соседствуют тогда, когда пересечение множеств их границ не пусто (это значит, что у них есть общие границы). Дальше его мысли устремились так быстро, что пальцы едва успевали тыкать по клавишам. Вот плод его труда.

```
{ P_38_3 - поиск стран-соседей      }

type  TBoundSet = set of byte;            { множество границ }
      TStateSet = set of Char;            { множество стран }

      {----- Распечатка множества стран (символов) -----}
procedure WriteCharSet(var aFile: text; const aSet : TStateSet);
var c : char;
begin
      for c:='A' to 'Z' do    if c in aSet then Write(aFile, c:2);
      Writeln(aFile);
end;
```

```
      {----- Ввод множества границ (чисел) -----}
procedure ReadSet(var aFile: text; var aSet : TBoundSet);
var k : integer;
begin
      While not Eoln(aFile) do begin
            Read(aFile, K);   aSet:= aSet+[K];
      end;
      Readln (aFile);
end;

var   FileIn, FileOut: text;
      R: TStateSet;           { множество соседей (результат) }
      SA, S : TBoundSet;      { границы царства «A» и прочих }
      State: char;            { буква с названием очередной страны }

begin       {----- Главная программа -----}
      Assign(FileIn, 'P_38_3.in'); Reset(FileIn);
      Assign(FileOut, ''); Rewrite(FileOut);
      R:= [];  SA:=[];  State:='A'; { начнем с царства «A» }
      ReadSet(FileIn, SA);    { из первой строки читаем границы для «A»}
      while not Eof (FileIn) do begin     { цикл по странам }
            State:= Succ(State);          { буква следующей страны }
            S:=[]; ReadSet(FileIn, S);    { читаем границы страны }
            { если граничит с царством «A», добавляем к результату }
            if S*SA <> [] then R:= R + [State];
      end;
      WriteCharSet(FileOut, R);  Readln; { вывод результата }
      Close(FileIn);   Close(FileOut);
end.
```

Программа Ника вычислила, что царство A соседствует с царствами B, D, F, I. Со временем проверка на местности это подтвердила.

Царь щедро наградил программиста, но история на этом не закончилась. О великом научном успехе скоро знала и последняя собака на материке. Но больше других этот успех заинтересовал купцов, плативших пошлины при пересечении границ. Они явились к Нику с предложением, от которого тот не смог отказаться. Хотите продолжения сказки? — оно ждёт вас в главах 49, 57 и 58.

Решето Эратосфена

Древние греки не знали, что они древние. И компьютеров тоже не знали, зато дышали бодрящим морским воздухом, коротая досуг в философских и

математических размышлениях. Греческий досуг оказался не таким уж пустым, — иные задачки, придуманные под ласковый шепот волн, не решены по сию пору! Одна из них — вычисление простых чисел.

Прежде всего, выясним, что это за числа? Простым называют число, которое делится без остатка лишь на само себя и единицу. Все прочие числа являются составными. Возьмем, к примеру, числа от 1 до 10 и подчеркнем среди них составные:

```
1 2 3 4 5 6 7 8 9 10
```

Здесь отмечены составные числа 4, 6, 8, 9 и 10, — они делятся без остатка либо на 2, либо на 3. Оставшиеся числа 1, 2, 3, 5 и 7 являются простыми.

Кто-то из греков задался вопросом: можно ли вычислить очередное простое число, если известны все предыдущие? Например, исходя из того, что числа 1, 2, 3 и 5 простые, определить следующее простое число (7). Как ни мудрили мудрецы, такой формулы или алгоритма пока не придумали! Но усилия в этом направлении породили целые отрасли математики, — вот такой полезный неуспех!

Размышлял над задачей и грек Эратосфен. Он тоже не решил её, однако нашел остроумный способ отсеивать простые числа, не превышающие некоторого числа **N**. Вот суть его идеи.

Положим, мы ищем простые числа не превышающие 20. Выпишем на морском песочке в ряд числа с 1 до 20. Первые два числа — 1 и 2 — простые, их не тронем, а среди остальных сотрем каждое второе, то есть 4, 6, 8 и так далее.

Затем находим первое нестертое число — это три. Сотрем каждое третье после тройки: 6, 9, 12, 15 и 18 (хотя часть из них уже стерта, лишний раз это сделать не повредит). Повторяя процедуру, находим следующее нестертое число — это пять. Стираем каждое пятое после пятерки: 10, 15, 20 (хотя все они уже стерты). Достигнув середины этого списка — числа 11, остановимся. Дальше двигаться нет смысла, поскольку на песке остались лишь простые числа.

Примечание. Если говорить точнее, лучше остановиться на числе, которое составляет корень квадратный из числа **N**, в данном случае это 5. Но для упрощения задачи мы будем обрабатывать больше чисел — половину ряда.

Вот результат этой пляжной математики, где стираемые числа подчеркнуты, а стертые обозначены звездочками. Здесь хватило всего двух просевов:

```
1-й отсев чисел, кратных 2:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
2-й отсев чисел, кратных 3:
1 2 3 * 5 * 7 * 9  * 11  * 13  * 15  * 17  * 19  *
Результат – простые числа:
1 2 3 * 5 * 7 * *  * 11  * 13  *  *  * 17  * 19  *
```

А если бы Эратосфен жил в наше время? Стал бы он царапать на песке? Конечно, нет, — на что ж тогда компьютеры? Программа P_38_4 находит все простые числа, не превышающие 255, — роль песка исполняет множество чисел.

```
program P_38_4;     { Решето Эратосфена }
var Simples : set of byte;  { множество чисел }
    n, m : integer;
    F : text;
begin
  Assign(F, 'P_38_4.out'); Rewrite(F);
  Simples:= [2..255];  { Сначала множество полное }
  { Цикл вычеркивания составных чисел }
  for n:=2 to (255 div 2) do begin
      { если число ещё не вычеркнуто }
      if n in Simples then
            { проверяем на кратность ему все последующие }
            for m:=2*n to 255 do
                  { если остаток(m/n) равен нулю, то m – составное }
                  if (m mod n)=0
                        { и его надо вычеркнуть из множества}
                        then Simples:= Simples – [m];
  end;
  { Распечатка множества простых чисел }
  for n:=2 to 255 do if n in Simples then Writeln(F,n);
  Close(F);  Readln;
end.
```

Мелочь, а приятно

Одну из первых своих программ мы снабдили разумом попугая, научив повторять имя пользователя. После ввода имени в переменную **S** программа печатала:

```
    Writeln ('Здравствуй, '+ S);
```

Сделаем её чуть умнее, научив отличать мальчиков от девочек. По крайней мере, для русских имен. У нас женские имена оканчиваются на буквы «а» или «я» (Анна, Светлана, Мария и так далее), чего не скажешь о мужских. Последнюю букву имени можно «выдернуть» в символьную переменную **C** таким оператором:

```
    C:= S[Length(S)];
```

И теперь приветствовать пользователя можно так:

```
    if (C='А') or (C='а') or (C='Я') or (C='я')
          then Writeln ('Здравствуй, девочка '+ S)
          else Writeln ('Здравствуй, мальчик '+ S);
```

Здесь проверяется совпадение переменной **C** с буквами верхнего и нижнего регистров, поскольку нельзя предсказать, в каком регистре будет введено имя. Условный оператор выглядит громоздко, но, призвав на помощь множество, мы упростим его:

```
    if C in ['А', 'а', 'Я', 'я']
          then Writeln ('Здравствуй, девочка '+ S)
          else Writeln ('Здравствуй, мальчик '+ S);
```

Переменную **C** проверяем на попадание в множество символов. Согласитесь, этот вариант читается приятней.

Итоги

Если вовремя смекнуть, что имеете дело с множествами, сложные задачи, как по волшебству, превратятся в простые!

А слабо?

А) Напишите программу для решения директорских задач и повторите подвиг контрразведчика. Или слабо?

Б) На острове действовали забавные законы по части транспортных средств — автобусов, грузовиков и легковушек. Во-первых, общее количество автомобилей на острове не должно было превышать 256. Автомобилям назначались номера с 0 до 255, при этом соблюдались следующие правила.

Номера, делившиеся без остатка на 7, назначались автобусам. Те, что делились без остатка на 5, назначались грузовикам, а все прочие — легковушкам. Например, номера 35 и 70 (они делятся и на 7, и на 5) доставались автобусам, а не грузовикам.

Схожие правила применялись и к окраске автомобилей, а именно: если номер авто делился на 4, его красили красным, если на 3 — желтым, если на 2 — белым, а остальные автомобили красили черным.

- Сформируйте три множества по классам автомобилей – автобусы, грузовики и легковушки. Вычислите количество машин каждого класса (Ответ: 37, 44, 175).
- Сформируйте четыре множества по цвету автомобилей – красные, желтые, белые и черные. Определите количество машин каждого цвета (Ответ: 64, 64, 43, 85).
- Столица того государства – деревня Кокосовка – страдала от пробок. Для их устранения ввели ограничение на въезд транспорта. Так, в один из дней недели в столицу пускали только красные легковушки, белые грузовики и все автобусы. Найдите номера всех этих машин. Сколько всего автомобилей могло въехать в столицу в тот день?

В) Полицейская база островного государства содержала номера угнанных автомобилей — числа от 1 до 255. Это был текстовый файл такого, например, вида:

```
120 31 16 25
```

То есть, номера перечислялись через пробел и следовали в произвольном порядке, что неудобно при поиске вручную. Ваша программа должна создать файл с номерами, упорядоченными по возрастанию. Подсказка: примените множество чисел.

Г) Генерация пароля длиной не менее восьми символов. В пароль входят символы трёх сортов: цифры, заглавные и строчные латинские буквы, например: «**7UpJ7rsT**», «**PasCal701**». Сделайте четыре варианта так, чтобы соблюдались следующие условия:

- символ каждого сорта входит в пароль не менее двух раз, некоторые символы могут повторяться;
- все символы пароля уникальны (примените множество);
- символы одного сорта не соседствуют, например: «**Pa7sCaL5**», уникальность символов не требуется;
- символы одного сорта не соседствуют и все символы уникальны.

Д) Напишите четыре булевы функции, проверяющие, является ли введенная пользователем строка правильно сформированным паролем согласно условиям предыдущей задачи.

Глава 39
Командная игра (массивы)

В чём сила компьютеров? В умении стремительно перемалывать огромные объемы данных: сотни, тысячи, миллионы элементов! Под элементами данных мы разумеем числа, строки и тому подобное. Обратимся и мы к этой способности компьютера. Нет, с миллионом элементов погодим, начнем всего с нескольких: рассмотрим, к примеру, турнирную таблицу чемпионата.

Снежная лавина

Вот задача для болельщика: отсортировать команды в турнирной таблице чемпионата по убыванию набранных ими очков. Команд немного, всего 16. После каждого тура количество очков меняется, и таблица сортируется заново. Корпеть над этим вручную? — это не для нас! Итак, будущая программа должна принимать с клавиатуры очки, набранные командами, и распечатывать команды в порядке убывания этих чисел. При этом набранные очки мы будем вводить всегда в одном и том же порядке.

Сделаем это сначала для двух команд, пусть ими будут «Динамо» и «Спартак». Сортировка двух команд — что может быть проще?

```
{ ввод и сортировка двух команд (в программе 14 строк) }
var T1, T2 : integer;
begin
      Readln (T1, T2);        { Ввод очков для «Динамо» и «Спартак» }
      if T1>T2
      then  begin
              Writeln('1.Динамо');
              Writeln('2.Спартак');
            end
      else  begin
              Writeln('1.Спартак');
              Writeln('2.Динамо');
            end;
      Readln;
end.
```

Здесь для каждой из команд отведена переменная, хранящая набранные очки: **T1** — для «Динамо» и **T2** — для «Спартака». Вариантов расстановки всего два, поэтому и программа очень проста — всего 14 строк, не считая комментария.

Теперь добавим в чемпионат команду «Зенит». Вариантов расстановки стало втрое больше — шесть, и программа заметно усложнилась, вот она.

```
{ сортировка трех команд (в этой программе 45 строк) }
var T1, T2, T3 : integer;
begin
  Readln (T1, T2, T3);        { «Динамо», «Спартак», «Зенит» }
  if (T1>T2) and (T1>T3)
    then begin
           Writeln('1.Динамо');
           if T2>T3
              then begin
                    Writeln('2.Спартак');
                    Writeln('3.Зенит');
                   end
              else begin
                    Writeln('2.Зенит');
                    Writeln('3.Спартак');
                   end
         end
    else begin
           if (T2>T1) and (T2>T3)
              then begin
                     Writeln('1.Спартак');
                     if T1>T3
                        then begin
                              Writeln('2.Динамо');
                              Writeln('3.Зенит');
                             end
                        else begin
                              Writeln('2.Зенит');
                              Writeln('3.Динамо');
                             end
                   end
              else begin
                     Writeln('1.Зенит');
                     if T1>T2
                        then begin
                              Writeln('2.Динамо');
                              Writeln('3.Спартак');
                             end
                        else begin
                              Writeln('2.Спартак');
                              Writeln('3.Динамо');
```

```
                                   end
                         end
           end;
  Readln;
end.
```

Здесь уже 45 строк, что втрое больше, чем для двух команд. С добавлением последующих команд программа продолжит разбухать, как снежный ком. Для четырех команд она станет длиннее ещё в 4 раза (180 строк), для пяти — ещё в 5 раз (900 строк) и так далее. Дойдя до шестнадцати команд, мы насчитаем в программе триллионы строк. А ведь триллион — это «всего лишь» миллион миллионов! Скорей свернем с этой гибельной тропы, пока снежная лавина не накрыла нас с головой!

А где же волшебная палочка?

Вы ощущаете причину трудностей? В моих решениях нет циклов, способных выполнять огромное количество однообразных действий. Так, например, одним оператором цикла печатается хоть тысяча, хоть миллион чисел. Увы! Применить цикл к переменным с именами **T1**, **T2** и **T3** не получится. Хотя цифры в этих именах означают для нас порядковые номера команд, для компилятора они — всего лишь часть имени переменной, и не более. Как же втолковать компилятору то, чего мы добиваемся нумерацией переменных?

Для этого есть особый тип данных — массив переменных или, проще — массив. Вот она, спасительная волшебная палочка!

Массивы вокруг нас

Массив объединяет несколько однотипных переменных под одним общим именем. Отдельные переменные в массиве называют его элементами, и доступ к ним возможен по их номерам. Массивы придумали отнюдь не программисты. Возьмите любую спортивную команду — футбольную или хоккейную. Здесь, кроме фамилии, игрок снабжен номером, который лучше виден на поле. И это не единственный пример массива. Если отдельную переменную уподобить ящику с хранящейся в нём информацией, то массив переменных будет комодом с пронумерованными ящиками (рис. 88).

Рис. 88 – Примеры простых переменных (слева) и массивов (справа)

Итак, массив — это собранные в одну команду переменные. Они получают общее на всех имя — имя своей команды. А внутри команды каждая переменная — элемент массива — обладает своим номером. Ну, чем не игроки?

Объявление массивов

Прежде, чем пользоваться массивом, его надо объявить: либо в секции **VAR,** либо через объявление пользовательского типа в секции **TYPE**.

Рассмотрим сначала первый способ, — объявим массив в секции **VAR**:

```
VAR   Имя_Массива : ARRAY [<MIN>..<MAX>] OF <Тип элемента>
```

Здесь использована пара ключевых слов **ARRAY**… **OF**…, что переводится как «массив… из…». Имя массива — это обычный идентификатор, его программист придумывает сам; будем считать это имя названием команды переменных.

Справа, после двоеточия, указывают две характеристики массива: 1) диапазон для индексов и 2) тип элементов массива. Рассмотрим эти атрибуты массива подробней.

Диапазон для индексов определяет допустимые номера элементов внутри массива. Диапазон указывают в квадратных скобках после слова **ARRAY**, — это два выражения порядкового типа, условно обозначенные мною как **MIN** и **MAX**, они

разделяются двумя точками. Говоря спортивным языком, здесь назначается диапазон номеров для «игроков команды».

После ключевого слова **OF** следует второй атрибут массива — тип данных для всех его элементов. Прибегнув вновь к спортивному языку, скажем, что здесь объявляют «вид спорта» для команды.

Вот пример объявления трех массивов: **Names** (фамилии), **Ratings** (оценки) и **ChampShip** (чемпионат):

```
VAR   { объявления переменных-массивов }

      { 30 строковых переменных с фамилиями учеников класса }
      Names : ARRAY [1..30] OF string;

      { 30 байтовых переменных с оценками учеников этого класса }
      Ratings : ARRAY [1..30] OF byte;

      { 16 чисел с очками, набранными командами в чемпионате }
      ChampShip : ARRAY [1..16] OF integer;
```

Как видите, массив можно составить из элементов любого типа. Так, массив **Names** содержит внутри себя 30 переменных строкового типа: **Names[1]**, **Names[2]** и так далее (номера переменных указывают в квадратных скобках).

Объявление массивов в секции **VAR** не слишком удобно. Почему? Рассмотрим следующий пример:

```
var   A : array [1..5] of integer;
      B : array [1..5] of integer;
begin
      A:= B; { здесь компилятор видит ошибку несовместимости типов}
end.
```

Мы объявили массивы **A** и **B**; на первый взгляд, это массивы одного типа, поскольку каждый из них содержит по пять целых чисел. Для однотипных переменных, включая массивы, Паскаль допускает операцию копирования. Например, оператором

```
      A:=B
```

все элементы массива **B** копируются в элементы массива **A**. Увы, компилятор увидит здесь ошибку несовместимости типов. В чем дело? А в том, что он считает разнотипными массивы, объявленные в разных операторах. Даже если массивы

совершенно одинаковы! Скажете, компилятор недостаточно умен? Может быть, но нам придётся как-то выкручиваться, и для этого есть два пути.

Во-первых, переменные **A** и **B** можно объявить в одном операторе:

```
var   A, B : array [1..5] of integer;
```

Это устраняет проблему несовместимости типов.

Но есть и лучший способ — сначала объявить для массива пользовательский тип данных. Это делается в секции **TYPE** так.

```
TYPE  Имя_Типа = ARRAY [<MIN>..<MAX>] OF <Тип элемента>
```

В сравнении с объявлением переменной разница мизерная: вместо двоеточия видим знак равенства, а вместо имени переменной — имя типа. Но каковы последствия! Объявите лишь однажды нужный вам тип, и тогда применяйте его, где угодно. Вот объявления типов для указанных выше переменных:

```
TYPE  { примеры объявления типов-массивов }
      { тип для 30 строковых переменных с фамилиями учеников класса }
      TNames = ARRAY [1..30] OF string;

      { тип для 30 байтовых переменных с оценками учеников }
      TRatings = ARRAY [1..30] OF byte;

      { тип для 16 целых переменных с очками, набранными в чемпионате }
      TChampionShip = ARRAY [1..16] OF integer;
```

Здесь буква «Т» в имени типа напоминает о назначении этого идентификатора (помните наше добровольное соглашение об именах?). Теперь учрежденные типы данных можно употребить для объявления переменных и параметров в любом месте программы, вот пример:

```
TYPE  { тип для 30 байтовых переменных с оценками учеников }
      TRatings = ARRAY [1..30] OF byte;

VAR   { 30 байтовых переменных с оценками учеников }
      Ratings : TRatings;
```

```
procedure ABC (var arg: TRatings);  { параметр процедуры }
var   A, B, C : TRatings;           { локальные переменные }
begin
      . . .
end;
```

Здесь тип **TRatings** служит для объявления переменных и параметров в трех местах программы. В будущем мы всегда будем объявлять типы — как для массивов, так и для других сложных наборов данных.

Доступ к элементам (индексация)

Переменной-массивом можно ворочать как единым целым, например, при копировании одного массива в другой. Но чаще приходится работать с отдельными его элементами, как «выдернуть» их из массива?

Очень просто: воспользуйтесь индексацией, — она знакома вам по работе со строками. Как и для доступа к отдельному символу строки, для доступа к элементу массива надо указать его индекс, то есть порядковый номер в массиве. Индекс указывают в квадратных скобках, стоящих после имени массива, он представляет собой выражение порядкового типа. Кстати, сходство со строками не случайно, ведь строка — это особый род массива, составленного из отдельных символов.

Рассмотрим примеры доступа к элементам объявленных выше массивов.

Пример 1. Трем элементам массива **Names** даём фамилии хоккеистов:

```
      Names[1]:= 'Петров';
      Names[2]:= 'Михайлов';
      Names[3]:= 'Харламов';
```

Пример 2. Сравниваем третий и четвертый элементы массива **Ratings**. Здесь индексы заданы через целочисленную переменную **n**.

```
      ...
      Ratings[3]:= 12;
      Ratings[4]:= 8;
      n:=3;
      if Ratings[n] > Ratings [n+1] then ... else ...;
```

Как видите, индекс в массиве можно вычислять, а это открывает дорогу к циклам. И мы двинемся ею немедленно!

Ввод и вывод массивов

Ввод и вывод — это те задачи, не решив которые, не стоит помышлять о применении массивов. Ни то, ни другое не сделать одним махом. Здесь, как и для множеств, нужны циклы, обрабатывающие отдельные элементы массива.

Взять, к примеру, массив **Names**, ввести который можно так:

```
for i:=1 to 30 do Readln(F, Names[i]);
```

Здесь **F** — это открытый для чтения текстовый файл, каждая строка которого содержит фамилию.

На первый взгляд всё просто. Просто, да не гладко, — это будет работать лишь с файлом, в котором не менее 30 строк (по числу циклов). А иначе случится ошибка: противозаконное чтение за пределами файла. Как избежать её? Внимательней присматривайте за концом файла, вот так:

```
i:=1;
{ пока не конец файла и не введены все элементы }
while not Eof(F) and (i<=30) do begin
      Readln(F, Names[i]);
      i:= i+1;
end;
```

А вот ещё один хороший вариант:

```
for i:=1 to 30 do begin
      if Eof(F) then break;   { если конец файла, прервать цикл }
      Readln(F, Names[i]);
end;
```

Вывод массива в файл не представляет труда, вот пример:

```
for i:=1 to 30 do Writeln(F, Names[i]);
```

Разумеется, что файловая переменная **F** должна быть открыта для записи.

Ошибки индексации

Объявление массива, как сказано, содержит границы для индексов: **MIN** — номер первого элемента, и **MAX** — номер последнего. А что случится при попытке обратиться к элементу с меньшим, чем **MIN** номером? Или наоборот — с большим, чем **MAX**? Иначе говоря, что случится при попытке доступа к несуществующему

элементу массива? Такие ошибки преследуют даже опытных программистов, а последствия зависят от способа, которым вы совершите сей проступок.

Предположим, в программу вкрался такой оператор:

```
Names[200]:= 'Синичкин';
```

Поскольку в массиве **Names** нет элемента с индексом 200, здесь вас остановит компилятор, — ошибка слишком явна, чтобы он промолчал. Вам не останется ничего иного, как исправить индекс, иначе программа не откомпилируется.

Но, когда индекс вычисляется при исполнении программы, нарушение границ проявляется и обрабатывается иначе, например:

```
Readln(N);
Writeln(Names[N]);
```

Нам не угадать, что введет пользователь в переменную **N**, — здесь ошибка нарушения границ может возникнуть при выполнении программы. В главе 27 мы рассматривали ошибки времени исполнения, — это как раз такой случай. Если указать индекс, выходящий за границы массива, то реакция программы будет зависеть от настройки компилятора, точнее, от опции контроля диапазонов. Напомню, что эта опция управляется директивой **$R**, а также доступна через меню по цепочке:

Options → Compiler… → Runtime Errors → Range checking

Рассмотрим вариант компиляции при включенном контроле границ (**$R+**). Тогда, при нарушении границ индекса, программа выдаст аварийное сообщение «Range check error». То есть, она заметила нарушение границ индекса, «крикнула» об этом и прервала работу.

Теперь отключим контроль диапазонов (**$R-**) и перекомпилируем программу. Она станет «легче» и быстрее, и по ходу выполнения проверять границы не станет. Но ошибки не пройдут бесследно. Наоборот, последствия будут тяжелыми и непредсказуемыми! Отключать проверку диапазонов позволительно только в тщательно проверенной программе.

Лучший способ избежать нарушения границ индексов — взять проверку на себя. В данном случае это можно сделать так:

```
    repeat
          Readln(N);
          if N in [1..30]
                then Writeln(Names[N])
                else Writeln('Ошибка! Введите индекс от 1 до 30');
    until N in [1..30]
```

Этот цикл будет терзать пользователя, пока тот не введет допустимое значение индекса, или не выключит компьютер.

Итоги

- Массив – это сложный тип данных, объединяющий в себе несколько однотипных переменных – элементов массива.
- Все элементы массива носят одно общее имя – это имя самого массива. Внутри массива элементы различаются своими порядковыми номерами – индексами.
- В объявлении массива указывают две его характеристики: диапазон индексов и тип элементов.
- Индекс элемента может быть задан числом или выражением порядкового типа.
- Указание неверного индекса порождает ошибки либо при компиляции, либо при выполнении программы.
- Ввод массива из текстового файла и вывод в него возможен только поэлементно, для чего организуют цикл.

А слабо?

А) Массив **A** и переменная **C** объявлены так:

```
var   A : array ['a'..'z'] of integer;
      C: char;
```

Допустимо ли такое объявление массива и почему? Сколько элементов содержит массив? Какие из указанных ниже операторов будут (или могут) вызывать ошибки нарушения диапазонов?

```
    A['s']:= 10;
    A['R']:= 10;
    C:='d';     A[C]:= 10;
    Readln(C);  A[C]:= 10;
```

Проверьте свои решения на практике.

Глава 40
Пристрелка на знакомых мишенях

Итак, из арсенала Паскаля мы извлекли ещё одно мощное оружие — массивы. Опробуем его на знакомых мишенях, — некоторые наши программы можно улучшить, например, программу «вопрос-ответ» или полицейскую базу данных.

Вопрос-ответ – добиваемся гибкости

В 16-й главе мы смастерили шуточную программку, невпопад отвечающую на вопросы пользователей. Жаль только, что ответы намертво вбиты в саму программу. Скоро пользователям надоест смеяться над одним и тем же, и они забросят игрушку. Так пусть ваши приятели сами сочиняют смешные ответы и помещают их в текстовый файл, и тогда программа при запуске будет загружать их оттуда.

Прежде всего, подумаем над размещением вводимых из файла строк, где поселить их? «В массиве строк», — скажете, и будете правы. А сколько элементов запасти в этом массиве? Чем больше, тем лучше? Некоторые компиляторы накладывают ограничение на размер массива, но сотню строк они позволят, и этого пока достаточно. Итак, для хранения ответов объявим массив из 100 строковых переменных.

Перейдем к процедуре ввода этих строк. Техника ввода массива рассмотрена в предыдущей главе. Но теперь надо ещё и подсчитать введенные строки, иначе в дальнейшем мы не всегда сможем правильно индексировать массив, — ведь фактическое количество строк в файле может быть и меньше ста. С этой целью объявим переменную Fact, в которой и сделаем нужный нам подсчёт.

Обсудив эти моменты, обратимся к программе P_40_1.

```
{ P_40_1 - Программа "вопрос-ответ" с применением массива }

const CAnswers = 100;   { размер массива с ответами }
      { объявление типа для массива ответов }
type  TAnswers = array[1..CAnswers] of string;

var   Answers : TAnswers;     { объявление массива ответов }
      Fact : integer;         { фактическое количество ответов }
      F : text;               { файл с ответами }
      S : string;             { строка с вопросом }
```

```
  { Процедура ввода ответов из файла с подсчетом введенных строк }
procedure ReadFromFile(var aFile: text);
var i: integer;
begin
  Fact:=0;  { для начала подсчета строк обнуляем счетчик }
  { цикл по массиву строк }
  for i:=1 to CAnswers do begin
      if Eof(aFile) then Break;     { если конец файла - выход}
      Readln(aFile, Answers[i]);    { читаем строку в элемент массива }
      Fact:= Fact+1;                { наращиваем счетчик строк }
  end;
end;

begin    {--- Главная программа ---}
      Assign(F, 'P_40_1.in');  Reset(F);
      ReadFromFile(F);  Close(F);
      Randomize;  { чтобы порядок вопросов не повторялся }
      { Начало главного цикла }
      repeat
            Write('Введите вопрос: ');  Readln(S);
            if S<>'' then     Writeln(Answers[Random(Fact)+1]);
      until S='';
end.
```

Открыв файл «P_40_1.IN», мы вызываем процедуру **ReadFromFile** (читать из файла), которая загружает строки в массив **Answers** (ответы). Она же подсчитывает введенные строки в переменной **Fact**. Таким образом, если файл содержит больше сотни строк, то в массив попадёт первая сотня, а иначе — столько, сколько там есть фактически, и это количество покажет переменная **Fact**. Дальше всё работает, как в прежнем варианте: после ввода вопроса ответ случайным образом выбирается из массива. Индекс элемента с ответом определяется выражением **Random(Fact)+1**. Если помните, функция **Random(Fact)** возвращает значения в диапазоне от **0** до **Fact-1**, а индексы нашего массива начинаются с единицы.

Полицейская база данных – ускоряем поиск

А теперь освежите в памяти другое наше творение — программу поиска угнанных автомобилей в полицейской базе данных (глава 29). Её слабость в том, что поиск номеров выполняется в текстовом файле. Ах, если б вы знали, как «тормозит» такой поиск! Вы не заметили? Да, на десятках строк этого не ощутить, иное дело — сотни тысяч, или миллионы. Итак, перенесем список номеров из текстового файла в массив, и тогда поиск ускорится многократно!

В программе P_40_2 обратите внимание на пропуск пустых строк в процедуре **ReadFromFile**. Если этого не сделать, счётчик **Fact** может оказаться на 1 больше, чем должно, — так случится, если за последним числом будут пустые строки. Следующий далее оператор чтения числа пренебрегает границами между строками, поэтому в одной строке допустимы несколько чисел.

```
{ P_40_2 - Полицейская база данных с применением массива }

const CNumbers = 1000;  { размер массива с номерами автомобилей }
      { объявление типа для массива номеров }
type  TNumbers = array[1..CNumbers] of integer;
var   Numbers : TNumbers;     { объявление массива номеров }
      Fact : integer;         { фактическое количество номеров в файле }
      F : text;               { файл с номерами }
      Num : integer;          { номер проверяемого автомобиля }

      { Процедура ввода номеров из файла }
procedure ReadFromFile(var aFile: text);
var i: integer;
begin
  Fact:=0;  { для начала подсчета номеров обнуляем счетчик }
  for i:=1 to CNumbers do begin { цикл по массиву номеров }
    while Eoln(aFile) do     { Пропуск пустых строк }
      if Eof(aFile) then Break else Readln(aFile);
    if Eof(aFile) then Break;   { если конец файла - выход из цикла }
    Read(aFile, Numbers[i]);    { читаем номер в элемент массива }
    Fact:= Fact+1;              { наращиваем счетчик номеров }
  end;
end;

  { Функция поиска в массиве номеров автомобилей }
function FindNumber(aNum: integer): boolean;
var i: integer;
begin
      FindNumber:= false;
      for i:=1 to Fact do
            if aNum=Numbers[i] then begin
                  FindNumber:= true;      { нашли ! }
                  Break;                  { выход из цикла }
            end
end;
```

```
begin      {--- Главная программа ---}
      { открываем файл и читаем номера автомобилей }
      Assign(F, 'P_38_2.in');  Reset(F);
      ReadFromFile(F);        { ввод номеров из файла }
      Close(F);
      repeat      { Главный цикл }
            Write('Укажите номер автомобиля: ');     Readln(Num);
            if FindNumber(Num)
                  then Writeln('Эта машина в розыске, хватайте его!')
                  else Writeln('Пропустите его');
      until Num=0;  { 0 – признак завершения программы}
end.
```

Ещё раз о статистике

Следующая программка будет маленькой, да удаленькой. Вернемся к статистике, с которой познакомились при обработке классного журнала. Напомню, что статистика — это наука, изучающая массовые явления. В текстах наших программ полным-полно разных букв, — давайте посчитаем их. Результатом работы программы будет таблица, похожая на эту:

```
a     119
b      45
c      72
. . .
```

Здесь левый столбец составляют буквы, а правый — количество этих букв в некотором файле. Упростим себе задачу, ограничившись подсчетом лишь маленьких латинских букв от «a» до «z».

Для подсчета общего количества символов в файле хватило бы одного счетчика. Но здесь 26 букв, а значит и счетчиков надо столько же. Массив счетчиков напрашивается сам собой, его тип можно объявить так:

```
type TCounts = array [1..26] of integer;
```

Однако не спешите этого делать. Вспомните о том, что индексом массива может быть любой порядковый тип данных. А к ним, наряду с числами, относятся символьный и даже булев тип. Стало быть, допустимы такие массивы:

```
type  TA = array ['A'..'F'] of integer;
      TB = array [false..true] of integer;
```

Первый из них содержит 6 элементов, а индексируется символьным выражением. Второй содержит всего два элемента, индексы которого имеют булев тип. В решаемой задаче напрашивается символьная индексация, а потому объявим тип для массива счетчиков так:

```
type TCounts = array ['a'..'z'] of integer;
```

Теперь символ, прочитанный из файла, можно использовать как индекс в массиве счетчиков, надо лишь предварительно проверить его на попадание в нужный диапазон.

Входным файлом программы будет текст её самой же. Вот она, простая и красивая:

```
{ P_40_3 - Подсчет количества различных букв в файле }
      { Тип массива из целых чисел, индекс - символьный }
type TCounts = array ['a'..'z'] of integer;

var   Counts : TCounts; { массив из счетчиков букв }
      c: char;    { текущий символ файла, он же - индекс счетчика }
      F : text;   { файл с текстом программы }

begin       {--- главная программа ---}
  { Перед началом подсчета все счетчики обнуляем }
  for c:='a' to 'z' do Counts[c]:=0;
  { Открываем входной файл для чтения }
  Assign(F, 'P_40_3.pas');  Reset(F);
  while not Eof(F) do begin { Цикл чтения и подсчета букв }
    Read(F, c);             { чтение одного символа из файла }
    if c in ['a'..'z']      { если символ в нужном диапазоне }
      then Counts[c]:= Counts[c]+1; { наращиваем его счетчик }
  end;
  Close(F);
  { После подсчета распечатаем все счетчики }
  for c:='a' to 'z' do Writeln (c, Counts[c]:6);
  Write('Нажмите Enter'); Readln;
end.
```

Здесь осталась лишь одна шероховатость — при печати результатов часть строк не поместится на экране. Так направьте вывод в текстовый файл. Или слабо?

Итоги

- Массивы, как любые переменные, «живут» в оперативной памяти. Переместив данные из файлов в массивы, мы многократно ускорим их обработку.
- Для индексации массивов допустимы любые порядковые типы данных. Выбор подходящего типа для индекса упрощает и украшает программу.
- При чтении чисел из текстового файла в «боевых» программах необходимо учитывать возможное наличие в файле пустых строк. Такие строки могут привести к чтению оператором **Read** несуществующего пустого числа (см. процедуру **ReadFromFile** в программе P_40_2).

А слабо?

А) Напишите программу для подсчета различных цифр в файле полицейской базы данных (считать надо именно цифры, а не числа!).

Б) Объявите массив из сотни целых чисел, заполните его случайными числами в диапазоне от 0 до 255 и распечатайте этот массив.

В) Найдите в массиве (задание Б) все элементы, хранящие число 7 (если таковые найдутся). Напечатайте индексы элементов, которые содержат это число.

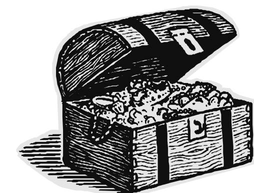
Г) Заполните массив (задание Б) случайными числами в диапазоне от 0 до 255 так, чтобы ни одно из них не повторялось. Воспользуйтесь вспомогательным множеством чисел, где будут запоминаться сгенерированные ранее числа.

Д) Найдите в массиве (задание Г) наименьшее и наибольшее числа, напечатайте их, а также соответствующие им индексы элементов массива.

Е) Вращение массива вправо. Объявите массив из 10 чисел и заполните его случайным образом. Напишите процедуру, перемещающую 1-й элемент на 2-е место, 2-й — на 3-е место и т.д. Последний элемент должен занять 1-е место.

Ж) Вращение массива влево. Напишите процедуру для перемещения 2-го элемента на 1-е место, 3-го — на 2-е место и т.д. При этом первый элемент должен стать последним.

И) Напишите функцию для подсчета количества номеров в полицейской БД при условии, что одна строка может содержать несколько номеров, а некоторые строки (в т.ч. в конце файла) могут быть пустыми.



Глава 41
По порядку, становись!

В 39-й главе, где состоялось наше знакомство с массивами, мы намерились отсортировать футбольные команды в порядке набранных ими очков. Следуя к этой цели, перенесемся ненадолго в прошлое — лет на триста назад.

Пиратская справедливость

Тогда в морях разбойничали «джентльмены», которых мы зовем пиратами. Одной из таких бригад повелевал некто Райт. Пиратские команды не отличались дисциплиной, но Райт добился порядка на корабле, избегая жестокостей. Отважный в бою, Райт давал пример и в мирных обстоятельствах, деля добычу если не поровну, то хотя бы по справедливости. Вожак брал равную со всеми долю, потому команда чтила его и подчинялась беспрекословно.

Однажды джентльменам удачи достался сундук с золотыми слитками. Пересчитав их, пираты выяснили, что каждому из них полагается ровно по два. Казалось бы, о чем тут думать? — садись и дели. Однако слитки существенно отличались формой и размерами. Пираты не могли распилить или переплавить их, сделав одинаковыми. Как быть? — с таким вопросом Райт обратился к экипажу.

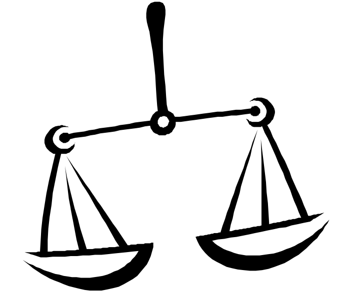
После недолгих споров разбойники уже готовы были бросить жребий (на случайную дележку никто не обижался). Но тут голос подал бывший аптекарь, а ныне корабельный лекарь по кличке Нашатырь.

— Я предлагаю, — молвил Нашатырь, — разложить слитки в порядке их веса. Затем кто-то из нас возьмет самый легкий и самый тяжелый из них. Другой — самый легкий и самый тяжелый из оставшихся, и так далее.

Свою мысль он сопроводил рисунком с шестью слитками разного веса и размера (рис. 89).

— Недурно, — согласился Райт, — но как взвесить слитки? У нас нет ни гирь, ни весов.

— Зачем мне гири? Для сравнения слитков годится вот это, — и лекарь достал из своего сундука самодельные чашечные весы.



— Ну что ж, — сказал Райт, — ты придумал, тебе и делить.

И под пристальными взглядами всей команды лекарь принялся за дело.

Вначале ряд слитков он выложил, как попало. Затем стал по очереди сравнивать соседние куски. Если первый из них был тяжелее второго, аптекарь

менял их местами. Так он сравнил первый и второй слитки, второй и третий, третий и четвертый и так далее до последнего слитка. В конце концов, самый тяжелый слиток оказался на последнем месте. Затем он повторил всё это ещё раз, и тогда второй по величине слиток оказался на предпоследнем месте. Проделав это **N-1** раз, где **N** — количество слитков, лекарь выложил слитки в нужном порядке: первым лежал самый легкий, а последним — самый тяжелый.

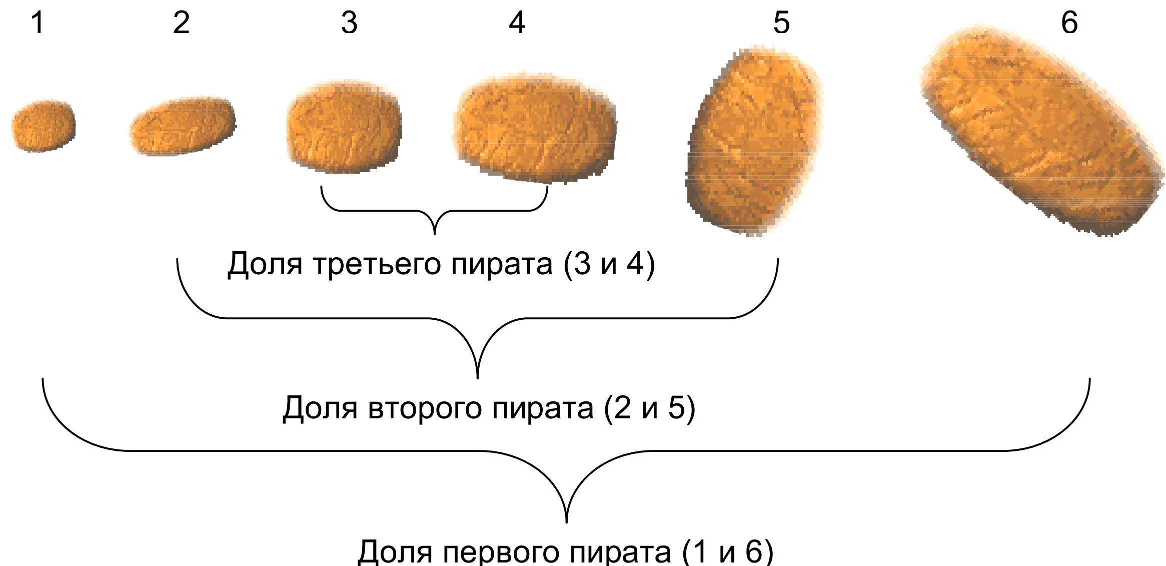


Рис. 89 – Пример справедливого дележа шести слитков между тремя пиратами

«А теперь бросайте жребий, — подытожил Нашатырь, скромно отходя в сторону, — пусть он определит порядок взятия долей». Пираты остались довольны.

Пузырьковая сортировка

Вернемся в наше время. О пиратской истории программист сказал бы так: разложив куски золота в нужном порядке, лекарь отсортировал массив слитков. Его метод известен как «пузырьковая сортировка» — Bubble Sort. Откуда взялось это название? Проследите за пузырьками в стакане газировки: по мере всплытия они объединяются с другими и становятся крупнее.

Воспользуемся методом лекаря-аптекаря для сортировки массива из 10 целых чисел — это будут наши золотые слитки. А для испытания алгоритма напишем программу P_41_1, которая вначале заполнит массив случайным образом и распечатает его. Затем отсортирует этот массив и снова распечатает. Работу по сортировке выделим в отдельную процедуру по имени **BubbleSort**, — она ещё пригодится нам в последующих проектах.

Исследуйте процедуру сортировки по шагам. Здесь «крутятся» два цикла **FOR-TO-DO**, вложенные друг в друга. Внутренний цикл со счетчиком **J**

сравнивает соседние числа и, при необходимости, меняет их местами. Переменная **T** нужна для перестановки соседних элементов. По завершении одного внутреннего цикла очередное крупное число оказывается на своем месте. Но, поскольку сортируются **CSize** чисел, то внутренний цикл надо повторить **CSize-1** раз, — это делает внешний цикл со счетчиком **I**.

```
      { P_41_1 – Сортировка массива целых чисел }
const CSize = 10; { размер массива }

      { объявление типа для массива }
type  TGolds = array [1..CSize] of integer;

var  Golds : TGolds;  { массив кусков золота }
      { Процедура "пузырьковой" сортировки }
      { Внимание! Параметр-массив передается по ссылке! }
procedure BubbleSort (var arg: TGolds);
var  i, j, t: Integer;
begin
for i:= 1 to CSize-1 do    { внешний цикл }
      for j:= 1 to CSize-1 do  { внутренний цикл }
      { если текущий элемент больше следующего ...}
      if arg[j] > arg[j+1] then begin
            { то меняем местами соседние элементы }
            t:= arg[j];       { временно запоминаем }
            arg[j]:= arg[j+1];      { следующий -> в текущий }
            arg[j+1]:= t;           { текущий -> в следующий }
      end;
end;
```

Обратите внимание: сортируемый массив передан в процедуру по ссылке **VAR**. Передача в процедуры массивов, множеств, строк и других сложных типов данных по ссылкам **CONST** и **VAR** — обычная практика. Это повышает скорость работы программ и уменьшает объём памяти, занимаемый параматрами.

```
var i:integer; { для индекса в главной программе }

begin {--- Главная программа ---}
      { заполняем массив случайным образом }
      Randomize;
      for i:=1 to CSize do Golds [i]:= 1+Random(1000);
      { распечатаем до сортировки }
      Writeln('До сортировки:');
      for i:=1 to CSize do Writeln(Golds [i]:3);
      { сортируем }
      BubbleSort(Golds);
      { распечатаем после сортировки }
      Writeln('После сортировки:');
      for i:=1 to CSize do Writeln(Golds [i]:3);
      Readln;
end.
```

При должном внимании вы обнаружите в этой сортировке небольшой изъян, суть которого такова. После отработки первого внутреннего цикла самый большой элемент окажется на последнем месте. А значит, на втором внутреннем цикле нет смысла сравнивать два последних элемента. На третьем проходе соответственно нет смысла сравнивать три последних элемента, — они уже лежат в нужном порядке. На этих сравнениях мы зря теряем время. Порок этот легко устранить, если поправить внутренний цикл так:

```
      for j:= 1 to CSize - i do  { внутренний цикл }
```

Теперь каждый следующий внутренний цикл будет на единицу короче предыдущего (ведь счетчик внешнего цикла **`I`** растет). В следующей программе мы так и сделаем.

Электронная делёжка

Рассмотрев хитрости пузырьковой сортировки, поможем теперь морским романтикам. Напишем программу для справедливой дележки золотых слитков. Основная работа уже проделана, — мы смогли отсортировать массив. Осталось лишь распечатать веса тех кусков, что достанутся каждому из пиратов. Известно, что первому пирату достанется первый и последний слитки, второму — второй и предпоследний и так далее. Иначе говоря, **`I`**-му пирату достанутся слитки с номерами **`I`** и **`CSize+1-I`**. Программа `P_41_2` «делит слитки», распечатывая после сортировки веса соответствующих пар.

```
{ P_41_2 – Пиратская делёжка по справедливости }

const CSize = 16; { размер массива слитков }
      { объявление типа для массива слитков }
type  TGolds = array [1..CSize] of integer;
var  Golds : TGolds;  { массив кусков золота }
      { Процедура "пузырьковой" сортировки }
procedure BubbleSort (var arg: TGolds);
var  i, j, t: Integer;
begin
for i:= 1 to CSize-1 do    { внешний цикл }
      for j:= 1 to CSize-i do  { внутренний цикл }
      { если текущий элемент больше следующего ...}
      if arg[j] > arg[j+1] then begin
            { то меняем местами соседние элементы }
            t:= arg[j];       { временно запоминаем }
            arg[j]:= arg[j+1];      { следующий -> в текущий }
            arg[j+1]:= t;           { текущий -> в следующий }
      end;
end;

var i:integer; { используется в качестве индекса в главной программе }
begin
      { заполняем массив случайным образом }
      Randomize;
      for i:=1 to CSize do Golds[i]:= 500 + Random(500);
      { сортируем }
      BubbleSort(Golds);
      Writeln('По справедливости:');
      for i:=1 to (CSize div 2) do begin
            { два куска по отдельности }
            Write(i:2, Golds[i]:5,' + ',Golds[CSize+1-i]:3,' = ');
            { сумма двух кусков }
            Writeln(Golds[i]+Golds[CSize+1-i] :4);
      end;
      Readln;
end.
```

Вот результат одной из таких делёжек:

```
По справедливости:
 1    506 + 975 = 1481
 2    556 + 967 = 1523
 3    587 + 954 = 1541
 4    629 + 916 = 1545
 5    691 + 876 = 1567
 6    694 + 872 = 1566
 7    749 + 845 = 1594
 8    751 + 800 = 1551
```

Здесь самый легкий и самый тяжелый слитки отличаются почти вдвое: 506 и 975 граммов. Но пары слитков, доставшихся пиратам, отличаются по весу незначительно.

Возвращение на футбольное поле

Закаленные морскими приключениями, вернемся к сортировке футбольных клубов (задача поставлена в главе 39, помните?). Что мы будем сортировать? Набранные очки? Да, но их надо как-то привязать к названиям команд.

Поступим так. Объявим два массива: один (числовой) — для набранных очков, другой (строковый) — для названий клубов. При вводе данных элементы двух массивов будут соответствовать друг другу, поскольку имена команд и набранные ими очки вводятся одновременно. Затем, в ходе сортировки, переставляя элементы с очками, будем менять местами и соответствующие им элементы с названиями команд. Так имена команд последуют за очками, заработанными командами. Всё это потребует небольших переделок в процедуре сортировки.

Впрочем, потребуется ещё одно мелкое изменение. Если при сортировке золотых слитков мы добивались возрастающего порядка, то теперь нужен противоположный, убывающий порядок сортировки. Как его добиться? Очень просто: изменим условие сравнения соседних элементов на противоположное. Вот собственно и всё, осталось лишь показать программу P_41_3.

```
{P_41_3 - Футбольный чемпионат }
const CSize = 16; { количество команд }
      { объявление типов для массивов }
type  TAces = array [1..CSize] of integer;  { тип для очков }
      TNames = array [1..CSize] of string;  { тип для названий }
var   Aces : TAces;  { набранные очки }
      Names: TNames; { названия команд }
      { Процедура "пузырьковой" сортировки очков с именами команд }
procedure BubbleSort2(var arg1: TAces; var arg2: TNames);
var   i, j, t: Integer;
      s: string;
begin
  for i:= 1 to CSize-1 do    { внешний цикл }
      for j:= 1 to CSize-i do  { внутренний цикл }
            { если текущий элемент меньше следующего ...}
            if arg1[j] < arg1[j+1] then begin
                  { то меняем местами соседние элементы }
                  t:= arg1[j];            { временно запоминаем }
                  arg1[j]:= arg1[j+1];    { следующий -> в текущий }
                  arg1[j+1]:= t;          { текущий -> в следующий }
                  { меняем местами и названия команд }
                  s:= arg2[j];            { временно запоминаем }
                  arg2[j]:= arg2[j+1];    { следующий -> в текущий }
                  arg2[j+1]:= s;          { текущий -> в следующий }
            end;
end;
var i: integer;
begin       { главная программа }
  { Вводим названия команд и набранные очки }
  for i:=1 to CSize do begin
      Write('Название команды: '); Readln(Names[i]);
      Write('Набранные очки:   '); Readln(Aces[i]);
  end;
  BubbleSort2(Aces, Names);  { сортируем }
  Writeln('Итоги чемпионата:');
  Writeln('Место      Команда            Очки');
  for i:=1 to CSize do
      Writeln(i:3,' ':3, Names[i], Aces[i]:20-Length(Names[i]));
  Readln;
end.
```

Спецификатор ширины поля в операторе печати задан выражением

```
      20 - Length(Names[i])
```

Здесь перед колонкой с очками будет тем больше пробелов, чем короче название команды, — так выравниваются колонки таблицы.

Для проверки программы я ввел наобум имена четырех команд нашего чемпионата и очки, якобы заработанные ими (количество команд **CSize** установил равным 4), и вот что у меня вышло:

```
Итоги чемпионата:
Место     Команда       Очки
  1   Локомотив          55
  2   Крылья Советов     54
  3   Спартак            47
  4   Зенит              43
```

Болельщики вправе оспорить результат, но я им доволен.

Итоги

- Расположение данных в порядке возрастания или убывания называется сортировкой.
- Простейший алгоритм сортировки массива – «пузырьковая» сортировка. Она состоит в сравнении и перестановке соседних элементов массива, при этом организуются два вложенных цикла.

А слабо?

А) Напишите программу для сортировки фамилий учеников в алфавитном порядке (фамилии берутся из файла). Программа должна сортировать их как по возрастанию, так и по убыванию фамилий, — на выбор пользователя.

Б) Придумайте самый несправедливый способ пиратской дележки по два слитка и напишите программу для неё.

В) Напишите программу для дележки случайным образом (как это собирались сделать пираты). Насколько отличаются ваши результаты от справедливого способа?

Г) Напишите функцию, проверяющую, упорядочен ли числовой массив. Функция должна вернуть **TRUE**, если массив упорядочен по возрастанию. Массив внутрь функции передайте параметром по ссылке.

Глава 42
Кто ищет, тот всегда найдет

Все кругом ищут что-то: Карабас-Барабас — золотой ключик, Лиса с Котом — дураков, а Буратино — Страну Дураков. И я ищу: то ключи в карманах, то тапочки под диваном. А сколько всего таится в Интернете! Благо, искать информацию там помогают компьютеры.

Где эта улица, где этот дом?

В 40-й главе мы смастерили программу для поиска угнанных автомобилей. Испытаем её вон на той легковушке. Вводим номер машины и… оп! Вот так удача! Автомобильчик-то в розыске, — надо вернуть его владельцу. Однако, кто он? Где живет? А телефончик не подскажете? К сожалению, в нашей базе данных этих сведений нет, — её следует дополнить.

Добавим в программу поиска автомобилей массив строк, где будем хранить сведения о владельце: его имя, фамилию и телефон.

```
const CNumbers = 100;   { размер массивов }
type  TNumbers = array[1..CNumbers] of integer;
      TNames = array[1..CNumbers] of string;
var   Numbers : TNumbers;     { массив номеров автомобилей }
      Names : TNames;         { массив сведений о владельце }
```

Здесь добавлен массив **Names** (имена), содержащий столько же строк, сколько номеров в базе данных. Эти строки соответствуют элементам массива номеров **Numbers**. Так, если элемент **Numbers[7]** содержит число 123, а элемент **Names[7]** — строку «Горбунков С.С., тел. 11-22-33», то значит, гражданин Горбунков владеет автомобилем с номером 123.

Что связывает массивы **Names** и **Numbers**? Ответ очевиден — общий индекс. Определив индекс автомобиля в массиве номеров, мы получим доступ и к сведениям о его владельце в строковом массиве.

Последовательный поиск

Напомню, что в полицейской базе данных из 40-й главы заголовок функции поиска был таким:

```
function FindNumber(aNum: integer): boolean;
```

Функция **FindNumber** выясняет, существует ли искомый номер в массиве **Numbers**, то есть она дает булев результат. Теперь этого мало, — хочется получить индекс элемента, где хранится искомый номер. А если такого номера в

массиве нет? Пусть тогда функция вернет некоторое условное значение, например, минус единицу.

С учетом этих пожеланий, напишем новую функцию поиска и дадим ей имя **FindSeq** (от слов FIND — «искать», SEQUENCE — «последовательно»).

```
      { Функция поиска в массиве Numbers позиции числа aNum }
function FindSeq (aNum: integer): integer;
var i: integer;
begin
      FindSeq:= -1;           { если не найдем, то результат будет -1 }
      for i:=1 to Fact do
            if aNum=Numbers[i] then begin
                  FindSeq:= i;      { нашли, возвращаем индекс }
                  Break;            { выход из цикла }
            end
end;
```

Новая функция сравнивает искомое число с элементами массива, перебирая их последовательно до тех пор, пока не найдет подходящий элемент или не уткнется в конец массива. В случае успеха она вернет индекс элемента в массиве, а иначе — минус единицу.

Этот способ называют поиском прямым перебором или линейным поиском. Линейный поиск прост, но крайне медлителен. Если бы библиотекарь искал заказанную книгу прямым перебором, клиент дремал бы в ожидании заказа месяцами! Но библиотекарь справляется с поиском, живо находя нужное среди сотен тысяч томов. Как ему удается это?

Всё дело в порядке. Там, где порядок, искать проще и быстрей. Вы ищите ложку в кухонном шкафу, а ботинки — на обувной полке, но не обшариваете весь дом. Есть свой порядок и в библиотеке, потому персонал и справляется с работой. Компьютер ищет куда быстрее человека, и всё же понуждать его к линейному поиску — проявление крайней жестокости. Впрочем, пострадает не столько компьютер, сколько уснувший в томлении пользователь.

Двоичный поиск

Один удачливый зверолов в минуту откровенности поделился секретом своих успехов. «Вначале я делю лес своей огромной сетью примерно пополам, и выясняю, в которой из двух половин очутился нужный мне зверь — пояснил охотник. — Затем половину со зверем опять делю пополам и гляжу, где он теперь. И так поступаю, пока животное не окажется в тесном загоне». И зверолов нацарапал на песке рис. 90.

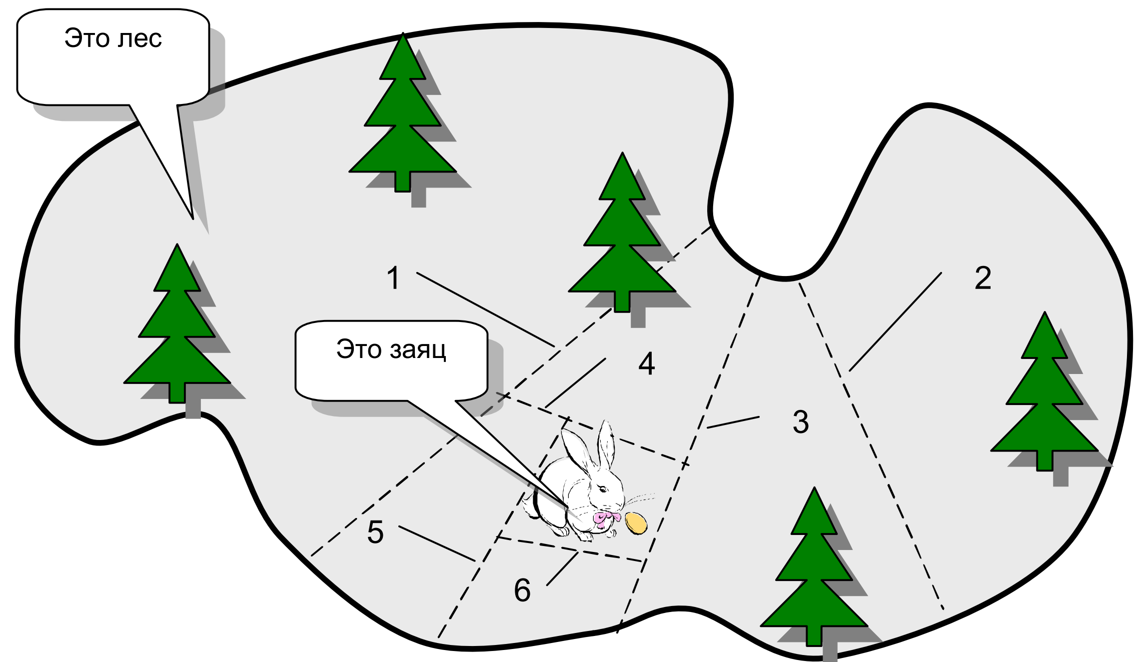


Рис. 90 – Поимка зайца шестью сетями

Здесь показано, как шестью сетями (они обозначены цифрами) был изловлен несчастный заяц. Обратите внимание на нумерацию сетей, — они расставлялись в этом порядке.

Не воспользоваться ли уловкой звероловa для поиска в массиве? Ускорит ли это дело? Конечно! Но массив должен быть заранее отсортирован. На рис. 91 показан отсортированный по возрастанию массив, содержащий 12 чисел. Для наглядности числа изображены столбиками. Среди них я выбрал наугад число 32, и прямым перебором нашел его позицию (индекс) в массиве. Очевидно, что я выполнил 8 шагов поиска, поскольку число 32 хранится в 8-м элементе массива.

А теперь применим метод звероловa. Обратимся к среднему элементу массива, индекс которого равен полу-сумме первого и последнего индексов, то есть:

```
(1+12)/2 = 6
```

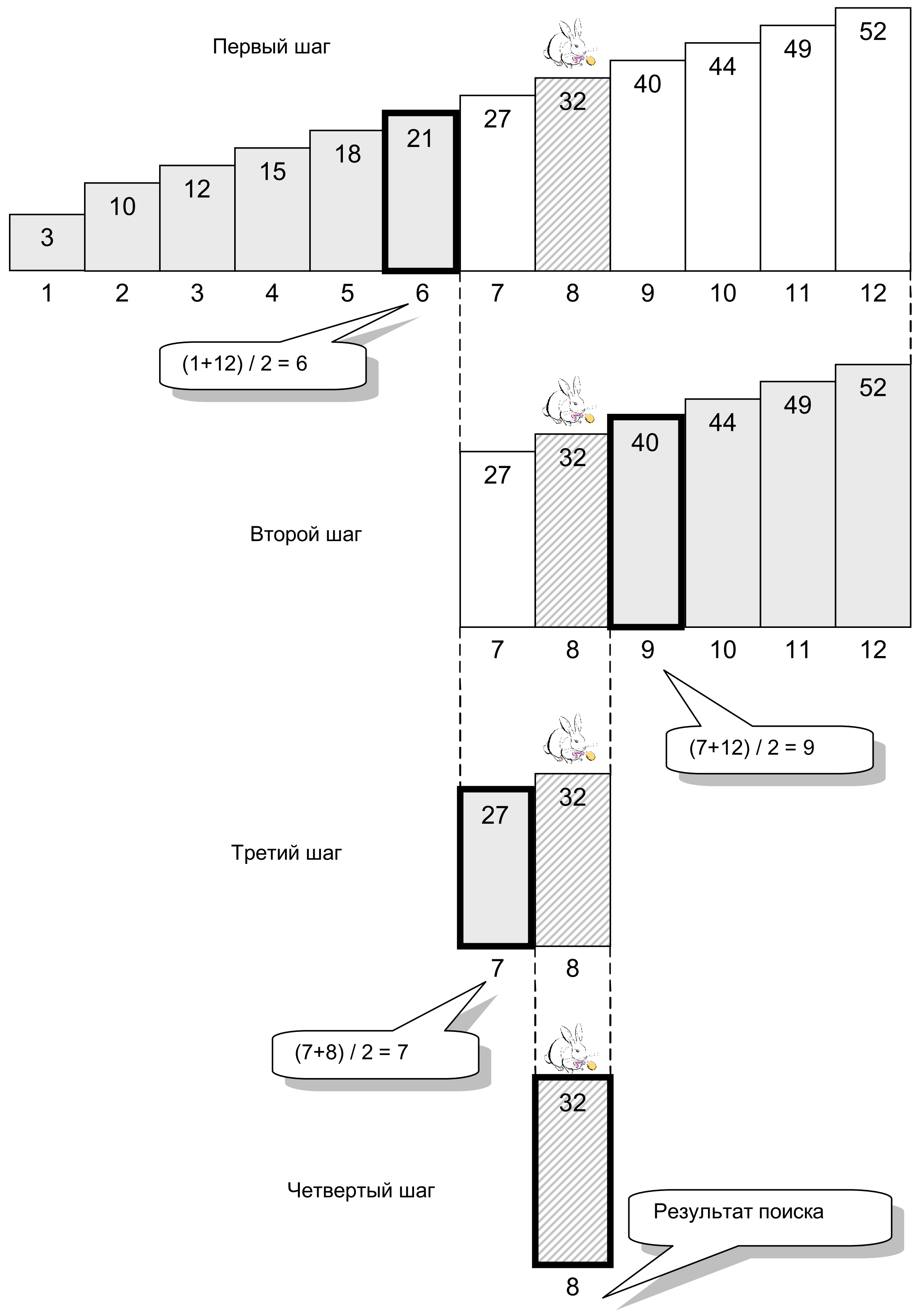


Рис. 91 – Двоичный поиск в отсортированном массиве

Поскольку индекс — это целое число, дробную часть при делении отбросим. Итак, в позиции 6 оказалось число 21, которое меньше искомого числа 32. Это значит, что «зверь притаился» где-то правее. Раз так, элементы массива, расположенные левее, нас уже не интересуют, — мысленно отбросим их.

С оставшейся частью массива поступим точно так же, то есть, исследуем средний его элемент с индексом

```
(7+12)/2 = 9
```

Сравним «живущее» там число 40 с искомым числом 32. На этот раз оно оказалось больше искомого, а значит, искать надо левее, а все, что справа, отбросить. Так, на третьем шаге поиска из 12 элементов массива остались лишь два. Рассуждая тем же порядком, выделяем элемент с индексом

```
(7+8)/2 = 7
```

и отбрасываем на этот раз число 27. И вот на последнем четвертом шаге остался лишь один элемент с искомым числом 32.

Подведем итог: вместо 8 шагов последовательного поиска, метод зверолова сделал то же самое за 4 шага. Скажете: всего-то? Восемь шагов или четыре — разница невелика. Так проверим оба метода на большом наборе данных, — поищем в массиве из тысячи чисел. Только избавьте меня от ручной работы, этот эксперимент поручим компьютеру, для чего соорудим несложную программу.

Исследование двоичного поиска

Частью этой программы будет функция двоичного поиска, алгоритм которой раскрыл звероловов. Но не худо привести и блок-схему этого чудесного изобретения. На блок-схеме (рис. 92), как и в программе, индексы элементов обозначены начальными буквами соответствующих английских слов: **L** — левый индекс (Left), **R** — правый индекс (Right), и **M** — средний индекс (Middle).

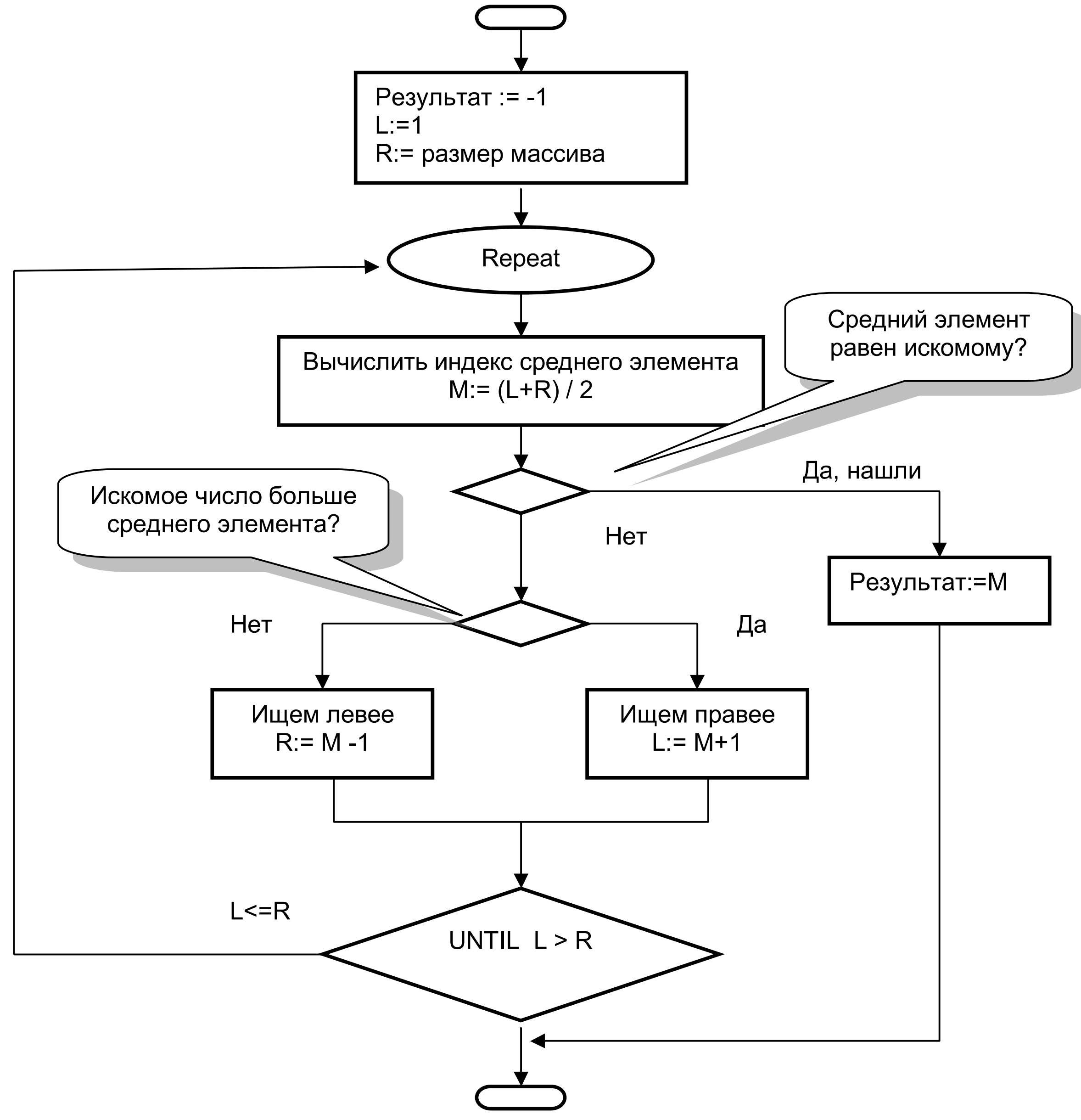


Рис. 92 – Блок-схема алгоритма двоичного поиска

Функцию, работающую по этому алгоритму, я назвал **FindBin** (Find — «поиск», Binary — «двоичный»), она показана ниже. Полагаю, что приведенных в ней комментариев будет достаточно.

```
      { Функция двоичного поиска }
function FindBin (aNum: integer): integer;
var L, M, R : integer;  { левый, правый и средний индексы }
begin
      FindBin:= -1;           { результат на случай неудачи }
      L:= 1;  R:= CSize;      { начальные значения индексов }
      repeat
            M:= (L+R) div 2;        { индекс среднего элемента }
            if aNum= ArrSort[M] then begin
                  FindBin:= M;      { нашли ! }
                  Break;            { успешный выход из цикла }
            end;
            if aNum > ArrSort[M]    { где искать дальше? }
                  then L:= M+1      { ищем правее }
                  else R:= M-1;     { ищем левее }
      until L > R;      { выход при неудачном поиске }
end;
```

Теперь мы готовы создать исследовательскую программу, которая будет сравнивать два способа поиска.

Поступим так. Объявим два массива по 1000 чисел в каждом. Заполним их случайным образом и один из них отсортируем. Затем сделаем ряд экспериментов, каждый из которых состоит в следующем. Выбрав наугад одно из чисел массива, программа вызовет по очереди две функции: сначала последовательно найдет число в несортированном массиве, а затем двоичным поиском — в сортированном. Поскольку искомое число выбрано из массива, то поиск всегда будет успешным. Затраченные на поиск шаги подсчитаем, и результаты запишем в текстовый файл. После каждого такого эксперимента программа будет ожидать команды пользователя: приняв число ноль, она завершится, а иначе повторит эксперимент.

Подсчет шагов будем вести в глобальной переменной **`Steps`** (шаги). Перед вызовом функций поиска она обнуляется, а внутри функций наращивается (эти операторы внутри функций мною подчеркнуты). Вот и всё, полюбуйтесь на эту «экспериментальную установку», введите в компьютер и запустите на выполнение.

```
            { P_42_1    - Исследование методов поиска }
const CSize = 1000; { размер массива }
            { объявление типа для массива }
Type  TNumbers = array [1..CSize] of integer;
Var   ArrRand : TNumbers;  { несортированный массив }
      ArrSort : TNumbers;  { сортированный массив }
      Steps   : integer;   { для подсчета числа шагов поиска }
{ Процедура "пузырьковой" сортировки чисел в порядке возрастания }
procedure BubbleSort(var arg: TNumbers);
var  i, j, t: Integer;
begin
  for i:= 1 to CSize-1 do           { внешний цикл }
      for j:= 1 to CSize-i do { внутренний цикл }
            if arg[j] > arg[j+1] then begin     { обмен местами }
                  t:= arg[j]; arg[j]:= arg[j+1]; arg[j+1]:= t;
            end;
end;
      { Функция последовательного поиска (Find Sequence) }
function FindSeq (aNum: integer): integer;
var i: integer;
begin
      FindSeq:= -1;            { если не найдем, результат будет -1 }
      for i:=1 to CSize do begin
            Steps:= Steps+1;    { подсчет шагов поиска }
            if aNum= ArrRand[i] then begin
                  FindSeq:= i;      { нашли, возвращаем позицию }
                  Break;            { выход из цикла }
            end;
      end;
end;
      { Функция двоичного поиска (Find Binary) }
function FindBin (aNum: integer): integer;
var L, M, R : integer;
begin
      FindBin:= -1;
      L:= 1;  R:= CSize;
      repeat
            Steps:= Steps+1;        { подсчет шагов поиска }
            M:= (L+R) div 2;
            if aNum= ArrSort[M] then begin
                  FindBin:= M;      { нашли ! }
```

```
                Break;           { выход из цикла }
          end;
          if aNum > ArrSort[M]
                then L:= M+1
                else R:= M-1;
     until L > R;
end;
                {--- Главная программа ---}
Var  i, n, p : integer;      { вспомогательные переменные }
     F: text;                { файл результатов }
begin
     Assign(F,'P_42_1.OUT');   Rewrite(F);
     { Заполняем массив случайными числами }
     for i:=1 to CSize do ArrRand[i]:=1+Random(10000);
     ArrSort:= ArrRand;      { копируем один массив в другой }
     BubbleSort(ArrSort);    { сортируем второй массив }

     repeat      { цикл с экспериментами }
          i:= 1+ Random(CSize);   { индекс в пределах массива }
          n:= ArrRand[i];         { случайное число из массива }

          Writeln(F,'Искомое число= ', n);
          Steps:=0;               { обнуляем счетчик шагов поиска }
          p:= FindSeq(n);         { последовательный поиск }
          Writeln(F,'Последовательный: ', 'Позиция= ',
                       p:3, '  Шагов= ', Steps);
          Steps:=0;               { обнуляем счетчик шагов поиска }
          p:= FindBin(n);         { двоичный поиск }
          Writeln(F,'Двоичный поиск:   ', 'Позиция= ',
                       p:3, '  Шагов= ', Steps);
          Write('Введите 0 для выхода из цикла '); Readln(n);
     until n=0;
     Close(F);
end.
```

Вот результаты трех экспериментов:

```
Искомое число= 5026
Последовательный: Позиция= 544  Шагов= 544
Двоичный поиск:   Позиция= 518  Шагов= 10
Искомое число= 8528
Последовательный: Позиция= 828  Шагов= 828
Двоичный поиск:   Позиция= 854  Шагов= 10
Искомое число= 7397
Последовательный: Позиция= 100  Шагов= 100
Двоичный поиск:   Позиция= 748  Шагов= 9
```

Я не поленился проделать 20 опытов, результаты которых занес в табл. 7. Среднее число шагов поиска для каждого из методов посчитано мною на калькуляторе и внесено в последнюю строку таблицы.

Табл. 7- Результаты исследования алгоритмов поиска

Экспе-римент	Искомое число	Количество шагов поиска	
		Последовательный поиск	Двоичный поиск
1	5026	544	10
2	8528	828	10
3	7397	100	9
4	2061	52	9
5	8227	634	9
6	9043	177	10
7	4257	10	10
8	3397	704	5
9	4021	887	10
10	8715	815	9
11	6811	53	9
12	5959	141	10
13	928	859	7
14	3295	26	10
15	9534	935	10
16	1618	8	6
17	1066	105	8
18	7081	989	10
19	218	290	9
20	6927	952	10
Среднее количество шагов		455	9

Что вы скажете об этом? Двоичный поиск дал превосходный результат: любое число находится не более чем за 10 шагов! Это любопытно, и побуждает разобраться в алгоритме глубже.

Ах, время, время!

Принимаясь за что-либо, мы прикидываем, сколько времени займет то или иное дело. Поиск может отнять уйму времени, вот почему важно оценить его трудоемкость. Сравним алгоритмы поиска по затратам времени. Только время будем измерять не секундами, а особыми единицами — шагами поиска. Почему? Да потому, что у нас с вами разные компьютеры. Поскольку ваш «станок» мощнее, ту же работу он выполнит быстрее моего, а это нечестно! Мы ведь алгоритмы сравниваем, а не процессоры.

Если улыбнется удача, поиск завершится на первом шаге. Иногда — по закону подлости — тратится максимальное число шагов. Но эти крайние случаи — редкость; обычно поиск занимает какое-то промежуточное время, и наш эксперимент подтвердил это. Программистов интересует время поиска в двух случаях: в худшем, и в среднем (то есть, усредненное по многим случаям).

Начнем с линейного поиска. Очевидно, что в массиве из **N** элементов худшее время поиска составит **N** шагов. Что касается среднего времени, то чутье подсказывает, что оно составит половину максимального времени, то есть **N/2**. Судите сами: искомое число с равной вероятностью может оказаться и ближе и дальше середины массива. Табл. 7 подтверждает эту догадку, — среднее количество шагов там составило **455**, что очень близко к значению **1000/2**.

Теперь рассмотрим двоичный поиск. Вначале оценим худшее время. Рассудим так. Сколько шагов поиска нужно в массиве из одного элемента? Правильно, один. А теперь вспомним, что при двоичном поиске всякий раз отбрасывается половина оставшегося массива. Значит, посчитав, сколько раз число **N** делится пополам для получения единицы, мы определим максимальное число шагов. Так и поступим; следите, честно ли я «распилил» нашу тысячу:

```
1. 1000 / 2 = 500
2. 500 / 2 = 250
3. 250 / 2 = 125
4. 125 / 2 = 62
5. 62 / 2 = 31
6. 31 / 2 = 15
7. 15 / 2 = 7
8. 7 / 2 = 3
9. 3 / 2 = 1
```

При делении я отбрасывал дробную часть, поскольку в двоичном алгоритме так и делается. Всего потребовалось 9 операций деления. Это значит, что максимальное число шагов поиска равно 10 (с учетом поиска в одном оставшемся элементе). Удивительная прозорливость, — ведь наш эксперимент (табл. 7) показал то же самое!

Теперь оценим среднее время двоичного поиска. Думаете, что оно составит **10/2 = 5** шагов? Как бы ни так! Дело в том, что любой алгоритм поиска в среднем исследует половину массива. Двоичный поиск отбрасывает половину массива на первом же шаге. А это значит, что в среднем число шагов будет всего лишь на единицу меньше худшего, то есть 9. Смотрим в табл. 7, — точно! Наша догадка подтвердилась! Таким образом, двоичный поиск не только быстрее линейного, но и более предсказуем: его худшее время почти не отличается от среднего.

Логарифмы? Это просто!

Разобравшись с тысячей элементов, оценим трудоемкость двоичного поиска при других размерах массива. Метод оценки остаётся тем же: делим размер массива пополам до получения единицы.

Для таких вычислений математики придумали особую функцию — логарифм (не путайте её с рифмой, ритмом и алгоритмом!). Логарифмы бывают разные: десятичные, натуральные и прочие. Нам интересен двоичный логарифм, который по-научному называется так: «логарифм числа **N** по основанию два». Математики записывают его следующим образом:

$$\mathbf{Log_2\ N}$$

Связь между числом **N** и его двоичным логарифмом легко проследить на следующих примерах. Слева представлено разложение на множители нескольких чисел, а справа — двоичные логарифмы этих же чисел.

$4 = 2 \cdot 2$	$Log_2\ 4 = 2$
$16 = 2 \cdot 2 \cdot 2 \cdot 2$	$Log_2\ 16 = 4$
$64 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$	$Log_2\ 64 = 6$

Итак, двоичный логарифм числа равен количеству двоек (ой, нехорошее слово!), перемножаемых для получения этого числа. Например, для получения числа 8 надо перемножить три двойки, и его логарифм равен трем. Кстати, для получения единицы из восьмерки, её тоже «пилят» пополам трижды. Значит, оба способа вычисления логарифма — через умножение, и через деление — равноценны.

Если вы завтра же не забросите программирование, то табл. 8 с логарифмами нескольких чисел ещё пригодится вам.

Табл. 8 – двоичные логарифмы некоторых чисел

N	Log_2 N	N	Log_2 N	N	Log_2 N	N	Log_2 N
2	1	32	5	512	9	8192	13
4	2	64	6	1024	10	16384	14
8	3	128	7	2048	11	32768	15
16	4	256	8	4096	12	65536	16

По таблице можно оценить как среднее, так и худшее время двоичного поиска: среднее время равно двоичному логарифму от размера массива, а худшее — на единицу больше.

А как определить логарифмы других чисел, например, числа 50? Поскольку оно лежит между 32 и 64, его логарифм должен быть где-то между 5 и 6? Так оно и есть: логарифм 50 равен приблизительно 5,64 (это я на калькуляторе посчитал). Но, поскольку мы применяем логарифмы для подсчета шагов поиска, то погрешностью в доли шага можно пренебречь. К чему мелочиться? Будем считать, что логарифм числа 50 тоже равен 6. Мало того, назначим это значение логарифма всем числам в промежутке от 33 до 64.

На рис. 93 сопоставлен рост числа с ростом его логарифма. Когда число увеличивается вдвое, его логарифм возрастает лишь на единицу. Вот почему с ростом размера массива время двоичного поиска растет так медленно (что очень радует нас!).

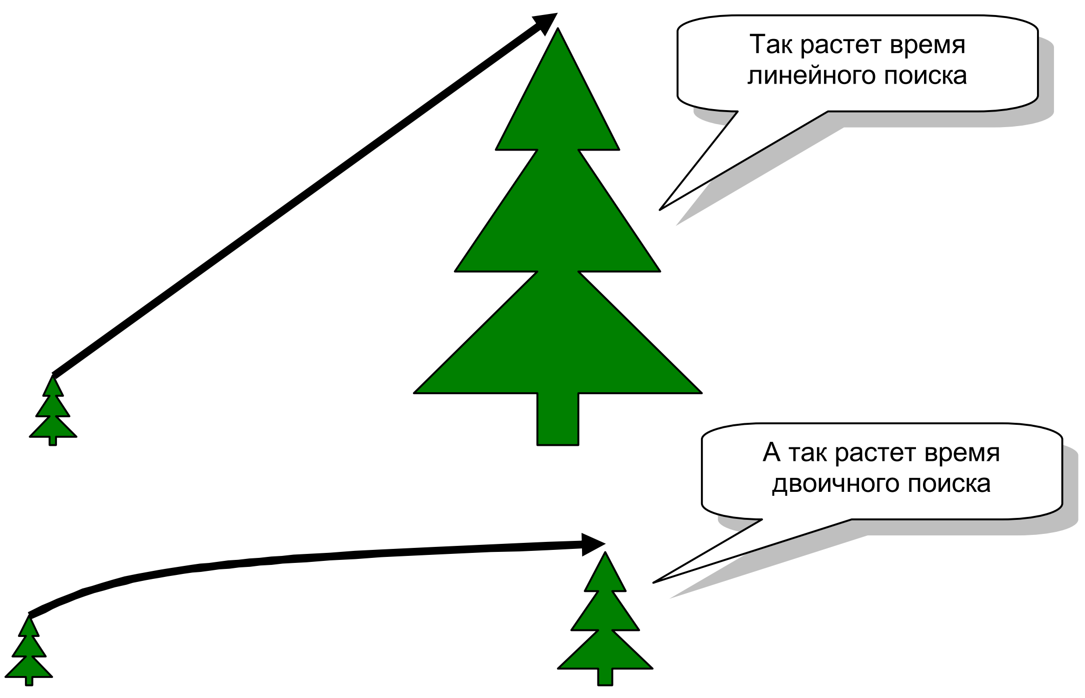


Рис. 93 – Сравнение времени линейного и двоичного поиска

Итоги

- Компьютерные базы данных (БД) содержат разнородную информацию, отдельные элементы которой связаны общим индексом.
- Поиск в массиве состоит в определении индекса искомого элемента; зная индекс, можно извлечь всю прочую информацию о нужном объекте.
- Для поиска применяют два способа: последовательный перебор и двоичный поиск.
- Последовательный перебор (линейный поиск) очень прост, но время поиска пропорционально размеру массива, что для больших объемов данных бывает неприемлемо.
- Двоичный поиск очень быстр, – с ростом размера массива затраты времени на поиск растут по логарифмическому закону. Однако, двоичный поиск работает только в отсортированных массивах.

А слабо?

А). Будет ли линейный поиск работать быстрее в сортированном массиве? Проверьте на практике.

Б) Сколько шагов двоичного поиска потребуется в массиве из миллиона элементов? А из миллиарда? Сравните с трудоемкостью линейного поиска.

В) Напишите полицейскую базу данных, содержащую номера автомобилей и сведения о владельцах. Данные должны вводиться из файла, каждая строка которого содержит номер автомобиля и сведения о владельце, например:

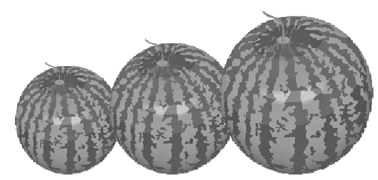
```
123 Горбунков С.С., ул. Тепличная, д. 21, тел. 11-22-33
35 Стелькин И.Н., ул. Тенистая, д. 5, тел. 33-22-11
```

Примените массивы и учтите опыт обработки классного журнала.

Г) Отсортируйте полицейскую базу данных и напишите программу для двоичного поиска в ней.

Д) Папа Карло опасался Буратино, и прятал спички в сейфе. Код замка из четырех цифр он доверил лишь своему приятелю — честному малому Джузеппе, который не поддавался ни на какие уговоры деревянного мальчишки. Тогда тот пустился на хитрость. Ладно, — предложил Буратино, — не можешь открыть мне код, — не надо. Давай тогда в игру сыграем: я буду спрашивать, а ты отвечай только «да» или «нет». Первый вопрос был таким: код замка больше 5000? Через несколько минут Буратино уже рылся в папином сейфе. Сделайте программу для быстрого угадывания числа методом Буратино. Роль Буратино (угадывающего) должен исполнять компьютер.

Глава 43
Сортировка по-взрослому



Наше новейшее открытие — быстрый, как ракета, двоичный поиск. Но работает он лишь в сортированном массиве. Так в чем вопрос? Разве сортировка «пузырьком» нам не покорилась? Увы! «Пузырек» насколько же нетороплив, насколько и прост. Много ли проку от быстрого поиска, если выигрыш времени съест сортировка? Так ускорим её!

«Фермерская» сортировка

Отчего так медленно «всплывают пузырьки»? Не оттого ли, что мы сравниваем и обмениваем соседние элементы? Уяснить тонкости сортировки нам поможет правдивая история из жизни двух друзей.

Некий фермер — левша по имени Лефт — удостоился чести поставлять арбузы к столу Его Величества. Желая превзойти конкурентов и сохранить за собой королевский заказ, Лефт решил отбирать из урожая самые крупные ягоды (арбуз — это ягода). Выложив арбузы в длинный ряд, Лефт занялся их сортировкой. Работая в одиночку, он применил единственный доступный ему способ — «пузырёк». После трёх дней мучительных сравнений и перестановок Лефт понял, что без помощника ему не обойтись.

Сортировать урожай следующего года он позвал своего соседа и приятеля по имени Райт. Вдвоём они стали работать новым, придуманным Лефтом способом. Лефт стал у первого арбуза, а его приятель Райт побежал в конец ряда. Оттуда Райт стал продвигаться навстречу Лефту, сравнивая арбузы с тем, у которого прохлаждался Лефт (арбузы взвесили заранее, а вес нацарапали на кожуре). Когда Райт находил арбуз легче того, у которого стоял Лефт, их меняли местами: друзья просто швыряли арбузы друг другу.

Наконец Райт вплотную подошел к Лефту, и тогда на месте, где стоял Лефт, оказался самый легкий арбуз. Лефт шагнул ко второму арбузу, а Райт снова побежал в конец ряда, и всё повторилось. По окончании второго цикла на втором месте в ряду, где стоял Лефт, очутился второй по величине арбуз. Теперь первые два арбуза были отсортированы, и Лефт соизволил шагнуть к третьему. К сумеркам, совершив **`N-1`** таких циклов, друзья закончили работу. Лефт, свежий как огурчик, ступил, наконец, к последнему арбузу, недоумевая, отчего его приятель Райт едва волочит ноги?

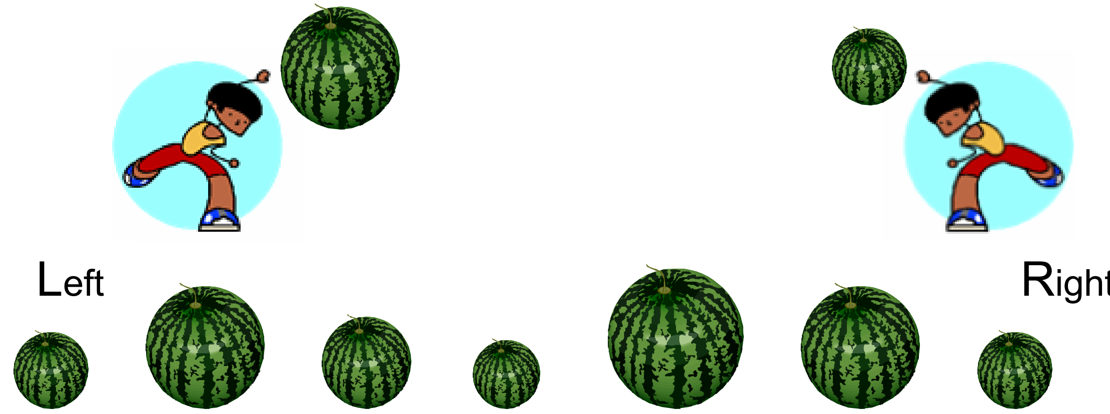


Рис. 94 – Сортировка массива с встречным движением индексов

Пусть друзья отдыхают, а мы поразмыслим, много ли толку в изобретении Лефта? Поскольку при каждом обмене арбузы перемещались на большое расстояние, то возможно, что таких обменов потребовалось меньше, чем при «пузырьке». Пока это лишь догадка, которую предстоит проверить. А пока соорудим и испытаем процедуру сортировки, придуманную Лефтом, назовём её **FarmSort** — «фермерская» сортировка.

Программа с процедурой перед вами. Введите её и проверьте, действительно ли процедура Лефта сортирует массив, не ошибся ли фермер?

```
{ P_43_1  Программа для проверки «фермерской» ссортировки }

const
      CSize=10;    { размер массива }
type
      TNumbers = array [1..CSize] of Integer;  { тип для массива }
var
      Arr  : TNumbers;        { сортируемый массив }
```

```
      { Процедура «фермерской» сортировки }
procedure FarmSort(var arg: TNumbers);
var  L, R, T: Integer;
begin
  for L := 1 to CSize-1 do
      { Сдвигаем правый индекс влево }
      for R := CSize downto L+1 do begin
            { Если левый элемент оказался больше правого,
            то меняем элементы местами }
            if arg[L] > arg[R] then begin
                  { Перестановка элементов массива }
                  T:= arg[L];  arg[L]:= arg[R]; arg[R]:= T;
            end;
      end;
end;

      { Процедура распечатки массива, arg - строка сообщения }
procedure ShowArray(const arg: string);
var i: integer;
begin
      Writeln(arg);
      for i:=1 to CSize do Writeln(Arr[i]);
      Readln;
end;
var i: integer;
begin
      { Заполняем массив случайными числами }
      for i:=1 to CSize do Arr[i]:=1+Random(1000);
      ShowArray('До сортировки:');
      FarmSort(Arr);
      ShowArray('После сортировки:');
end.
```

Быстрая сортировка

«Здесь что-то не так, — стучало в голове Райта, пока он челноком мотался из конца в конец ряда, — почему я бегаю, а он стоит? Это несправедливо! К следующему урожаю я придумаю лучший способ сортировки!».

Через год Лефт опять позвал Райта на помощь.

Хорошо, — согласился Райт, — но теперь командовать буду я.

Пройдясь вдоль ряда, Райт прикинул на глазок вес среднего по величине арбуза. «Запомни этот вес, — сказал он Лефту, — и ступай к началу ряда», — а сам отправился в другой конец. «Теперь иди ко мне, пока не найдешь арбуз тяжелее указанного мною». Лефт так и сделал, — найдя первый такой арбуз, он остановился и крикнул об этом Райту. «Теперь моя очередь!» — отозвался Райт и стал двигаться навстречу Лефту, попутно отыскивая арбуз легче среднего. Дойдя до такого арбуза, Райт остановился и скомандовал: «Меняемся арбузами!», — и друзья швырнули арбузы друг другу.

— Снова твоя очередь! — крикнул Райт, — продолжай двигаться ко мне! — а сам присел отдохнуть.

Так поочередно друзья шли навстречу друг другу, время от времени обмениваясь арбузами. Где то в середине ряда они встретились.

— Ну и чем обернулась твоя идея со средним арбузом? — ядовито осведомился Лефт, — мы уже встретились, не отсортировав ни единого!

— Да, — согласился Райт, — зато любой арбуз на твоей левой половине ряда легче любого на моей правой половине.

— Откуда ты знаешь?

— Оттуда! Ведь все твои арбузы легче среднего, а все мои — тяжелее!

— Да, пожалуй, так, но что нам это даёт? — не унимался Лефт.

— А то, что теперь эти две половинки ряда можно сортировать отдельно. Смекнул? Нам меньше бегать придется, ведь расстояния вдвое короче!

— И как же мы будем сортировать эти половинки?

— Тем же порядком, но по очереди, — сначала твою половину, потом мою.

И Райт стал прикидывать средний вес арбузов в левой половине ряда. Этот средний вес был, разумеется, меньше того, что в первом случае. Переведя дух, друзья повторили те же действия с левой половиной ряда. В серединке этой половинки они встретились вновь.

— Видишь, как быстро мы сошлись, — отметил Райт, — а всё потому, что ряд стал вдвое короче. Теперь все арбузы в левой четвертинке легче тех, что в правой четвертинке. Продолжим действовать так же, и отсортируем четвертинки по отдельности.

И фермеры продолжили дележку ряда: первую четвертинку разбили на две осьмушки, первую осьмушку — на две шестнадцатых и так далее, пока кусочек ряда не съежился до одного-двух арбузов. Этот кусочек они отсортировали моментально и пружина стала раскручиваться в обратную сторону. В конце

концов, они вернулись к оставленным ранее частям ряда и отсортировали вторую осьмушку, вторую четвертинку и вторую половинку.

— Готово! — радостно выдохнул Райт. — Гляди-ка, ещё утренняя роса не просохла!

— Нич-ч-чо не понимаю, — сдался Лефт, — но ты, похоже, гений!

И приятели отправились завтракать.

Процедура быстрой сортировки

Друзья заслужили отдых, и теперь наш черед. Возьмем алгоритм Райта и проверим, так ли он хорош?

В целом алгоритм ясен, неясно лишь, как выбрать средний арбуз? В идеале его вес должен быть таким, чтобы половина арбузов в сортируемой части массива была легче среднего, а другая половина — тяжелее. Только тогда массив будет разрублен строго пополам. Увы! У нас нет простого способа найти вес такого арбуза! Даже усреднив веса арбузов в сортируемой части, мы можем не угадать это число.

К счастью, всё не так уж плохо. Опыт показал, что делить массив строго пополам совсем не обязательно. Например, при делении ряда в пропорции 1/3 и 2/3 сортировка почти не ухудшится. Значит, можно оценивать вес среднего арбуза «на глазок» (как это делал Райт). Будем вычислять его как среднее арифметическое для трех арбузов: двух крайних и того, что лежит в середине сортируемой части массива.

Тогда формула для определения веса среднего арбуза будет такой:

```
Средний вес := (Вес[L] + Вес[(L + R)/2] + Вес [R]) / 3;
```

Здесь **L** и **R** — индексы элементов для начала и конца сортируемой части массива. Повторяю: это лишь один из возможных вариантов определения среднего веса.

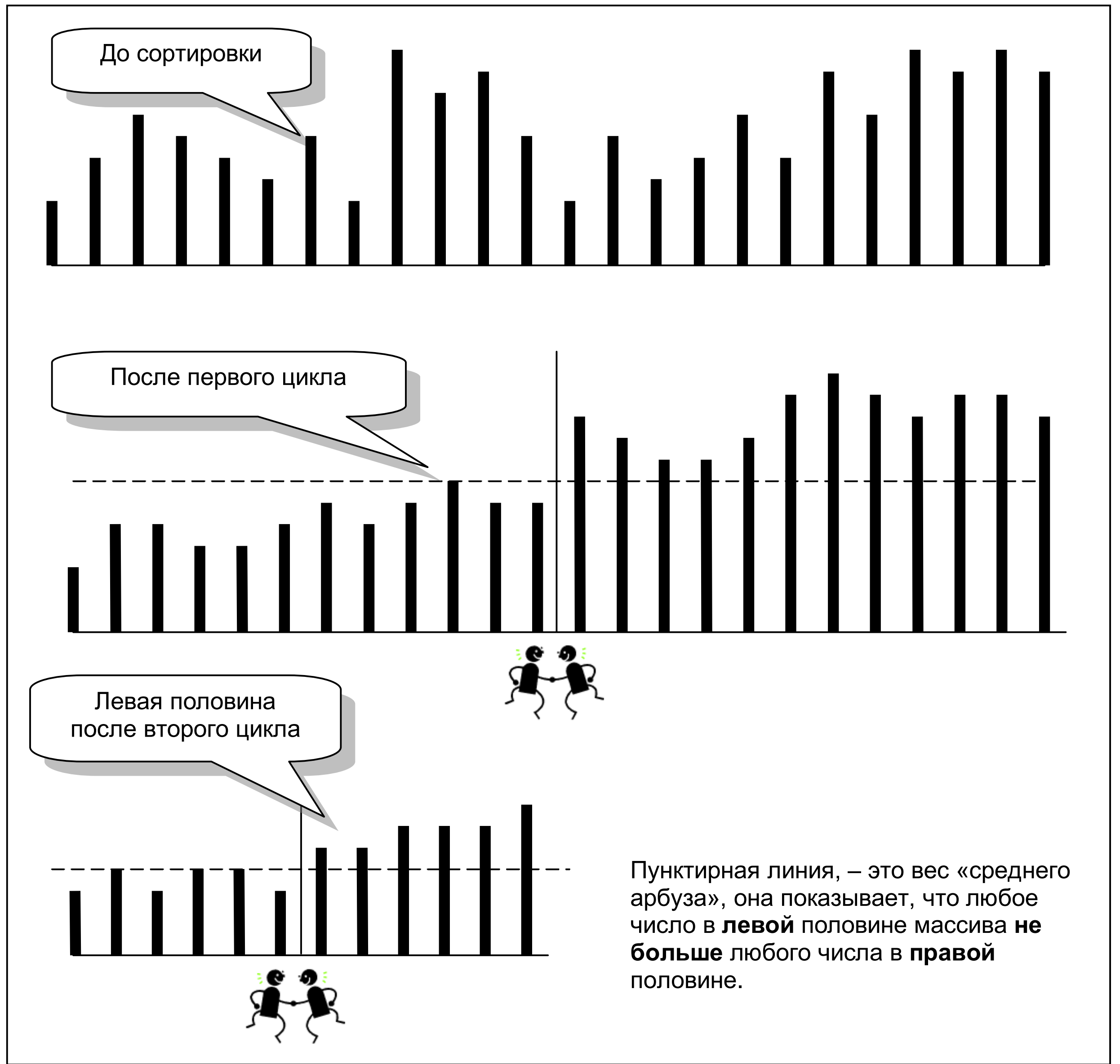


Рис. 95 – Изменения массива при быстрой сортировке

Вы принимаете эту формулу? Тогда перейдем к процедуре быстрой сортировки по имени **`QuickSort`** (Quickly — «быстро», Sort — «сортировка»). Вот она вместе с проверяющей её программой.

```
{ P_43_2    QuickSort  – Быстрая сортировка }

const CSize=10;    { размер массива }
type TNumbers = array [1..CSize] of Integer;
var  Arr  : TNumbers;

      { Процедура быстрой сортировки }
procedure QuickSort(var arg: TNumbers; aL, aR: Integer);
var
      L, R : integer; { левый и правый индексы }
      M, T : Integer; { среднее значение и временное хранилище }
begin
      { Начальные значения левого и правого индексов }
      L:= aL;    R:= aR;
      { Вычисляем среднее по трём (порог для сравнения ) }
      M:= (arg[L] + arg[(L + R) div 2] + arg[R]) div 3;
      repeat      { Цикл встречного движения }
            { Пока левый элемент меньше среднего,
            двигаем левый индекс вправо }
            while arg[L] < M do L:=L+1;
            { Пока правый элемент больше среднего,
            двигаем правый индекс влево }
            while arg[R] > M do R:=R-1;
            { После остановки сравниваем индексы }
            if L <= R then begin
                  { Здесь индексы ещё не "встретились", поэтому,
                  если левый элемент оказался больше правого,
                  меняем их местами }
                  if arg[L]>arg[R] then begin
                        t:= arg[L];  arg[L]:= arg[R]; arg[R]:= t;
                  end;
                  { Индексы «делают шаг» навстречу друг другу }
                  L:=L+1;  R:=R-1;
            end;
      until L > R; { пока индексы не "встретятся" }
      { если левая часть не отсортирована, то сортируем её }
      if R > aL then QuickSort(arg, aL, R);
      { если правая часть не отсортирована, то её тоже сортируем }
      if L < aR then QuickSort(arg, L, aR);
      { выход после сортировки обеих частей }
end;
```

```
      { Процедура распечатки массива, arg - строка сообщения }
procedure ShowArray(const arg: string);
var i: integer;
begin
      Writeln(arg);
      for i:=1 to CSize do Writeln(Arr[i]);
      Readln;
end;
var i: integer;
begin       {--- Главная программа ---}
      { Заполняем массив случайными числами }
      for i:=1 to CSize do Arr[i]:=1+Random(1000);
      ShowArray('До сортировки:');
      QuickSort(Arr, 1, CSize);
      ShowArray('После сортировки:');
end.
```

Взгляните на параметры процедуры **QuickSort**. Вместе со ссылкой на массив, в процедуру передаются левая (**aL**) и правая (**aR**) границы сортируемой части массива (индексы). В процедуре вычисляется вес среднего арбуза по выбранной нами формуле и организуется поочередное встречное движение левого и правого индексов.

Самое интересное происходит после «встречи» индексов, когда массив разбит на две части. Удивительно, что теперь снова дважды вызывается та же самая процедура **QuickSort**: сначала для левой части массива, а затем — для правой (эти операторы мною подчеркнуты). Вспомните, — точно так же поступали и фермеры при сортировке арбузов.

«Так там фермеры, а здесь Паскаль! Позволено ль процедуре вызывать саму себя?» — слышу недоверчивый вопрос. Мы свыклись с тем, что из одной процедуры вызывают другую, из второй, — третью и так далее. Но чтобы саму себя? Это ж змея, глотающая свой хвост! Не оттого ли запутался фермер Лефт?

О рекурсии и стеке

Такой самовызов процедур называют рекурсией. «У попа была собака…» — помните? Это рекурсия, познакомимся с нею ближе (с рекурсией, а не собакой).

Легко заметить, что повторные вызовы процедуры **QuickSort** выполняются с другими значениями левой и правой границ. Чем глубже вызов, тем уже эти границы. С некоторого момента условия (**R > aL)** и (**L < aR)** перестают выполняться, и мы выходим из процедуры, — здесь фермеры возвращаются к несортированным частям массива. И тогда при выходе мы снова попадаем в эту же

процедуру, но в другое место — следующее за вызовом. Окончательный выход из процедуры в главную программу случится лишь по завершении сортировки всего массива.

Напрашивается вопрос: какова судьба локальных переменных и параметров при повторных входах в процедуру? Ведь они изменятся, а это должно нарушить работу процедуры. Параметры и локальные переменные действительно изменяются, но это не путает алгоритм. Почему?

Разгадка в том, что при каждом входе в процедуру для её параметров и локальных переменных выделяется новый участок памяти. Теперь это будут уже другие параметры и локальные переменные, но с прежними названиями. Однако предыдущие их значения не теряются, а сохраняются в памяти, называемой стеком (Stack).

Что такое стек и как он работает? Случалось ли вам паковать рюкзак или глубокую сумку? Тогда вы знакомы со стеком. Всё, что уложено в рюкзак, будет извлекаться из него в обратном порядке. Так же устроен и стек: при каждом вызове процедуры память для параметров и локальных переменных выделяется на его вершине. Эти новые значения временно закрывают предыдущие значения параметров, и так происходит при каждом входе в процедуру.

При выходе из неё последние значения параметров и локальных переменных удаляются с вершины стека, и тогда вновь открываются ранее скрытые. Так процедура «вспоминает» о неоконченной работе, и продолжает действия с параметрами, сохраненными в стеке ранее. В некоторый момент стек пустеет (когда все вещи из рюкзака вынуты), и тогда происходит окончательный выход из процедуры в главную программу.

Механизм стековой памяти заложен в конструкцию процессора, и он не требует участия программиста. Стек используется для любых процедур и функций, а не только рекурсивных.

Но стоп! О стеке поговорим позже, а сейчас займемся делом. Введите программу P_43_2 и убедитесь в её правильной работе.

Алгоритмы, на старт!

Теперь в нашем распоряжении есть три процедуры сортировки, не устроить ли состязание между ними? На старт вызываются:

- **BubbleSort** — «пузырьковая» сортировка,
- **FarmSort** — «фермерская» сортировка,
- **QuickSort** — быстрая сортировка.

Время «спортсменов» мы будем засекать не по часам. И вы знаете, почему: мы сравниваем алгоритмы, а не компьютеры. В 42-й главе, где сравнивались

алгоритмы поиска, мы оценивали время по количеству выполненных шагов. Поступим и здесь похожим образом. Вспомним, в чем состоит сортировка? — в сравнениях и перестановках. И много-много раз... Значит, трудоемкость сортировки можно оценить количеством этих двух операций — сравнений и перестановок, надо их только посчитать.

Что ж, дело нехитрое, сейчас посчитаем, — перед вами программа для наших опытов (P_43_3). Количество сравнений и перестановок будем накапливать в переменных **C1** и **C2**. Обратите внимание на их тип — **EXTENDED**, — это переменные действительного типа. Почему не длинное целое? При сортировке больших массивов может потребоваться столь много операций, что не хватит целочисленной переменной, — она переполнится, «лопнет», и результат исказится. Потому выбран тип **EXTENDED**.

Далее в программе следуют знакомые вам процедуры сортировки, — это наши «спортсмены». В их тела «вживлены» дополнительные операторы для подсчета сравнений (**C1**) и перестановок (**C2**), — они мною подчеркнуты. Наконец, главная программа, — она вызывает по очереди каждую из процедур и печатает количество сравнений и перестановок. Здесь равные условия для соревнующихся создаются загодя сформированным случайным массивом **Arr0**, который копируется в массив **Arr** перед каждой сортировкой.

Вам осталось лишь задать размер массива константой **CSize**, скомпилировать и запустить программу.

```
{ P_43_3    - Сравнение алгоритмов сортировки }
const CSize=100;    { размер массивов }

type  TNumbers = array [1..CSize] of Integer;
var   Arr0 : TNumbers;    { несортированный массив-заготовка }
      Arr  : TNumbers;    { сортируемый массив }
      C1, C2 : extended; { счетчики сравнений и перестановок }

{ BubbleSort "пузырьковая" сортировка }
procedure BubbleSort(var arg: TNumbers);
var  i, j, t: Integer;
begin
      for i:= 1 to CSize-1 do
            for j:= 1 to CSize-i do begin
                  C1:=C1+1; { подсчет сравнений }
                  if arg[j] > arg[j+1] then begin
                        C2:=C2+1; { подсчет перестановок }
                        t:= arg[j]; arg[j]:= arg[j+1];  arg[j+1]:= t;
                  end;
            end;
end;

{ FarmSort – «фермерская» сортировка }
procedure FarmSort(var arg: TNumbers);
var  L, R, T: Integer;
begin
      for L := 1 to CSize-1 do
            for R := CSize downto L+1 do begin
                  C1:=C1+1; { подсчет сравнений }
                  if arg[L] > arg[R] then begin
                        C2:=C2+1; { подсчет перестановок }
                        T:= arg[L];  arg[L]:= arg[R]; arg[R]:= T;
                  end;
            end;
end;

{ QuickSort – Быстрая сортировка }
procedure QuickSort(var arg: TNumbers; aL, aR: Integer);
var    L, R, Mid, T: Integer;
begin
  L:= aL;    R:= aR;
```

```
  Mid:= (arg[L] + arg[(L + R) div 2] + arg[R]) div 3;
  repeat
      while arg[L] < Mid do begin L:=L+1; C1:=C1+1 end;
      while arg[R] > Mid do begin R:=R-1; C1:=C1+1 end;
      if L <= R then begin
            if arg[L]>arg[R] then begin
                  C2:=C2+1; { подсчет перестановок }
                  t:= arg[L];  arg[L]:= arg[R]; arg[R]:= t;
            end;
            L:=L+1;  R:=R-1;
      end;
      until L > R;
      if R > aL then QuickSort(arg, aL, R);
      if L < aR then QuickSort(arg, L, aR);
end;
const CFName = 'P_43_3.out';

var i: integer;
    F: text;
begin
  Assign(F,CFName);  Rewrite(F);
  for i:=1 to CSize do Arr0[i]:=1+Random(10000);
  Writeln(F, 'Размер массива = ', CSize);
  Writeln(F, 'Алгоритм       Количество    Количество');
  Writeln(F, 'сортировки     сравнений    перестановок');
  C1:=0; C2:=0;        { очистить счетчики }
  Arr:= Arr0;          { заполнить сортируемый массив }
  BubbleSort(Arr);
  Writeln(F, 'Пузырьковая:', C1:12:0, C2:12:0);
  C1:=0; C2:=0;        { очистить счетчики }
  Arr:= Arr0;          { заполнить сортируемый массив }
  FarmSort(Arr);
  Writeln(F, '«Фермерская»     :', C1:12:0, C2:12:0);
  C1:=0; C2:=0;        { очистить счетчики }
  Arr:= Arr0;          { заполнить сортируемый массив }
  QuickSort(Arr, 1, CSize);
  Writeln(F, 'Быстрая    :', C1:12:0, C2:12:0);
  Writeln('OK !');  Readln;
  Close(F);
end.
```

Вот что получилось для массива из 1000 элементов.

```
Размер массива = 1000
Алгоритм        Количество     Количество
сортировки      сравнений     перестановок
Пузырьковая:      499500       248061
Фермерская :      499500        80887
Быстрая    :        5871         2417
```

Проведя три опыта с массивами из 100, 1000 и 10000 элементов, я получил результаты, представленные в двух табличках. Что сказать по этому поводу?

Табл. 9 – Количество сравнений в разных методах сортировки

Размер массива	Пузырьковая сортировка	Фермерская сортировка	Быстрая сортировка
100	4 950	4 950	417
1 000	499 500	499 500	5 871
10 000	49 995 000	49 995 000	79 839

Из табл. 9 следует, что по количеству сравнений «фермерская» сортировка не лучше «пузырька». Зато быстрая сортировка оправдывает свое название, — выигрыш составляет от 10 до 600 раз! И чем больше массив, тем заметней этот разрыв.

Табл. 10 – Количество перестановок в разных методах сортировки

Размер массива	Пузырьковая сортировка	Фермерская сортировка	Быстрая сортировка
100	2 305	805	141
1 000	248 061	80 887	2 417
10 000	24 903 994	6 154 077	31 011

А что с количеством перестановок (табл. 10)? Здесь «фермерская» сортировка всё же в 3 — 4 раза обгоняет «пузырёк». Так что фермеру Лефту в сметливости не откажешь. И всё же этому «спортсмену» далеко до изобретения Райта! В сравнении с «пузырьком» выигрыш стократный, и стремительно растёт с ростом размера массива. Слава, слава фермеру Райту! Кстати, историки выяснили его настоящее имя: это англичанин Чарльз Хоар.

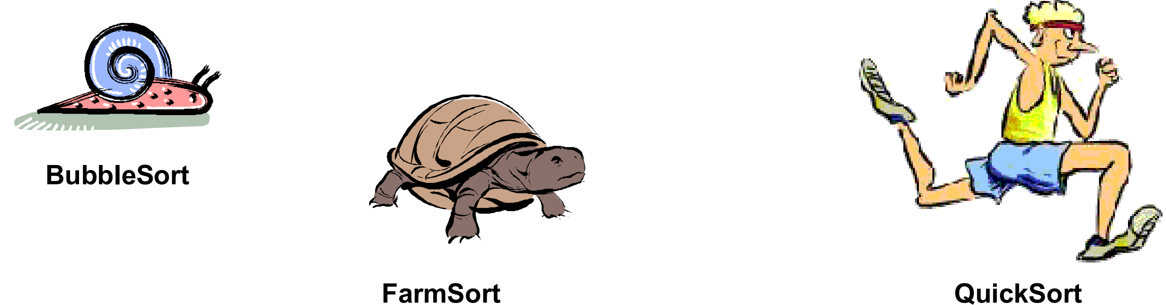


Рис. 96 – Кто здесь чемпион?

Итак, с чемпионом все ясно (рис. 96), но в чем секрет его успеха? Сравним чемпиона с отставшими. Чем больше массив, тем дольше он сортируется, — это справедливо для любого алгоритма. Долгие методы обрабатывают весь массив **N** раз, где **N** — размер массива. Поэтому их трудоёмкость пропорциональна произведению **N•N**. По этой формуле вычисляют площадь квадрата, и такую зависимость называют квадратичной.

Иное дело — чемпион. Дробя массив на мелкие участки, быстрый алгоритм уменьшает число прохождений всего массива до величины **Log_2N**. Поэтому его трудоемкость растёт пропорционально произведению **N•Log_2N**. Опять логарифм? Да, мы помним его по двоичному поиску. Логарифм числа растёт очень медленно, и это объясняет чемпионство **QuickSort** (рис. 97).

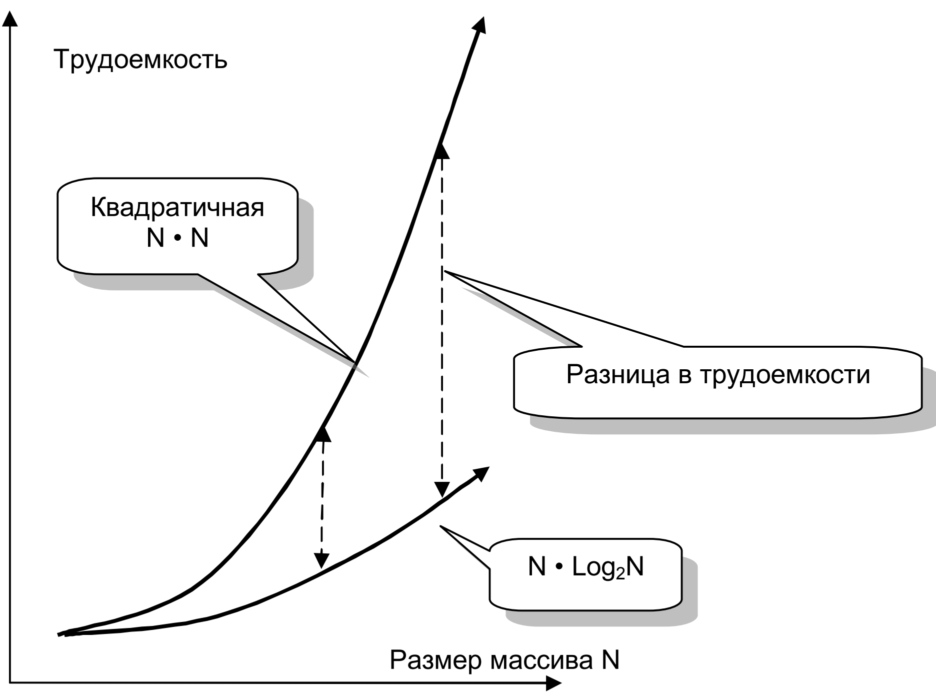


Рис. 97 – Рост трудоемкости сортировки с ростом размера массива

Итоги

- Двоичный поиск – это самый быстрый способ поиска, но он требует предварительной сортировки массива.
- Простые методы сортировки – BubbleSort и FarmSort – работают очень медленно, съедая всю экономию от двоичного поиска.
- QuickSort – это быстрый способ сортировки, который в сочетании с резвым двоичным поиском ускорит работу ваших программ.

А слабо?

А) Исследуйте процедуру быстрой сортировки в пошаговом режиме (задайте небольшой размер сортируемого массива). Обратите внимание на изменение границ сортируемой части.

Б) Определите количество повторных входов в процедуру **QuickSort** и выходов из нее. Объявите глобальную переменную, назовем её **Level** — «уровень». В главной программе, перед вызовом процедуры **QuickSort**, эту переменную надо обнулить, а внутри процедуры добавить следующие операторы.

В начале процедуры **QuickSort**:

```
begin
      Inc(Level); Writeln('Уровень на входе = ', Level);
```

В конце процедуры **QuickSort**:

```
      Dec(Level); Writeln('Уровень на выходе = ', Level);
end;
```

В) Если каждый вызов **QuickSort** делит массив примерно пополам, то наибольшее значение переменной **Level** должно составить приблизительно $\mathbf{Log_2 N}$ (у нас размер массива задан константой **CSize**). Проверьте эту догадку компиляцией и запуском программы для нескольких значений **CSize**.

Г) В одном ряду вперемежку лежат дыни и арбузы. Могут ли фермеры отсортировать их за один проход ряда так, чтобы в начале оказались все дыни, а в конце ряда — все арбузы? Напишите такую программу, обозначив арбузы единицами, а дыни — нулями.

Глава 44
Строки

Строковый тип **STRING** известен нам с первых глав книги, без него компьютер не общался бы с нами на «человечьем» языке. Так изучим строки получше. В современных версиях Паскаля применяют несколько строковых типов, сейчас мы рассмотрим только короткие строки, введенные ещё в Borland Pascal (в новых версиях Паскаля этот тип называется **ShortString**).

Строка – особый род массива

С первого взгляда строка похожа на массив символов. Так ли это? — проверим. Известно, что строка может вместить до 255 символов. Объявим массив из 255 символов и сравним его размер с размером строки. Напомню, что функция **SizeOf** возвращает размер памяти, занимаемой переменной.

```
var   S1 : array [1..255] of CHAR;  { это массив из 255 символов }
      S2 : String;                  { это строка длиной 255 символов }
begin
      Writeln (SizeOf(S1));   { печатает 255 }
      Writeln (SizeOf(S2));   { печатает 256 }
      Readln;
end.
```

Запустили программку? И что? Странно, но размер строки **S2** оказался равным 256 байтам, что на единицу больше размера массива. Почему? Где прячется ещё один байтик? Ответ представлен на рис. 98.

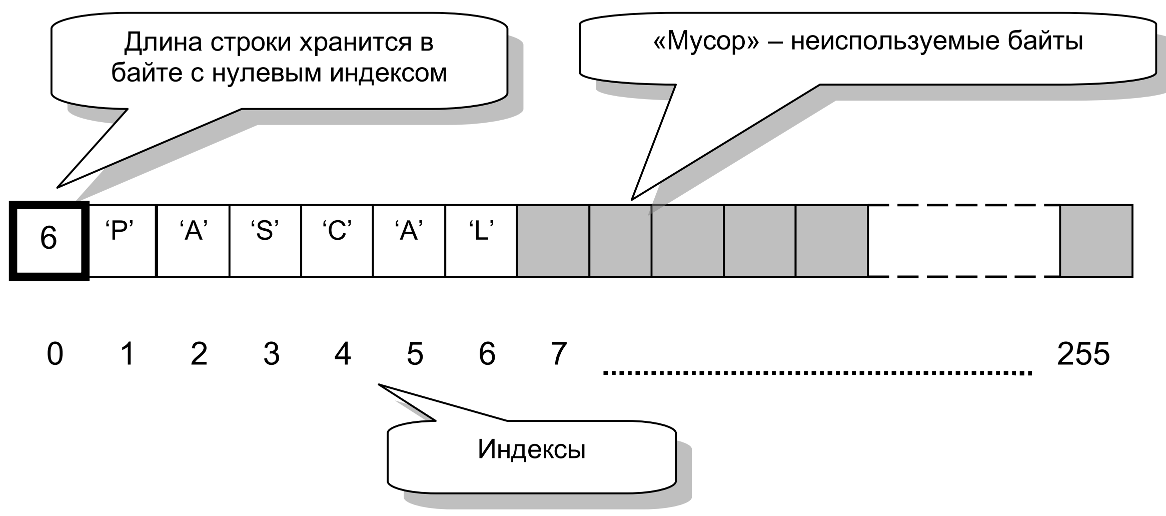


Рис. 98 – Размещение слова «PASCAL» в строковой переменной

Здесь показана внутренность строковой переменной со словом «PASCAL». Байты с 1-го по 6-й содержат буквы этого слова, а остальные байты не заняты. Но в

начале массива обнаружен ещё один байт — с нулевым индексом. Он содержит число 6 — это длина слова «PASCAL». Значит, строковый тип — это массив со скрытым нулевым байтом, хранящим фактическую длину строки (эту длину возвращает функция **Length)**.

Укороченные строки

Память — жилище переменных — всегда чем-нибудь занята. Даже пустая строка (нулевой длины) занимает 256 байтов памяти, и содержит что либо. Это «что либо» программисты называют мусором, а мусор никому не интересен. Так разумно ли отводить 256 байтов для строки, если большая её часть забита всяким вздором? Ведь память — ценный ресурс, и профессионал бережет её. К примеру, для строки, хранящей фамилию, вполне хватило бы и 20 байтов.

Это понимали и создатели Паскаля, они позаботились об экономии памяти. Строковые типы можно объявлять с указанием длины. Для этого после слова **STRING** в квадратных скобках указывают нужный размер строки, например:

```
type  TStrA = string[11];     { строка для 11 символов }
      TStrB = string[31];     { строка для 31 символа }
var   A : TStrA;        B : TStrB;
```

Здесь объявлены два строковых типа данных; первый из них вмещает до 11 символов, а второй — до 31. Соответственно переменная **A** будет занимать в памяти 12 байтов, а переменная **B** — 32 байта (с учетом нулевого байта). Согласитесь, — экономия солидная, особенно для массива из таких строк. Во всем остальном, кроме размера, короткие строки ничем не отличаются от переменных типа **STRING**.

А что случится при копировании длинной строки в короткую? А ничего, — не вместившиеся символы будут «отрублены». Следующая ниже программа P_44_1 подтверждает это, испытайте её.

```
{ P_44_1    - укороченные строки }
var   S1 : string;        { размер строки по умолчанию = 255 }
      S2 : string[5];     { размер укороченной строки = 5 символов }
begin
  S1:='abc';      S2:='abcdefgh';
  Writeln('Строка S1:  Размер =', SizeOf(S1):4,'  Длина = ',
           Length(S1):4,'  Значение= '+S1);
  Writeln('Строка S2:  Размер =', SizeOf(S2):4,'  Длина = ',
           Length(S2):4,'  Значение= '+S2);
  Writeln('Нулевой байт строки S1 = ', Byte(S1[0]));
  Writeln('Нулевой байт строки S2 = ', Byte(S2[0]));
  Readln;
end.
```

Операции со строками

Итак, уяснив внутреннее устройство строк, обратимся к связанным с ними операциям. Что мы умеем делать со строками сейчас? А вот что:

- вводить и выводить строки процедурами ввода и вывода;
- объединять несколько строк в одну (складывать);
- определять длину строки функцией **`Length`**;
- проверять строки на равенство и неравенство;
- обращаться к отдельным символам строки (доступ по индексу).

Учитывая важность строкового типа, разработчики Паскаля предусмотрели для строк ещё несколько процедур и функций, позволяющих:

- искать одну строку внутри другой;
- копировать часть строки в другую строку;
- вставлять одну строку внутрь другой;
- удалять часть символов из строки;
- сравнивать две строки в смысле алфавитного порядка.

Рассмотрим всё это подробней. Представленные далее объявления процедур и функций даны мною лишь для пояснений, их не надо вставлять в программы.

Поиск в строке (Pos)

Функция **`Pos`** ищет одну строку внутри другой, её объявление выглядит так.

```
function Pos(SubS: string; S: string): Integer;
```

Функция принимает два параметра:

- **SubS** – подстрока, которую ищут (то есть фрагмент строки);
- **S** – строка, в которой ищут.

Если искомый фрагмент **SubS** найден, функция возвращает его позицию — индекс первого символа **SubS** внутри строки **S**, а иначе возвращает ноль. Если строка **S** содержит несколько искомых фрагментов, возвращается индекс первого из них. Вот примеры.

```
S:= 'BORLAND PASCAL';
p:= Pos('LA', S);      { 4 }
p:= Pos('PAS', S);     { 9 }
p:= Pos('pas', S);     { 0 – подстрока не найдена }
p:= Pos('A', S);       { 5 – первая из трех букв "A" }
```

Искомым фрагментом может быть и отдельный символ. Поиск ведется с учетом регистра; это значит, что заглавная и строчная буквы «P» считаются разными буквами.

Копирование части строки (Copy)

Функция **Copy** возвращает часть заданной строки.

```
function Copy(S: string; Index, Count: Integer): string;
```

Входных параметров три:

- **S** – строка, из которой копируются символы;
- **Index** – индекс первого копируемого символа;
- **Count** – количество копируемых символов.

А вот примеры её применения.

```
S:= 'Free Pascal forever!';
T:= Copy(S, 6, 6);     { 'Pascal' }
T:= Copy(S, 6, 255);   { 'Pascal forever!' }
```

Если копируемых символов затребовано больше, чем содержится в исходной строке, то скопируются все символы до конца строки (как в последнем примере).

Вставка в строку (Insert)

Объединять строки сложением просто. А если надо вставить строку в середину другой? Тогда обратитесь к процедуре **Insert**.

```
procedure Insert(S1: string; var S2: string; Index: Integer);
```

Входные параметры:

- **S1** – вставляемая строка;
- **S2** – ссылка на принимающую строку;
- **Index** – позиция вставки.

Вот один пример:

```
S:='Спартакчемпион!';
{ В позицию 8 вставляются три символа: тире и два пробела }
Insert(' – ', S, 8);          { Спартак – чемпион! }
```

Если позиция вставки превышает длину строки **S2**, то строка **S1** добавится в конец **S2**. Если длина итоговой строки **S2** превысит допустимый размер, лишние символы будут отброшены.

Удаление символов из строки (Delete)

Говорят: ломать — не строить. Попытайтесь, однако, удалить часть символов из строки. Слабо? А процедура **Delete** справляется с этим играючи.

```
procedure Delete(var S: string; Index, Count : Integer);
```

Параметры таковы:

- **S** – ссылка на строку;
- **Index** – индекс первого удаляемого символа;
- **Count** – количество удаляемых символов.

Вот случай применения процедуры.

```
S:= 'Free Pascal forever!';
Delete(S, 6, 7);        { 'Free forever!' }
```

Сравнение строк

Мы уже сравнивали строки на равенство (вспомните проверку пароля). Но строки сравнивают и на больше–меньше — лексикографически. При этом сравниваются слева направо коды символов двух строк в смысле их алфавитного порядка. Если длины строк разные и короткая совпадает с началом длинной, то большей считается длинная строка. Вот примеры:

```
Writeln ('Borland' > 'Pascal'); { false, поскольку 'B' < 'P'  }
Writeln ('ABC' > 'AB');       {true, поскольку 'ABC' длиннее 'AB' }
Writeln ('ABC' > 'abc');      { false, поскольку 'A' < 'a'  }
Writeln ('45' > '1000');      { true, поскольку '4' > '1' }
```

В первом примере код буквы «B» меньше кода буквы «P», поэтому левая строка меньше правой. Во втором случае первые символы совпадают, но левая строка длиннее, а значит больше. В третьем примере левая строка меньше, — тоже в соответствии с таблицей кодировки. Обратите внимание на неожиданный результат сравнения строк, составленных из цифр, — это вам не числа!

Сравнивая строки, можно отсортировать их в лексикографическом порядке (как если бы они располагались в словаре). К сожалению, такое сравнение работает только для латинских букв, для русских оно не всегда верно, приходится изобретать свою функцию сравнения (в DELPHI этой проблемы нет).

Перевод символов в верхний регистр (UpCase)

Функция **UpCase** меняет код латинской буквы, переводя её из нижнего в верхний регистр. Иными словами, она превращает строчную (маленькую) латинскую букву в заглавную (большую). Объявление функции таково.

```
function UpCase(Ch: Char): Char;
```

Входной параметр — символ, а возвращается почти тот же символ, только «подросший», вот примеры.

```
c:= UpCase('r');        { 'R' }
c:= 'n';
c:= UpCase( c );        { 'N' }
```

Подсунув этой функции большую латинскую букву, цифру или знак препинания, вы получите назад свой символ неизменным. То же будет и с русскими буквами — они не обрабатываются функцией **UpCase**.

```
     c:= UpCase('R');        { 'R' }
     c:= UpCase('8');        { '8' }
     c:= UpCase('ы');        { 'ы' }
```

Функцией **UpCase** обычно приводят введенные строки к определенному виду. Ведь пользователь может ввести данные как заглавными, так и строчными буквами, а это иногда мешает правильной обработке строки.

Ознакомившись со строковой теорией, применим её, что называется, «в бою».

Подсчет слов в строке

Вот вам строка, посчитайте в ней количество слов «Pascal». Чуть подумав, вы остановитесь на функции **Pos**, — ведь она возвращает позицию искомого слова. Но функция обнаруживает лишь первое вхождение фрагмента, а как быть с остальными? Я предлагаю постепенно разрушать исходную строку. То есть, найдя искомый фрагмент, будем удалять его из строки и снова повторять поиск. На этом и построена программа P_44_2.

```
{ P_44_2 -   Подсчет слов «PASCAL» в строке }
var   S : string;   { исходная строка }
      p : integer;  { позиция в строке }
      c : integer;  { счетчик слов }
begin
  S:='Лучший язык программирования - это PASCAL!'+
      'Изучите PASCAL! PASCAL не подведет!';
  c:=0;
  repeat
      p:= Pos('PASCAL', S);   { ищем слово «PASCAL» }
      if p>0 then begin       { если нашли }
            Inc(c);           { то наращиваем счетчик }
            { и удаляем это слово из строки }
            Delete(S, p, Length('PASCAL'));
      end
  until p=0;      { выход, если слов «PASCAL» больше нет }
  Writeln('Найдено слов PASCAL: ',c);  Readln;
end.
```

Контекстная замена

Любой текстовый редактор умеет заменять одну подстроку на другую, — это называется контекстной заменой. Устроим такую замену в строковой переменной. Итак, дана строка, содержащая несколько слов «Pascal». Заменим все вхождения слова «Pascal» словом «Паскаль» (чем не англо-русский переводчик?).

Разобравшись с предыдущей задачей, вы легко одолеете и эту. Для проверки вашего решения сравните его с моим (P_44_3).

```
{ P_44_3 -  Замена слов «Pascal» на «Паскаль» }
var   S : string;   { исходная строка }
      p : integer;  { позиция в строке }
begin
  S:='Лучший язык программирования – Pascal! '+
      'Изучите Pascal! Pascal не подведет!';
  Writeln(S); { исходная строка }
  repeat
      p:= Pos('Pascal', S);        { ищем слово 'Pascal' }
      if p>0 then begin       { если нашли }
            { удаляем это слово из строки }
            Delete(S, p, Length('Pascal'));
            { и вставляем в этом месте слово 'Паскаль'}
            Insert('Паскаль', S, p);
      end
  until p=0;  { выход, если слов 'Pascal' больше нет }
  Writeln(S); { строка результата }
  Readln;
end.
```

Итоги

- Строка родственна массиву символов. Дополнительный нулевой элемент этого массива содержит длину строки.
- Строка, объявленная без указания размера, по умолчанию занимает 256 байтов памяти и может содержать до 255 символов.
- Для экономии памяти используют строки меньшего размера. При объявлении таких строк размер указывают внутри квадратных скобок после слова **STRING**.
- В Паскале предусмотрен ряд встроенных процедур и функций, облегчающих обработку строк.

А слабо?

А) Напишите процедуру, переводящую все символы строки (латинские буквы) к верхнему регистру.

Б) Напишите функцию для приведения любой буквы к верхнему регистру (включая и русские). Подсказка: вспомните о таблице кодировки.

В) Напишите функцию для приведения любой буквы к нижнему регистру.

Г) Напишите собственные процедуры и функции обработки строк, повторяющие те, что встроены в Паскаль. Дайте им названия, похожие на стандартные, например: **`MyCopy`**, **`MyDelete`** и так далее.

Д) Вращение строки вправо. Напишите процедуру, перемещающую 1-й символ строки на место 2-го, 2-й — на место 3-го и т.д. Последний символ должен занять 1-е место. Примените средства обработки строк.

Е) Вращение строки влево. Напишите процедуру для перемещения 2-го символа на место 1-го, 3-го — на место 2-го и т.д. Первый символ должен стать последним.

Ж) Строка содержит несколько слов — предложение. Напишите программы для решения следующих задач.

- Напечатать в столбик отдельные слова введённого предложения.
- Определить количество слов в строке.
- Равномерно расставить пробелы между словами так, чтобы удлинить строку до 80 символов (исходная строка короче 80).

З) Напишите булеву функцию, определяющую, является ли строка (параметр) палиндромом. Палиндром читается одинаково в обоих направлениях.

И) Напишите булеву функцию, определяющую, можно ли из букв первого слова составить второе (например, «клавиша» и «вилка» — **`TRUE`**). Учитывается только набор букв, а не их количество. Подсказка: примените множества.

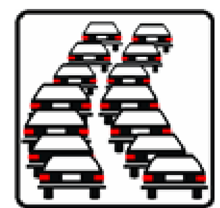
К) Дана строка, содержащая не менее трёх символов. Найти в ней три стоящих подряд символа, дающих максимальную сумму своих кодов.

Л) В строке найти возрастающую последовательность символов наибольшей длины (сравнивайте коды символов).

М) Напишите булеву функцию, проверяющую, следуют ли символы строки по неубыванию своих кодов.

Н) Напишите функцию для шифрования строки путём перестановки её символов, расположенных на нечётных позициях: первый символ обменивается с последним, третий — с третьим от конца и т.д.

Глава 45
Очереди и стеки



Хорошей вещи найдется уйма применений. Взять хотя бы строки, из которых мы построим сейчас инструменты системных программистов — очереди и стеки.

Для меня системные программисты — это боги, ступившие на землю (не путайте их с системными администраторами!). Операционные системы, компиляторы, вирусы и антивирусы — это их мозгов дело. Настоящего системщика я видел лишь разок, да и то издали. Таинственное системное программирование! — дерзнем ли коснуться его? Почему нет? Надо же когда-то начинать! Ведь стеки и очереди отчасти вам знакомы.

Вовочка в потоке событий

Переживайте неприятности по мере их поступления — эта житейская мудрость касается любого из нас. Жизнь — это поток событий, барахтаясь в котором, мы поминутно решаем: прервать ли начатое дело и хвататься за другую проблему? Или отложить эту новую проблему «на потом»?

Смотрите: вот компьютер и Вовочка, мирно играющий в «звездные войны». Входит мама: «Вова, ты уроки сделал?». Вова неохотно откладывает игру и приступает к урокам. Через полчаса мама снова беспокоит его: «Вовочка, я не могу отойти от плиты. Сгоняй-ка быстро за луком». Уроки откладываются, и Вова отправляется в магазин. По пути он видит гоняющих мяч приятелей, — им не хватает вратаря. Как не помочь друзьям? Поход в магазин откладывается, и Вовочка на страже ворот. Делу время, а потехе час. Закончилась игра, и можно выполнить мамино поручение. Купив луку, Вова доделывает отложенные уроки и возвращается к первому занятию — «звездным войнам».

Вовочка обрабатывал события по принципу стека, — скажет об этом системный программист. Другое название стека — дисциплина обслуживания LIFO, что значит «Last-In, First-Out», или по-русски: «последним пришел — первым ушел». Мы соприкоснулись со стеком, разбирая рекурсивную процедуру быстрой сортировки. Помните пример с укладкой рюкзака? А вот ещё пример: магазин для патронов. Патрон, вставленный в магазин последним, выстрелит первым, и наоборот.

Итак, пример с Вовочкой показал, что важные события мы обслуживаем по принципу стека.

А очередь? Какое отношение к нам имеет этот символ потерянного времени? Заметив, как «тормозит» порой ваш компьютер, вспомните о ней. Иногда компьютер реагирует на мышку и клавиатуру с заметным опозданием, но не забывает о нажатиях и щелчках. Они накапливаются в очереди, а затем извлекаются оттуда и обрабатываются.

Или другой пример — печатание документов на сетевом принтере. Когда принтер занят, операционная система ставит запросы на печать в очередь. А затем — по мере готовности принтера — выбирает файлы из этой очереди и отправляет принтеру.

За очередью тоже закрепилось свое сокращение — FIFO, что значит «First-In, First-Out» — «первым пришел, первым ушел». В отличие от стека, в очередь попадают маловажные события.

Обратите внимание, что элементами очередей и стеков может быть что угодно: события, предметы и даже люди. Рассмотрим два примера: запись в танцевальный кружок и сортировочную горку.

Танцевальный кружок

В городе набирали детей в кружок бального танца. Запись в кружок вела преподающая в нём дама. Приходящих мальчишек и девчонок она стремилась объединять в пары при первой возможности с тем, чтобы они сразу приступали к занятиям. Однако на запись дети приходили поодиночке в разное время и в случайном порядке, — то несколько мальчиков подряд, то несколько девочек. Дама не слыхала о системном программировании, а потому прибегла к здравому смыслу. Взяв пару записных книжек, она поступала так.

Если к ней являлось больше мальчиков, дама заносила их в мальчишечью записную книжку, то есть, ставила в очередь. Приняв девочку, она выбирала из этой очереди первого мальчика и составляла пару. Если же являлось больше девочек, то дама ставила их в девчоночью очередь (в другой записной книжке), а с приходом очередного мальчика составляла новую пару. Так, в порядке поступления, составлялись пары мальчиков и девочек.

Пусть наша новая программа повторит действия танцевального тренера, — инженеры называют это моделированием. Работать будем, конечно, не с живыми детьми, мы представим их как-то иначе. Условимся обозначать их латинскими буквами: девочек — строчными, а мальчиков — заглавными (только потому, что они выше ростом).

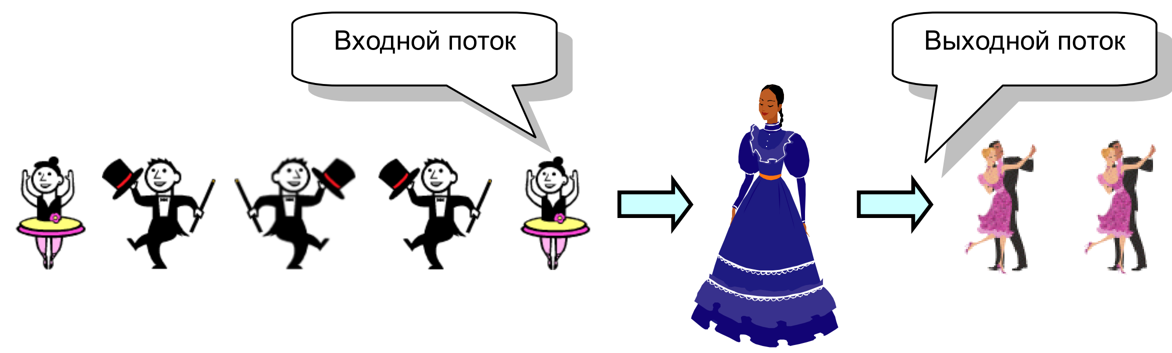


Рис. 99 – Запись в танцевальный кружок

Теперь станьте на место учителя: к вам приходят то мальчик, то девочка, но вы не знаете, кто будет следующим, — это поток, и в нём доступен только первый его элемент. Организовать входной поток можно посимвольным вводом «мальчиков» и «девочек». Но мы сделаем ещё проще: представим поток детишек строкой, и будем считать, что к преподавателю они являются слева направо по одному. Например, строка

```
ZHJKqwertASDyuiopQWERTYUIOPasdf
```

означает, что первым явится мальчик по имени **Z**, а последней — девочка по имени **f**. Первая пара составится из мальчика **Z** и девочки **q**. Это упрощение не меняет сути нашей модели, но избавляет её от второстепенных деталей.

Итак, с учетом всех договоренностей, явим задачу в окончательном виде. Дана строка, состоящая из заглавных и строчных букв латинского алфавита — «мальчиков» и «девочек». Мы должны сформировать другую строку, состоящую из тех же символов, но следующих попарно: сначала заглавная буква — «мальчик», затем строчная — «девочка». Пары разделяются пробелом. Например, для указанной выше строки, пары должны быть составлены так.

```
Zq Hw Je Kr At Sy Du Qi Wo Ep Ra Ts Yd Uf
```

А напоследок программа должна напечатать имена тех, кто временно остался без пары. Здесь это будут пришедшие в числе последних мальчики **I**, **O** и **P**.

Если логика программы вам ясна, разрешим теперь главный вопрос: как организовать очередь символов? Ведь очередь — это не просто массив данных, а механизм, содержащий и хранилище данных и процедуры для работы с ними.

Сделаем так. Элементы очереди — символы — будем хранить в строковых переменных. К ним добавим ещё две процедуры: одну — для установки элемента в очередь, другую (это будет функция) — для извлечения из очереди первого элемента. Назовем их соответственно **PutInQue** — «поставить в очередь» и **GetFromQue** — «извлечь из очереди» (Queue — «очередь» или «хвост»). Всё это представлено в программе P_45_1.

```
{ P_45_1    - Запись в танцевальный кружок }

{ Постановка символа arg в очередь Que }

procedure PutInQue(var Que: string; arg: char);
begin
  Que:= Que + arg;   { добавляем в конец строки }
end;

{ Выбор из очереди Que элемента в параметр arg }

function GetFromQue(var Que: string; var arg: char): boolean;
begin
  if Length(Que) = 0               { если очередь пуста }
      then GetFromQue:= false
      else begin
            GetFromQue:= true;     { если не пуста }
            arg:= Que[1];          { запоминаем первый элемент }
            Delete (Que, 1, 1);    { и удаляем его из очереди }
            end
end;

      { Глобальные переменные }
var   S_IN : string;    { входной поток - символы }
      S_OUT : string;   { выходной поток (пары) }
      Boys  : string;   { очередь мальчиков }
      Girls : string;   { очередь девочек }
      c1,c2 : char;     { очередная пара - символы строки }
      i : integer;      { индекс во входном потоке }

begin {--- Главная программа ---}

  { задаем (вводим) входной поток: A..Z - мальчики, a..z - девочки }
  S_IN:='ZHJKqwertASDyuiopQWERTYUIOPasdf';
  S_OUT:='';              { выходной поток пока пуст }
  Boys:='';  Girls:=''; { Очищаем очереди мальчиков и девочек }
```

```
  { Цикл обработки входного потока }
  for i:=1 to Length(S_IN) do begin
      c1:= S_IN[i];     { выбираем из входного потока }
      if c1 in ['A'..'Z']
            then begin { если это мальчик...}
                  { если в очереди есть девочка }
                  if GetFromQue(Girls, c2)
                        { добавляем пару в выходной поток }
                        then S_OUT:= S_OUT+c1+c2+' '
                        { а иначе помещаем мальчика в очередь }
                        else PutInQue(Boys, c1);
            end
            else begin { а если это девочка...}
                  { если в очереди есть мальчик }
                  if GetFromQue(Boys, c2)
                        { добавляем пару в выходной поток }
                        then S_OUT:= S_OUT+c2+c1+' '
                        { а иначе помещаем девочку в очередь }
                        else PutInQue(Girls, c1);
            end
  end;
  Writeln('Входной поток:' );
  Writeln(S_IN);
  Writeln('Выходной поток:' );
  Writeln(S_OUT);

  if Length(Boys)>0 then begin
      Writeln('В очереди мальчиков остались:' );
      Writeln(Boys);
  end;

  if Length(Girls)>0 then begin
      Writeln('В очереди девочек остались:' );
      Writeln(Girls);
  end;
  Readln;
end.
```

Процедура **PutInQue** просто добавляет символ в конец строки. Строго говоря, если длина строки достигнет 255, то новый символ не попадет в очередь. Но мы не станем усложнять программу дополнительными проверками, — считаем, что емкости очереди нам достаточно.

Но для функции **GetFromQue**, выбирающей из очереди первый символ, контроль строки на пустоту необходим, иначе работа модели нарушится. Функция возвращает состояние очереди, бывшее до извлечения символа (**TRUE**, если очередь не была пуста). А сам извлекаемый символ возвращается через параметр **arg**, — это ссылка на символьную переменную. Вот, пожалуй, и вся премудрость. Испытайте эту программу. Добавьте операторы печати для наблюдения за очередями.

Скитания товарного вагона

Прежде, чем углубиться в стек, вникнем в работу железной дороги. Вы знаете, как железнодорожники доставляют товарный вагон из пункта «А» в пункт «Б»? «Очень просто, — скажете, — цепляют к составу и тащат!» Тогда взгляните на рис. 100.

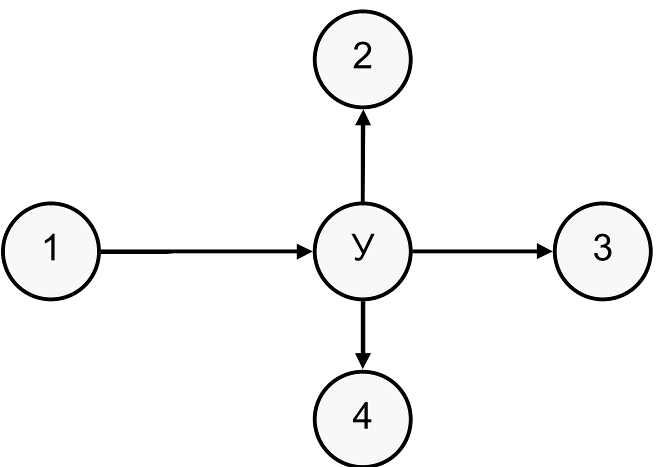


Рис. 100 – Доставка вагонов между несколькими станциями

Здесь показаны пять железнодорожных станций, четыре из которых обозначены цифрами, а пятая — узловая станция — буквой «У». Предположим, что со станции «1» надо доставить несколько десятков вагонов на другие станции (по направлению стрелок). С этих станций тоже везут вагоны, но соответствующие стрелки я не показал. Тащить вагоны поодиночке разорительно! Поэтому их собирают в составы по нескольку десятков вагонов. Накопив такой состав на станции «1», железнодорожники доставляют его на узловую станцию; сюда же стекаются составы с других направлений. На узловой творится самое интересное, — здесь из одних составов формируют другие с тем, чтобы тащить их далее в нужном направлении. Эта работа называется сортировкой состава. В нашей стране сотни товарных станций, многие из которых узловые. Прежде чем попасть по назначению, вагон кочует между узловыми станциями, проходя через несколько сортировок. А вы говорите: просто, просто!

Но это ещё присказка, — сказка впереди.

Сортировочная горка

Итак, на узловых станциях формируют новые составы с тем, чтобы каждый вагон следовал далее в нужном направлении. Для сортировки устроены так называемые сортировочные горки. Горка — это слегка наклоненный участок пути; если отцепить от стоящего на нём состава вагон, последний покатится под горку. Но укатится недалеко, — под горкой устроено несколько тупиков. Тупик — это обычное состояние программиста, но здесь я говорю о других тупиках, железнодорожных. Это участки пути, ограниченные с одной стороны земляным валом. Уткнувшись в этот вал, вагон остановится. Горка соединяется с тупиками железнодорожными стрелками, переключая которые можно направить катящийся с горки вагон в тот или иной тупик (рис. 101).

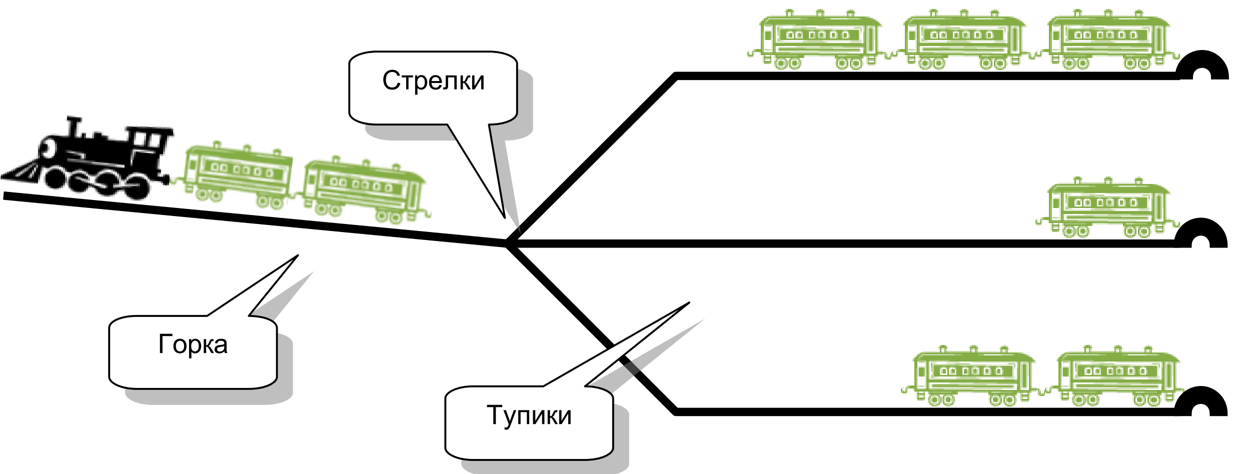


Рис. 101 – Схема сортировочной горки

Как сортируют состав? Основную работу выполняют двое: сцепщик и стрелочник (который всегда виноват!). От стоящего на горке состава сцепщик отсоединяет по очереди вагон за вагоном и сообщает стрелочнику по рации, в который из тупиков направить очередной вагон, — тому остаётся лишь переводить свои стрелки. Вкатившись в тупик, вагон тормозится земляным валом или слегка соударяется с уже стоящим там вагоном и автоматически сцепляется с ним. «Разбросав» по тупикам один состав, на горку выкатывают другой и продолжают сортировку, пока в тупиках не сформируются новые составы, готовые к дальнейшему пути.

Наша цель: смоделировать сортировочную горку, то есть создать программу, ведущую себя подобно такой горке. Вагоны заменим символами; следовательно, и состав и стоящие в тупиках вагоны мы представим строками. Будем считать первый символ строки первым вагоном состава, — он прицеплен к локомотиву. В тупике первым будет вагон, стоящий у земляного вала. Легко догадаться, что обрабатывать вагоны будем по принципу стека, поскольку сцепщику всегда доступен только последний вагон.

Договорившись об этом, сформулируем задачу окончательно. Дана строка символов (состав), из которой надо сформировать три других. Вагоны, обозначенные большими буквами **'A'…'Z'**, отправим на станцию «A»; другие, обозначенные маленькими буквами **'a'…'z'**, — поедут к станции «B», а третьи, обозначенные цифрами **'0'…'9'**, — к станции «C». Программа должна сформировать три строки — это вновь собранные составы. Первый символ в них, — это вагон, прицепленный непосредственно к локомотиву.

Для решения задачи надо всего лишь в точности повторить действия сцепщика и стрелочника. Будем «отцеплять» символы от строки и «заталкивать» их в стеки — это наши тупики. Значит, надо построить механизм для стеков. Он будет похож на механизм для очереди: элементы храним в строковых переменных, а для занесения и извлечения элементов из стека учредим две процедуры. По традиции программисты называют эти процедуры так: **Push** — затолкнуть в стек, и **Pop** — вытолкнуть из стека.

Процедура заталкивания в стек **Push** присоединяет символ к концу строки. Она точь-в-точь повторяет процедуру установки в очередь, только называется иначе.

А функция выталкивания из стека **Pop** возвращает последний символ строки, одновременно удаляя его оттуда. Если же стек окажется пуст, функция сообщит об этом. Сходство с функцией извлечения из очереди очевидно, разница лишь в позиции извлекаемого символа: для очереди это первый символ, а для стека — последний.

Теперь вам не составит труда разобраться в показанной ниже программе P_45_2. Обратите внимание на отцепку вагонов от исходного состава: она тоже выполняется функцией выталкивания **Pop**, поскольку исходный состав трактуется как непустой стек.

```
{ P_45_2  - Сортировочная станция }

{ Помещение элемента в стек }
procedure Push(var aStack: string; arg: char);
begin
      aStack:= aStack + arg;  { добавляем в конец строки }
end;

{ Извлечение элемента из стека }
function Pop(var aStack: string; var arg: char): boolean;
begin
  if Length(aStack) = 0    { если стек пуст }
      then  Pop:= false       { сообщаем об этом }
      else  begin
            { возвращаем последний элемент }
            arg:= aStack[Length(aStack)];
            { и удаляем его из стека }
            Delete(aStack, Length(aStack), 1);
            Pop:= true; { признак того, что стек не был пуст }
            end;
end;

var   S : string;              { исходный состав }
      SA, SB, SC : string;     { три сортировочных тупика A,B,C}
      c : char;                { очередной вагон }
begin
  S:= 'HEjd31kDJK62px912se3BKdwL9';  { Исходный состав }
  Writeln('Исходный состав:  '+S);
  SA:=''; SB:=''; SC:='';  { очистка тупиков }
  { Отцепляем вагоны от исходного состава и заталкиваем в тупики }
  while Pop(S, c) do begin
      if c in ['A'..'Z'] then Push(SA, c);
      if c in ['a'..'z'] then Push(SB, c);
      if c in ['0'..'9'] then Push(SC, c);
  end;
  { Теперь исходный состав пуст, то есть S='' }
  { Выкатываем вагоны из тупика A и цепляем к первому составу }
  while Pop(SA, c) do Push(S, c);
  Writeln('На станцию A   :  '+S);
  S:='';  { Освобождаем пути }
  { Выкатываем вагоны из тупика B и цепляем ко второму составу }
```

```
  while Pop(SB, c) do Push(S, c);
  Writeln('На станцию B   :  '+S);
  S:='';  { Освобождаем пути }
  { Выкатываем вагоны из тупика C и цепляем к третьему составу }
  while Pop(SC, c) do Push(S, c);
  Writeln('На станцию C   :  '+S);
  Readln;
end.
```

Вы познакомились с механизмами очередей и стеков. Мы построили их на основе символьных строк, но в системном программировании это делается иначе, и скоро вы узнаете об этом больше.

Итоги

- Очереди и стеки – это механизмы, применяемые в системных программах. Элементами очередей и стеков могут быть любые объекты.
- Очередь обслуживает элементы по принципу «первый пришел – первый ушел» или сокращенно FIFO (First-In, First-Out).
- Стек обслуживает элементы по принципу «последний пришел – первый ушел» или сокращенно LIFO (Last-In, First-Out).

А слабо?

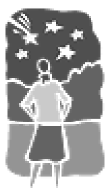
А) Исследуя модель танцевального кружка, можно заметить, что в любой момент одна из двух очередей обязательно пуста. В самом деле, если приходит больше мальчиков, то будет пуста девчоночья очередь и наоборот. Можно ли обойтись одной очередью? Придумайте, как это сделать.

Подсказка: добавьте функцию для тестирования очереди с тем, чтобы выяснить, не пуста ли она. И, если не пуста, то кто томится в ней — мальчик или девочка? Эта функция не должна изменять состояние очереди.

Б) На реальных станциях на горку последовательно загоняют несколько составов, а уж потом освобождают тупики. Добавьте в модель сортировочной горки возможность такой обработки. Исходные составы (строки) должны вводиться с клавиатуры, признак окончания ввода — пустая строка. Совет: выделите действия по сортировке одного состава в отдельную процедуру.

В) Постройте механизмы очереди и стека на базе массива символов, а не на базе строки. Какие дополнительные переменные здесь понадобятся?

Глава 46
Огромные числа



Давно минули времена, когда для счета всем хватало своих пальцев — первого «калькулятора» человечества, а наши потребности в счете всё растут и растут...

Сколько звезд на небе?

Некий король — любитель науки, порядка и немножко чудак — распорядился пересчитать всё, что ни есть, в своих владениях. Ладно бы, его интересовали крупные предметы вроде домов, машин и всего такого… Но нет, король велел пересчитать и капли в морях, и песчинки на берегах, и травинки в степях! Пока ученые придумывали способы подсчета, программисты возились с другой задачей — обработкой всего этого на компьютере. Они искали подходящий тип данных для хранения тех огромных чисел, что нужны ученым! Обратившись к описанию языка Паскаль, программисты заинтересовались двумя числовыми типами, которые на первый взгляд подходили для такого случая.

Первый из них — тип **LongInt** — длинное целое число. Наибольшее значение для этого типа чисел составляет **2'147'483'647**, то есть несколько больше двух миллиардов. Маловато будет, — рассудили мудрецы, и продолжили поиск.

Вскоре они наткнулись на другой тип чисел — **Extended**, который мог вмещать сказочные значения — вплоть до 10^{4932} ! Иначе говоря, эти числа могли содержать почти пять тысяч цифр! Но радость математиков была недолгой; рассмотрев тип **Extended** пристальней, они отвергли его. Оказалось, что из этих пяти тысяч цифр только первые двадцать — точные (их называют значащими), а остальные не внушают доверия, и могут быть замены чем угодно, хоть нулями.

Есть ли толк от этих неточных чисел? Есть. Дело в том, что в инженерных и научных расчетах этой точности вполне хватает. Но здесь был иной случай, — требовался абсолютно точный подсчет, государь не терпел огрехов. «Точность — вежливость королей!» — говаривал он. И программисты ткнулись, как обычно, в тупик.

Сложение «в столбик» никто не отменял

Выход из тупика нашелся случайно. Один из королевских программистов помогал сынишке справиться со школьными уроками — складывали числа «в столбик». Тут его и осенило: почему бы и нам, — смекнул папаша, — не складывать числа тем же способом? Так тряхнем стариной и вспомним это сложение «в столбик»? Рис. 102 освежит вашу память.

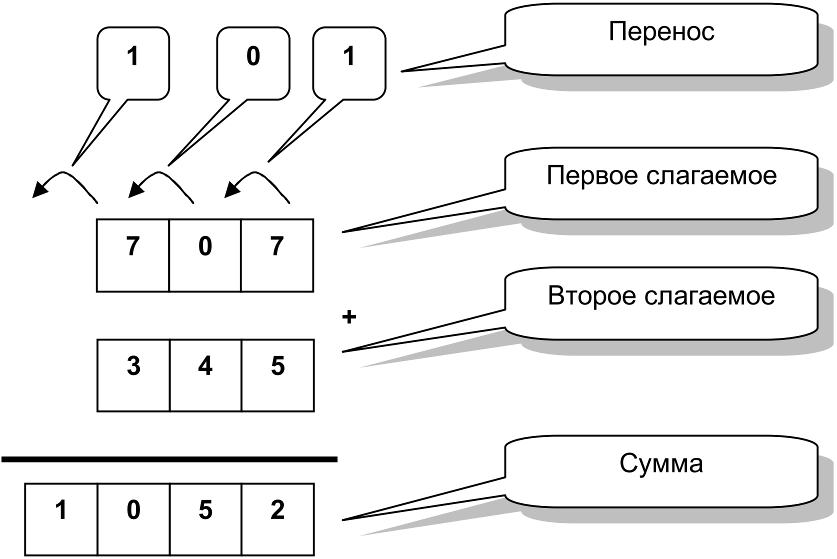


Рис. 102 – Пример сложения в столбик

Итак, сложение чисел начинаем с младшего, то есть крайнего правого разряда. Если сумма двух цифр превысит девять, то в разряд результата записываем остаток от деления этой суммы на 10 (то есть цифры от 0 до 9), а к следующему разряду добавляем перенос.

Если обозначить складываемые цифры буквами **A** и **B**, то алгоритм сложения «в столбик» для одного разряда с учетом предыдущего переноса запишется так.

```
цифра := (A + B + перенос) mod 10
перенос := (A + B + перенос) div 10
```

Напомню, что операция **MOD** вычисляет остаток от деления одного целого числа на другое, а операция **DIV** — частное с отбрасыванием остатка. Перед сложением самого младшего разряда перенос берётся равным нулю. Обратите внимание, что условный оператор здесь излишен.

Великая стройка

Уловив основную идею — действия «в столбик», — программисты задумались над хранением цифр огромного числа. Они рассмотрели два равно подходящих средства: либо массив байтов, каждый из которых будет содержать числа от 0 до 9, либо массив символов «0»…«9». Ученые остановились на символах, но от строки отказались, поскольку строка вмещает лишь 255 символов, а им требовалось больше.

В итоге объявление сверхбольшого числа получилось таким, как показано в программе P_46_1, — она была написана для отладки процедуры распечатки сверхбольшого числа.

```
{ P_46_1 – Распечатка сверхбольших чисел }
      { объявления для сверхбольшого числа }
const CSize = 500;  { размер массива для цифр }
type  TBigNumber = array [1..CSize] of char;
var   BN : TBigNumber; { очень большое число! }
      { Процедура распечатки сверхбольшого числа
      Младшие цифры числа располагаются в младших элементах массива.
      Но распечатывать надо, начиная со старших цифр.
      Поэтому обработку массива ведем от конца к началу.
      При этом старшие позиции, заполненные пробелами, не печатаем.}
procedure WriteBigNumber(var F: text; const aNum: TBigNumber);
var i : integer;
begin
  i:= SizeOf(aNum); { печать начинаем со старших цифр }
  { Пока встречаются незначащие цифры, пропускаем их }
  while (i>0) and not (aNum[i] in ['1'..'9']) do Dec(i);
  { Если весь массив заполнен пробелами, то печатаем ноль }
  if i=0 then Write(F, '0');
  { Теперь печатаем оставшиеся цифры }
  while i>0 do begin
      Write(F, aNum[i]);
      Dec(i);
  end;
  { Добавляем ещё одну пустую строчку для удобства созерцания }
  Writeln(F); Writeln(F);
end;
var   i : integer;
begin       { === Главная программа === }
  FillChar(BN, SizeOf(BN), ' ');  { заполняем пробелами }
  WriteBigNumber(Output, BN);
  FillChar(BN, SizeOf(BN), '7');  { заполняем семерками }
  WriteBigNumber(Output, BN);
  { заполняем случайными цифрами }
  for i:=1 to CSize-1 do BN[i]:= Char(Random(100) mod 10 + Ord('0'));
  WriteBigNumber(Output, BN);
  Readln;
end.
```

Итак, тип данных **TBigNumber** — это сверхбольшое число в виде массива из 500 цифр. Процедура **WriteBigNumber** — печать сверхбольшого числа — выполняет то, о чем говорит её название. Напомню, что примененная здесь процедура **Dec(i)** выполняет быстрое вычитание единицы.

В главной программе вы найдете процедуру **FillChar** — «заполнить символом». Для заполнения массива можно организовать цикл, но процедура **FillChar** делает это проще и быстрее, она объявлена в Паскале так.

```
procedure FillChar(var X; Count: Integer; Value: Byte);
```

Обратите внимание, что тип первого параметра **X** не указан, что крайне редко для Паскаля! По сути это ссылка на переменную любого типа. Второй параметр — **Count** — задает количество байтов, помещаемых в переменную **X**. Обычно значение **Count** совпадает с размером этой переменной и задается равным **SizeOf(X)**. И, наконец, третий параметр **Value** — «значение», тоже не совсем обычен. Его тип объявлен как байт (то есть число), но в действительности может принимать любой однобайтовый тип данных, например, символ или булево значение. Вот несколько примеров.

```
var   A : array 1..100 of char;
      B : array 1..200 of byte;
      C : array 1..50 of boolean;
. . .
      FillChar(A, SizeOf(A), '*');       { заполнение массива звездочками }
      FillChar(B, SizeOf(B), 0);         { заполнение массива нулем }
      FillChar(C, SizeOf(C), false);     { заполнение массива «ложью» }
```

Согласитесь, нелегко отказаться от применения столь удобной процедуры.

И последнее. В нашу процедуру **WriteBigNumber** передается ссылка на выходной файл, что придает ей универсальность. Вызывая её из главной программы, мы передаём туда файловую переменную **Output**, — это файл, связанный с экраном. Напомню, что файл **Output** не требует ни объявления, ни открытия, ни закрытия — он встроен в язык готовеньким. Существует и встроенный файл по имени **Input** — он служит для ввода данных с клавиатуры.

Длинная арифметика

Итак, испытав рассмотренную нами программу, королевские программисты сделали первый шаг к своей цели — освоили распечатку сверхбольших чисел. Теперь предстояло написать процедуру для сложения таких чисел, ей дали имя **AddNumbers** — «сложить числа». Она принимает два параметра — это ссылки на сверхбольшие числа, то есть на массивы. Работа процедуры основана на формулах сложения в столбик, причем младшей цифрой числа был выбран первый элемент массива.

Поскольку массив содержит символы **'0'…'9'**, а не числа **0…9**, при сложении символы преобразуем в числа и обратно (ведь символы складывать нельзя). Эти простые превращения выполняем по формулам.

```
      цифра := Ord (символ_цифры) - Ord ('0')
      символ_цифры := Char (Ord ('0') + цифра)
```

Вот эта чудесная программа целиком.

```
{ P_46_2 - Сложение сверхбольших чисел }

const CSize = 500;  { размер массива }

      { объявление типа для сверхбольшого числа }
type
      TBigNumber = array [1..CSize] of char;

var   BN1, BN2 : TBigNumber;       { два очень больших числа }

{ Процедура распечатки сверхбольшого числа }
procedure WriteBigNumber(var F: text; const aNum: TBigNumber);
var i : integer;
begin
  i:=CSize;
  while (i>0) and not (aNum[i] in ['1'..'9']) do Dec(i);
  if i=0 then Write(F, '0');
  while i>0 do begin
      Write(F, aNum[i]);
      Dec(i);
  end;
  Writeln(F); Writeln(F);
end;

{ Процедура сложения сверхбольших чисел в столбик.
  Результат помещается в первое число, что равносильно оператору сложения
  aNum1 := aNum1 + aNum2 }

procedure AddNumbers(var aNum1, aNum2 : TBigNumber);
var   i,j : integer;
      n1, n2 : integer;      { слагаемые цифры }
      sum, ovr : integer;    { сумма и перенос }
```

```
begin
  ovr:=0;  { в начале переполнение = 0 }
  { цикл по всем цифрам, кроме последней }
  for i:=1 to CSize-1 do begin
      j:=i;       { j используется после завершения цикла }
      { Если в текущей позиции пробел, то считаем его нулем,
      а иначе символ цифры преобразуем в цифру 0..9 }
      if aNum1[i]=' '
            then n1:=0
            else n1:=Ord(aNum1[i])-Ord('0');  { n1 = 0..9 }
      if aNum2[i]=' '
            then n2:=0
            else n2:=Ord(aNum2[i])-Ord('0');  { n2 = 0..9 }
      sum:= (n1+n2+ovr) mod 10;          { сумма sum = 0..9 }
      ovr:= (n1+n2+ovr) div 10;          { перенос ovr = 0 или 1 }
      { Преобразуем цифру в символ цифры }
      aNum1[i]:= Char(sum + Ord('0'));
  end;
  { Если было переполнение, то за последней цифрой помещаем единицу }
  if ovr<>0 then aNum1[j+1]:='1';
end;

var F : text;     i : integer;

begin       { === Главная программа === }
  Assign(F, ''); Rewrite(F);
  FillChar(BN1, SizeOf(BN1), ' ');  FillChar(BN2, SizeOf(BN2), ' ');
  for i:=1 to CSize-1 do BN1[i]:= Char(Random(100) mod 10 + Ord('0'));
  for i:=1 to CSize-1 do BN2[i]:= Char(Random(100) mod 10 + Ord('0'));
  WriteBigNumber(F, BN1);     { первое слагаемое }
  WriteBigNumber(F, BN2);     { второе слагаемое }
  AddNumbers(BN1, BN2);
  WriteBigNumber(F, BN1);     { сумма }
  Close(F); Readln;
end.
```

Вы заметили, что количество сложений в цикле на единицу меньше размера массива? — одно место в массиве припасено на случай переноса из старшего разряда. Результат работы программы на моем компьютере таков.

Первое слагаемое (499 цифр):

```
88034474755263463811157748177169236752040135153256253684350812170455816590318
```

Второе слагаемое (499 цифр):

Сумма (500 цифр):

открытия, ни закрытия, и могут передаваться в качестве параметров процедур, как и другие файловые переменные.

А слабо?

А) Постройте сверхбольшие числа на основе строковых переменных (количество цифр — не более 255).

Б) Напишите процедуру для вычитания сверхбольших чисел. Или слабо? Учтите, что разность может быть и отрицательной!

В) Автоматически объявленные файловые переменные **`Input`** и **`Output`** по умолчанию связаны соответственно с клавиатурой и экраном. Но их можно связать и с дисковыми файлами, например:

```
Assign(Input, 'Data.In'); Reset(Input);
Assign(Output, 'Data.Out'); Rewrite(Output);
Readln(S);          { Чтение строки из Data.In }
Writeln(S);         { Запись строки в Data.Out }
Close(Input);       Close(Output);
```

Воспользуйтесь этим приемом для вывода сверхбольшого числа в текстовый файл. Переделайте процедуру **`WriteBigNumber`**, устранив первый параметр, — файловую переменную.

Задачи на темы предыдущих глав

Г) Жители райцентра Бюрократовка дневали и ночевали в очереди за справками. Всё потому, что там применяли механический текстовый файл — огромную скрипучую книгу, которая листалась лишь в одном направлении — от начала к концу файла. Если первая буква фамилии очередного посетителя следовала по алфавиту далее, чем у предыдущего, то чиновник продолжал листать страницы с текущей позиции, а иначе открывал на первой и листал от начала. Переход от одной буквы алфавита к другой и возврат в начало занимали один час. Так, если буквы следовали в порядке «АБВ», то на выдачу справок тратилось три часа, а если в обратном порядке — «ВБА», — то шесть часов (3+2+1). Если же первые буквы фамилий совпадали, то книгу всё равно листали заново, поэтому на «БББ» тратилось шесть часов. Создайте функцию, принимающую «очередь посетителей» — строку из прописных латинских букв, — и возвращающую время, необходимое для выдачи всех справок.

Д) Томясь в бюрократической очереди, свинопас Гришка нашел способ ускорить выдачу справок путем частичного упорядочения очереди (см. задачу **Г**). Создайте функцию, возвращающую такую частично упорядоченную строку (воспользуйтесь множеством символов). Напишите программу для сравнения времен по условиям задач **Г** и **Д**.

Глава 47
Системы счисления

Эта глава промчит нас дорогой, по которой человечество брело несколько тысячелетий, — мы научимся изображать числа.

Из тьмы веков

Когда явилась потребность в счете? — никто не помнит этого, но мудрецы всех времен упорно искали удобные способы изображения чисел. Поиск систем счисления — так их теперь называют — это захватывающая история! Трудно поверить, но античные математики ещё не знали десятичной системы! И как они решали свои замысловатые задачи?

Первой системой счисления была, очевидно, единичная. Тогда некоторому количеству одних предметов сопоставляли такое же количество других (камушков, ракушек или зарубок на дереве). Что тут скажешь? — каменный век! Изображать большие числа в этой системе немыслимо.

Потребовались века, чтобы индийцы додумались до цифр. Их цифры были похожи на современные «1», «2», «3» и так далее. Но истинную революцию в арифметике содеяла цифра «0». Тот, кто её придумал, поставил всё на свои места, причем в буквальном смысле. Ведь ноль породил позиционную десятичную систему счисления, где «вес» цифры определяется её позицией внутри числа. Странно, что в просвещенной Европе удобная десятичная система приживалась непросто и вытеснила неудобную римскую только в 15-16 веках!

Наконец пробил час немецкого математика Лейбница, в голову которого пришла здравая мысль: «Зачем так много цифр? — изумился он, взглянув на циферблат своих часов, — когда вполне достаточно двух!». Так была изобретена двоичная система счисления, — «родная» для нынешних компьютеров.

Число и его изображение

Пора прояснить, что же такое системы счисления? Числа — это плод нашего воображения, в природе их никто не видел, они существуют лишь в наших головах. Не потому ли с числами связаны порой курьезные заблуждения? Иные полагают, что перевод числа из одной системы счисления в другую меняет это число. Вам смешно? Взгляните на рис. 103, где устроилась дюжина попугаев. Дюжина — это двенадцать, я написал это по-русски, а мог бы на другом языке. Или обозначил бы китайским иероглифом, — количество попугаев от этого не изменится. Как в поговорке: хоть горшком назови, только в печь не сажай!

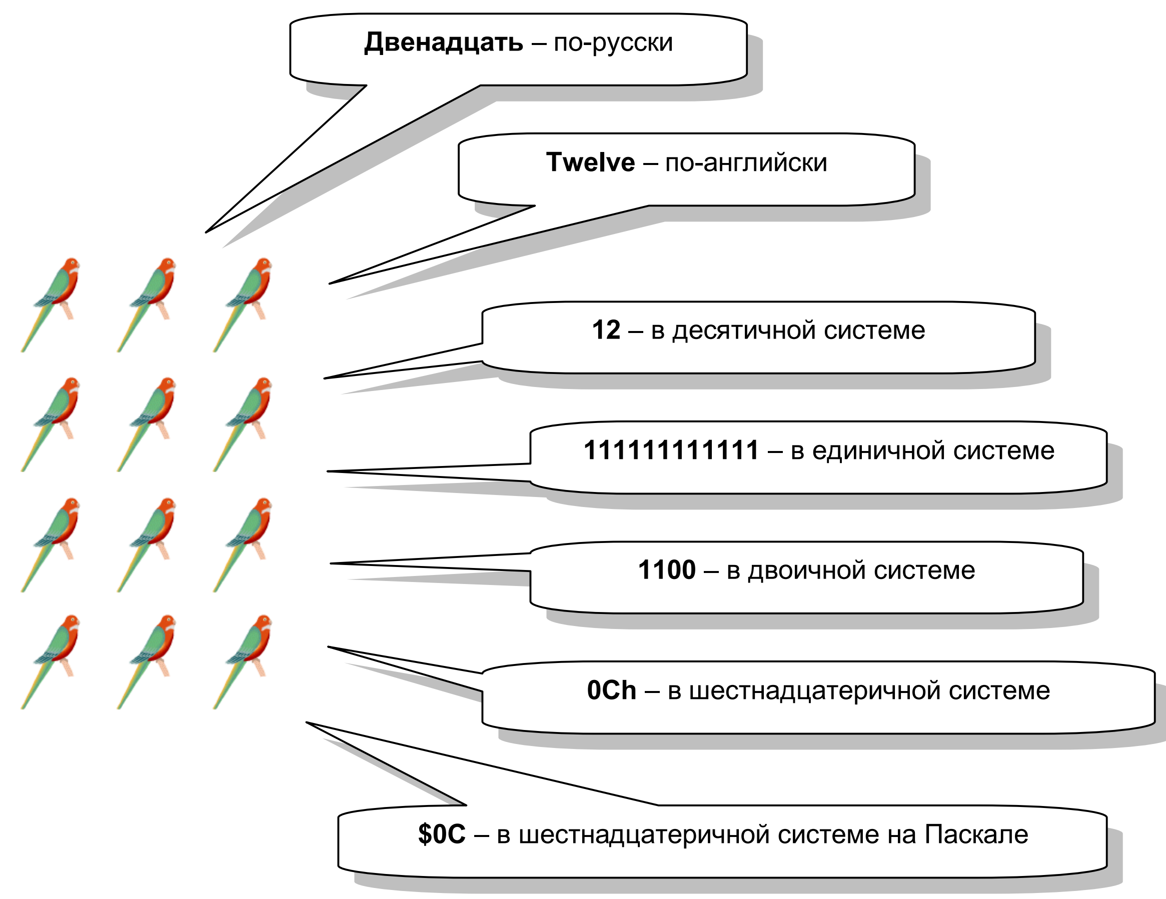


Рис. 103 – Способы изображения числа 12

Итак, что ни скажи, но на картинке всё те же двенадцать попугаев. Это число изображено рядом в нескольких системах счисления: единичной, десятичной, двоичной и шестнадцатеричной. И, хотя изображения не схожи меж собой, все они относятся к двенадцати попугаям. Стало быть, число и его изображение — не одно и то же!

Мы изображаем числа строками символов — цифрами. Поручив процедуре **Writeln** напечатать число, мы не задумываемся, как она делает это, — число превращается в строку цифр неведомым нам образом. Допустим на минуту, что процедура **Writeln** этого не умеет, и тогда явится потребность сделать такое преобразование самим. Итак, ставим себе первую задачу: преобразовать число в строку, то есть получить символьное изображение числа.

Справившись с первой задачей, займемся обратным преобразованием — строки в число. Это умеет процедура **Readln**, но мы пока забудем об этом. Дело в том, что упомянутые стандартные процедуры понимают лишь десятичную систему счисления. Мы же добиваемся большего, — мы хотим изображать числа в любой системе счисления (двоичной, троичной и так далее). А начнем, разумеется, с родной десятичной системы.

Десятичная система

Десятичную систему знает всякий: здесь крайняя правая цифра числа означает единицы, а последующие — десятки, сотни и так далее. Например, число 2048 представляется так.

$$2048 = \underline{\mathbf{2}} \cdot 1000 + \underline{\mathbf{0}} \cdot 100 + \underline{\mathbf{4}} \cdot 10 + \underline{\mathbf{8}} \cdot 1$$

Или так.

$$2048 = \underline{\mathbf{2}} \cdot 10^3 + \underline{\mathbf{0}} \cdot 10^2 + \underline{\mathbf{4}} \cdot 10^1 + \underline{\mathbf{8}} \cdot 10^0$$

То есть, позиция цифры в числе равна показателю степени при десятке, если счет позиций вести справа налево, начиная с нуля.

Повторю нашу цель: мы хотим превратить нечто цельное — число — в цепочку символов. Как это сделать? Есть мысли? Я предлагаю «откалывать» от числа цифру за цифрой, превращая их в символы и складывая в строку. Из опыта известно, что легче всего «отгрызть» от числа младшую цифру, вычисляя остаток от деления на десять, вот так.

```
младшая_цифра := число MOD 10
```

Тогда старшая часть числа отделится от младшей цифры делением на десять. При этом остаток будет отброшен, но он теперь и не нужен, поскольку сохранен в младшей цифре.

```
старшая_часть := число DIV 10
```

Так прояснилась схема дробления числа, показанная на рис. 104.

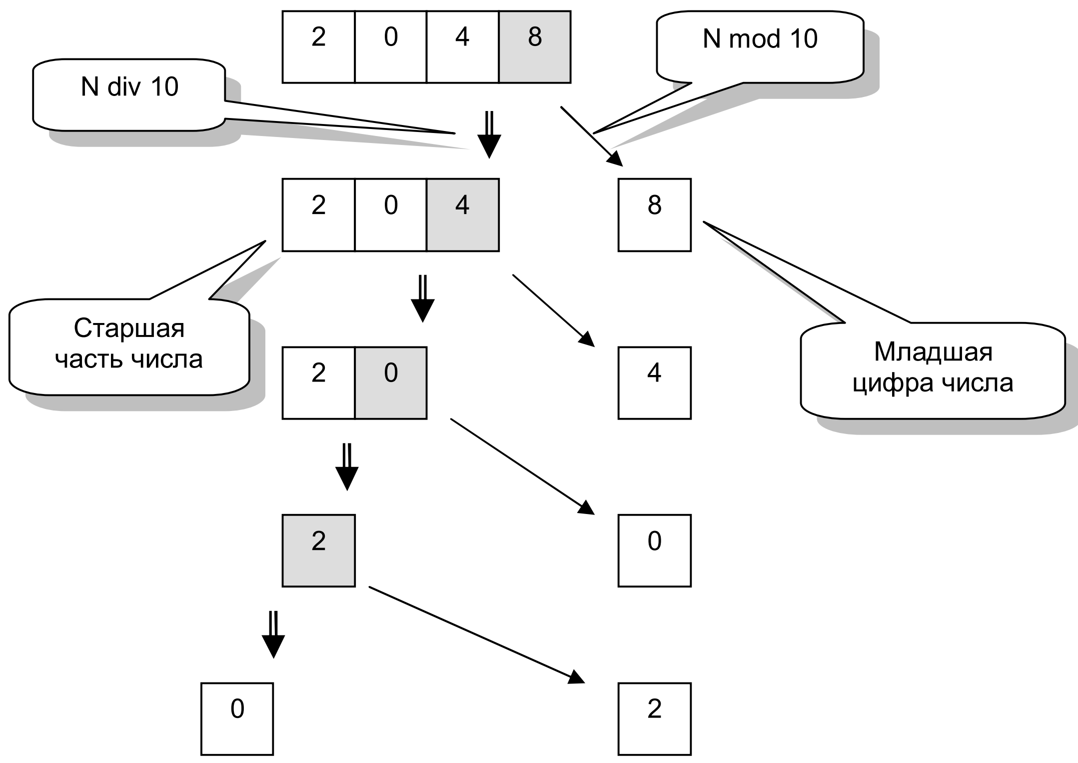


Рис. 104 – Выделение отдельных цифр десятичного числа

Число дробится, пока в старшей части не окажется ноль. Осталось лишь организовать цикл, условием выхода из которого будет равенство нулю старшей части. Эта несложная программа перед вами.

```
var N : integer;    S : string;
begin       { Преобразование числа в строку десятичных цифр }
  Write('N= '); Readln(N);
  S:='';
  repeat
    S:= Char((N mod 10)+Ord('0')) + S;    { выделение очередной цифры }
    N:= N div 10;                         { отделение старшей части }
  until N=0;
  Writeln(S);  Readln;
end.
```

Теперь, когда мы смогли превратить число в строку, займемся обратным превращением — соберем число из символов строки. Откуда подступиться к этой сборке? Запишем разложение числа с помощью скобок следующим образом:

$$2048 = \underline{\mathbf{2}} \cdot 1000 + \underline{\mathbf{0}} \cdot 100 + \underline{\mathbf{4}} \cdot 10 + \underline{\mathbf{8}} \cdot 1 = (((0 \cdot 10+\underline{\mathbf{2}}) \cdot 10+\underline{\mathbf{0}}) \cdot 10+\underline{\mathbf{4}}) \cdot 10+\underline{\mathbf{8}}$$

Правила действий со скобками требуют начать вычисление с внутренних, самых глубоких скобок. Следовательно, сборку числа из отдельных цифр начнем со старших разрядов, последовательно умножая накопленную сумму на 10. Внутри

самых глубоких скобок добавлено слагаемое 0•10. Не влияя на результат вычислений, оно придает общность алгоритму сборки, который показан на рис. 105.

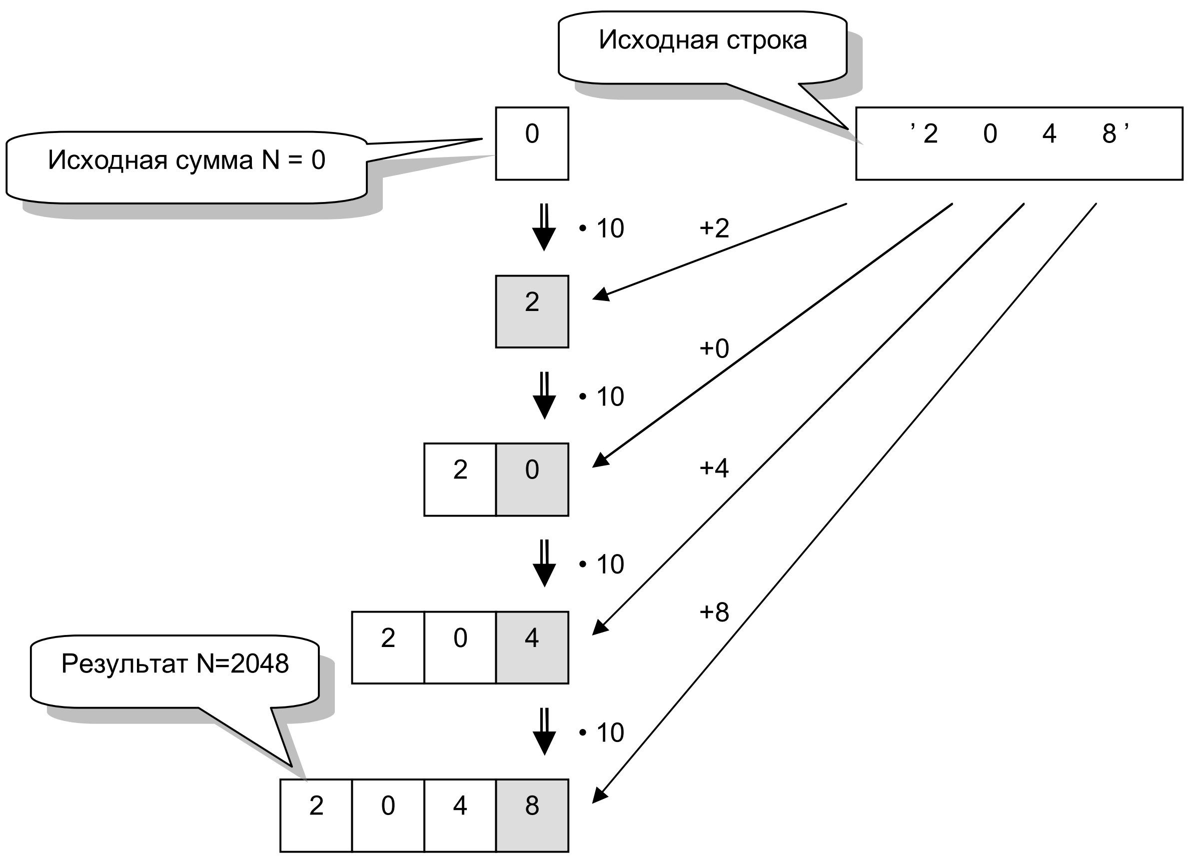


Рис. 105 – Сборка числа из строки десятичных цифр

Например, для числа 2048 сборка пойдет в таком порядке:

```
N = 0 - исходное значение
N = 0 • 10 + 2 = 2
N = 2 • 10 + 0 = 20
N = 20 • 10 + 4 = 204
N = 204 • 10 + 8 = 2048
```

А вот программа, работающая по этому алгоритму.

```
var N : integer;    i : integer;    S : string;
begin
  Write('S= '); Readln(S);
  N:=0;
  for i:=1 to Length(S) do   N:= 10*N + Ord(S[i]) - Ord ('0');
  Writeln(N);  Readln;
end.
```

Разобравшись со сборкой-разборкой десятичных чисел, замахнемся теперь на процедуры, пригодные для любых систем счисления. Но прежде ознакомимся с устройством этих систем.

Двоичная система

«Отец» двоичной системы Лейбниц не помышлял о великом будущем своей придумки, и на долгие годы о ней забыли. Но изобретатели компьютеров вспомнили. Все компьютеры — от первых моделей до самых современных — строятся из простейших элементов памяти — триггеров. Триггер — это электронная схема с двумя устойчивыми состояниями. Подобие триггера — комнатный выключатель, что может (если исправен) находиться в двух устойчивых состояниях: «включен» и «отключен». То есть, выключатель «помнит» состояние, в которое его привели в последний раз, и является элементом памяти.

Итак, элементы памяти с двумя состояниями — триггеры — составляют основу компьютеров (и почему их не назвали «двaггерами»?). Одно из состояний инженеры обозначили числом 0, а другое — 1. Стало быть, триггер способен «помнить» одно из этих чисел. Маловато для серьезного счета, не так ли? Тогда и вспомнили о двоичной системе Лейбница. Инженеры соединили несколько триггеров в цепочку и назвали эту «гирлянду» регистром. Каждый триггер в регистре, подобно цифрам в десятичном числе, обладает своим весом. В зависимости от позиции в регистре, вес триггера может составлять 1, 2, 4, 8 и так далее, — это степени числа 2. Например, число 12 изображается в двоичной системе так (рис. 106).

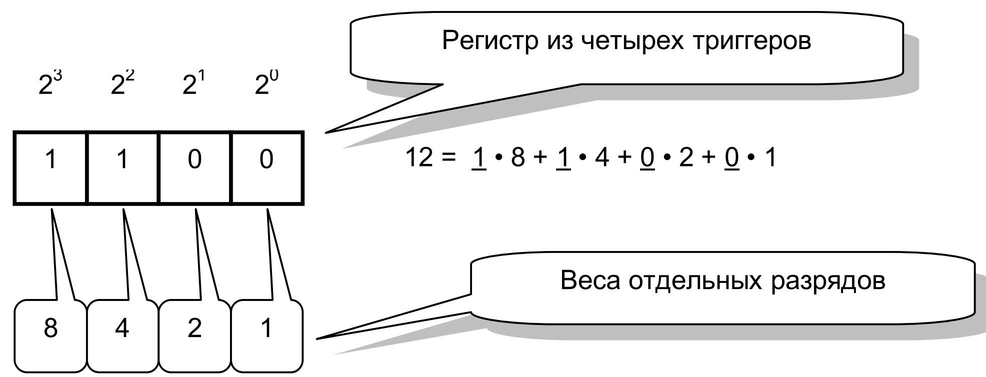


Рис. 106 – Изображение числа 12 в двоичной системе

Сравните эту кодировку с десятичной системой, — принцип тот же, только веса разрядов другие. Если в десятичной системе вес очередного разряда вдесятеро больше предыдущего, то в двоичной системе — вдвое. Числа, хранящиеся в триггерах (0 или 1) служат множителями этих весов. Таким образом, при достаточной длине регистра в двоичной системе можно изобразить сколь угодно большое число.

Договоримся о форме записи двоичных чисел, иначе путаницы не избежать. У программистов приняты две формы: к символам двоичного изображения добавляют либо суффикс «B» (от Binary — «двоичный»), либо маленькую двоечку. Например, число 12 в двоичной системе записывается так.

1100B или **1100b** или **1100_2**

А иначе эту запись можно понять как «тысяча сто» в десятичной системе.

Шестнадцатеричная система

Компьютеры никогда не жаловались на двоичную систему, она их вполне устраивает. Сетовать стали программисты, — уж очень громоздкой получалась запись сравнительно небольших чисел, например:

$4005 = 111110100101_2$

А если программистам несподручно, они что-нибудь придумают. Придумка была простой: двоичную запись разбили на группы по четыре двоичных цифры в каждой — тетрады (от греческого слова Tetra — «четыре»). И каждую тетраду записали в привычной для людей десятичной системе, разделяя тетрады точками. Например, десятичное число 4005 преобразили так.

$4005 = 111110100101_2 \rightarrow 1111.1010.0101_2 \rightarrow 15.10.05$

Тетрады могут содержать числа от 0 до 15 — всего получается 16 значений, потому систему назвали шестнадцатеричной. Со временем запись сделали ещё короче, заменив числа от 10 до 15 буквами латинского алфавита:

```
A=10
B=11
C=12
D=13
E=14
F=15
```

Тогда показанная выше запись преобразилась так: **15.10.05 → FA5**

Рис. 107 показывает это наглядней.

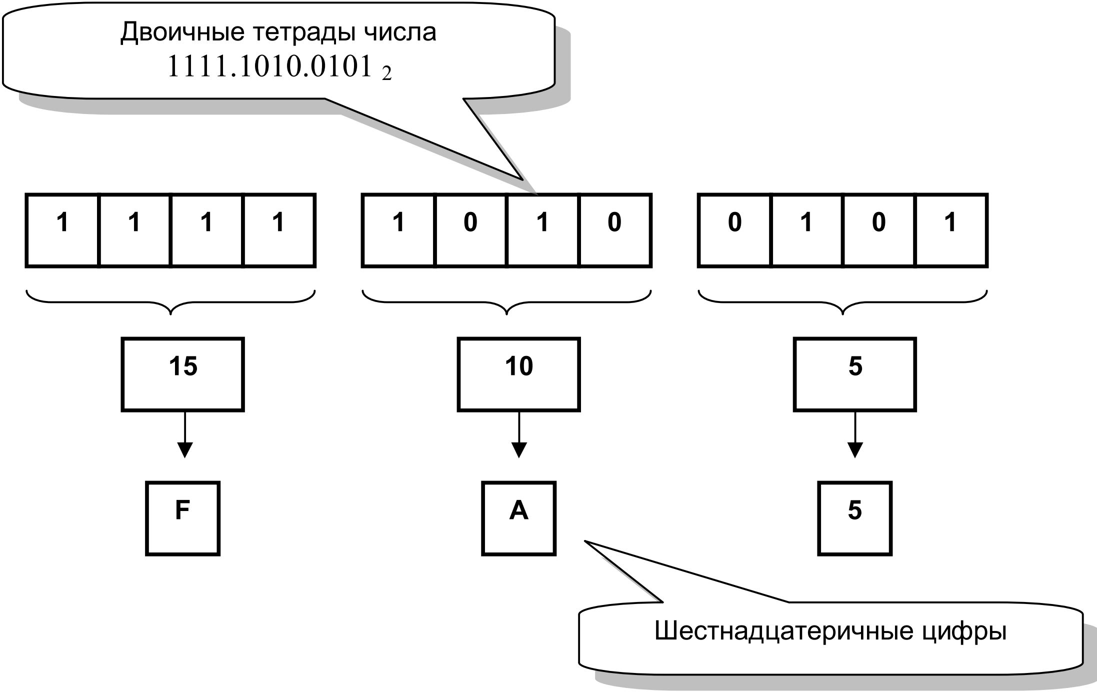


Рис. 107 – Преобразование двоичного числа в шестнадцатеричное

Шестнадцатеричную запись можно спутать с десятичной, и даже принять за слово, поскольку в ней встречаются буквы. Потому для таких чисел учредили свои правила: шестнадцатеричная запись числа должна начинаться с цифры, а завершаться суффиксом «H» (от Hexadecimal, Hex — «шестнадцатеричный»). Значит, изобразить число **FA5** правильней так:

0FA5H или **0FA5h**

Применяют и другие формы записи шестнадцатеричных чисел. Так, в языке Си принята приставка «0x» (**0xFA5**), а в Паскале начинают с приставки «$» — это знак доллара (**$FA5**). В таких записях лидирующий ноль не требуется, но для лучшего восприятия указывают обычно две, четыре, либо восемь цифр (в зависимости от величины числа или разрядности данных), например:

```
12   =   0x0C       =   $0C        ← байт (byte)
4005 =   0x0FA5     =   $0FA5      ← слово (word)
4005 =   0x00000FA5 =   $00000FA5  ← длинное слово (longint)
```

Чем хороша шестнадцатеричная система? Легкостью перевода чисел в двоичную систему и обратно. После небольшой тренировки любой может сделать это в уме. При переводе в двоичную систему заменяем каждую шестнадцатеричную цифру четырьмя двоичными и «склеиваем» эти тетрады между собой. И, хотя компьютеры по-прежнему работают в двоичной системе, программисты дружно перешли на шестнадцатеричную. Вот таблица для перевода небольших чисел из одной системы в другую.

Табл. 11 – Изображения чисел в различных системах счисления

Десятичная	Двоичная	16-ричная	Десятичная	Двоичная	16-ричная
0	**0000**	**0**	**8**	**1000**	**8**
1	**0001**	**1**	**9**	**1001**	**9**
2	**0010**	**2**	**10**	**1010**	**A**
3	**0011**	**3**	**11**	**1011**	**B**
4	**0100**	**4**	**12**	**1100**	**C**
5	**0101**	**5**	**13**	**1101**	**D**
6	**0110**	**6**	**14**	**1110**	**E**
7	**0111**	**7**	**15**	**1111**	**F**

Другие системы счисления

Итак, мы познакомились с тремя позиционными системами счислений: десятичной, двоичной и шестнадцатеричной. Существуют ли другие системы? Конечно! Во всех позиционных системах вес цифры определяется её положением в числе, сравните.

$2048 = \underline{\mathbf{2}} \cdot 10^3 + \underline{\mathbf{0}} \cdot 10^2 + \underline{\mathbf{4}} \cdot 10^1 + \underline{\mathbf{8}} \cdot 10^0$ - десятичная;

$12 = 1100_2 = \underline{\mathbf{1}} \cdot 2^3 + \underline{\mathbf{1}} \cdot 2^2 + \underline{\mathbf{0}} \cdot 2^1 + \underline{\mathbf{0}} \cdot 1^0$ - двоичная;

$4000 = \$FA0 = \underline{\mathbf{F}} \cdot 16^2 + \underline{\mathbf{A}} \cdot 16^1 + \underline{\mathbf{0}} \cdot 16^0$ - шестнадцатеричная.

Число, на котором построена система, называют её основанием. Можно выдумать столько систем счисления, сколько существует чисел, то есть, бесконечно много. Пока нам достаточно тех, что придуманы. А если с других планет прилетят существа с семью пальцами на руках? Для них, вероятно, «родной» будет семеричная система, и мы должны быть готовы к этому!

Так мы подошли к задаче по настоящему серьезной: изобразить число в некоторой системе счисления (основания систем ограничим числами от 2 до 16).

Изображение числа в заданной системе счисления

Преобразуя числа в десятичную систему, мы «отгрызали» цифры, начиная с младших разрядов, операциями деления и получения остатка. Точно так же преобразуют числа и в другие системы, только откалывают куски иного размера. Поскольку в двоичной системе есть только две цифры, то для неё младшая цифра отсекается операцией **MOD 2**, а старшая часть — операцией **DIV 2.** Для шестнадцатеричной системы — соответственно операциями **MOD 16** и **DIV 16.** Отсюда следует правило: для преобразования числа в **N**-ричную систему счисления

младшую цифру отделяют операцией **MOD N**, а старшую часть числа — операцией **DIV N**.

В программе P_47_1 функция **ConvertFromNumber** — «преобразовать из числа» — делает именно то, о чем сказано выше. Обратите внимание на строковую константу:

```
const CDigits : string = '0123456789ABCDEF';
```

Она служит для изящного преобразования чисел 0—15 в шестнадцатеричные цифры «0»—«F». Константы, для которых явно указан тип, называют типизированными, — это пример такой константы.

```
{ P_47_1 - Преобразование в произвольную систему счисления }
{ Функция преобразования десятичного числа в другие системы счисления }
function ConvertFromNumber(aBase, aNumber : integer): string;
const CDigits : string = '0123456789ABCDEF';
var n : integer;    c : char;    S : string;
begin
  S:=''; { Накопитель цифр }
  repeat
      n:= aNumber mod aBase;        { остаток от деления на основание }
      aNumber:= aNumber div aBase;  { частное от деления на основание }
      c:= CDigits[1+n];             { выбираем цифру из строки }
      S:= c + S;                    { вставляем цифру в результат }
  until aNumber=0;
  ConvertFromNumber:= S; { готово! }
end;

var B, N : integer;   { B - основание системы, N - число }
begin {=== Главная программа ===}
  repeat
      Write('Основание системы= '); Readln(B);
      if B in [2..16] then begin
            Write('Преобразуемое число= '); Readln(N);
            Writeln(ConvertFromNumber(B, N));
      end
  until not (B in [2..16]);
end.
```

Эта простая программа подарит вам счастье наблюдать знакомые десятичные числа в экзотических системах счисления, например, в троичной или пятеричной.

Обратное преобразование

Теперь займемся обратной задачей: пусть дана строка символов, изображающая некое число в известной системе счисления; требуется преобразовать эту строку в число и напечатать в десятичной системе.

Сборка числа из десятичных цифр нами освоена. Она выполнялась умножением накопленной суммы на десять с прибавлением очередной цифры, начиная со старшей. Надо ли объяснять, что сборка в других системах выполняется точно так же? Только умножать будем не на десять, а на основание системы счисления. В следующей ниже программе сборка выполняется функцией **`ConvertToNumber`** — «преобразовать в число».

```
{ P_47_2 - Преобразование из других систем счисления }
function ConvertToNumber(aBase: integer; aNumber: string): integer;
var   i,n, Sum : integer;
      c : char;
begin
  Sum:=0; { Накопитель результата }
  for i:=1 to Length(aNumber) do begin
      c:= Upcase(aNumber[i]);
      if c in ['0'..'9']
            then n:= Ord(c)-Ord('0')      {0..9}
            else n:= 10+Ord(c)-Ord('A');  {10..15}
      Sum:= aBase*Sum + n;    { Накопление суммы }
  end;
  ConvertToNumber:= Sum; { готово! }
end;

var   B : integer;    { Основание системы }
      N : string;     { Изображение числа в виде строки }
begin {=== Главная программа ===}
  repeat
      Write('Основание системы= '); Readln(B);
      if B in [2..16] then begin
            Write('Преобразуемое число= '); Readln(N);
            Writeln(ConvertToNumber(B, N));
      end
  until not (B in [2..16]);
end.
```

Как обычно, здесь подчеркнуты операторы, стоящие внимания. Функция **`UpCase`** преобразует строчные латинские буквы в заглавные. Ведь

шестнадцатеричные цифры от «A» до «F» могут быть введены пользователем в любом регистре, а последующие операторы преобразования цифры в число предполагают заглавные буквы, — вот потому и понадобилась функция **UpCase**.

Теперь о превращении символов в числа. Цифры от «0» до «9» преобразуются вычитанием из кода цифры кода символа «0». Для цифр от «A» до «F» после вычитания кода буквы «A» к разности прибавляем число 10. Всё сказанное относится к следующему условному оператору.

```
    if c in ['0'..'9']
          then n:= Ord(c)- Ord('0')           {0..9}
          else n:= 10 + Ord(c)- Ord('A');     {10..15}
```

Вот, пожалуй, и вся премудрость. Испытание этой программы убедит вас в том, что волшебства случаются не только в сказках!

Итоги

- Способ изображения чисел посредством знаков называется системой счисления.
- Одно и то же число может быть изображено в разных системах счисления.
- Все современные системы счисления – позиционные. Это значит, что вес цифры определяется позицией в числе.
- Преобразование числа в любую систему счисления (строку цифр) начинается с младших разрядов, а обратная сборка – со старших.

А слабо?

А) Напишите функцию для преобразования числа из одной системы счисления в другую. Функция должна принимать три параметра:

- строку в исходной системе счисления;
- основание исходной системы;
- основание конечной системы счисления.

Воспользуйтесь вызовами готовых функций **ConvertToNumber** и **ConvertFromNumber**.

Б) У программиста Ника была привычка запоминать сумму цифр в номерах автомобилей, попадавшихся ему на глаза. Однажды он стал свидетелем аварии, виновник которой скрылся. Ник сообщил полицейским только сумму цифр в номере нарушителя (сам номер Ник не помнил). Помогите полиции, и напишите программу, выводящую все трехзначные номера (от 001 до 999), сумма цифр которых равна **N** (значение **N** вводит пользователь).

В) Напишите функцию для представления чисел словами. Например, число 45 должно быть преобразовано в строку «сорок пять». Решайте задачу постепенно: сначала для однозначных и двузначных чисел, затем для более крупных. Или слабо?

Г) В романе «Евгений Онегин» есть такие строки: «Все предрассудки истребя, мы почитаем всех нулями, а единицами — себя». О какой системе счисления упомянул Александр Сергеевич?

Д) В функцию передаются три параметра: 1) число, 2) основание системы счисления, 3) символ цифры. Функция должна возвратить количество вхождений этой цифры в представление числа для указанной системы счисления.

Е) Напечатать все трехзначные числа, цифры которых (в десятичном представлении) различны, например: 123, 702.

Ж) Найти все шестизначные счастливые билеты. Счастливыми называют билеты, у которых сумма первых 3-х цифр равна сумме следующих 3-х. Например: 123 411. Напишите булеву функцию, определяющую «счастливость» билета.

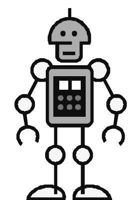
З) В заморской стране обращались денежные купюры достоинством в 1, 2, 5, 10 и 25 пиастров. Напишите программу для кассового аппарата, определяющую наименьший набор купюр, необходимый для выдачи сдачи на указанную сумму. Например, для сдачи 33 пиастров программа напечатает: 25 + 5 + 2 + 1.

И) Программа шифрования текстового файла заменяет каждый символ двумя шестнадцатеричными цифрами его кода. Например, три символа '**405**' заменяются на шесть символов '**343035**'. Символы разбивки строк не затрагиваются. Напишите программу для зашифровки и расшифровки файла по этой системе.

К) Чтобы усилить шифр предыдущей задачи, выполните вращение преобразованной строки на несколько позиций: влево — при зашифровке, и вправо — при расшифровке (смотрите задачи к главе 44).

Л) Напечатайте все числа, не превышающие 1000, такие, что делятся без остатка на каждую из своих цифр. Например: 24, 36, 184, 612. Определите количество таких чисел.

Глава 48
Железная логика



Разбираясь с двоичной системой, мы заглянули внутрь компьютера и обнаружили там регистры, что хранят и обрабатывают числа.

Два взгляда на компьютерные «кирпичики»

Регистры построены из триггеров — элементарных ячеек памяти, способных хранить один бит информации. В регистре может быть 8, 16, 32 или 64 триггера, что соответствует 1, 2, 4 или 8 байтам. Так видят устройство компьютера инженеры-электроники.

А программисты? Они видят то же самое, только называют иначе (рис. 108). То, что электроники именуют **триггерами**, программисты называют **битами**, а **регистры** нам видны как байты, слова и т.д. Так, в Паскале 8-битовый регистр соответствует типу `Byte`, 16-битовый — типу `Word`, а 32-битовый — типу `Longint`.

Я утверждал, что простые типы данных, такие как числа и символы, неделимы. Теперь признаюсь, что это не совсем так, поскольку в регистре процессора они представлены совокупностью битов. Процессор может работать не только с регистром в целом, но и с отдельными его битами. Иногда эту способность процессора используют и программисты, для чего Паскаль дает им надлежащие средства. Сейчас мы рассмотрим их.

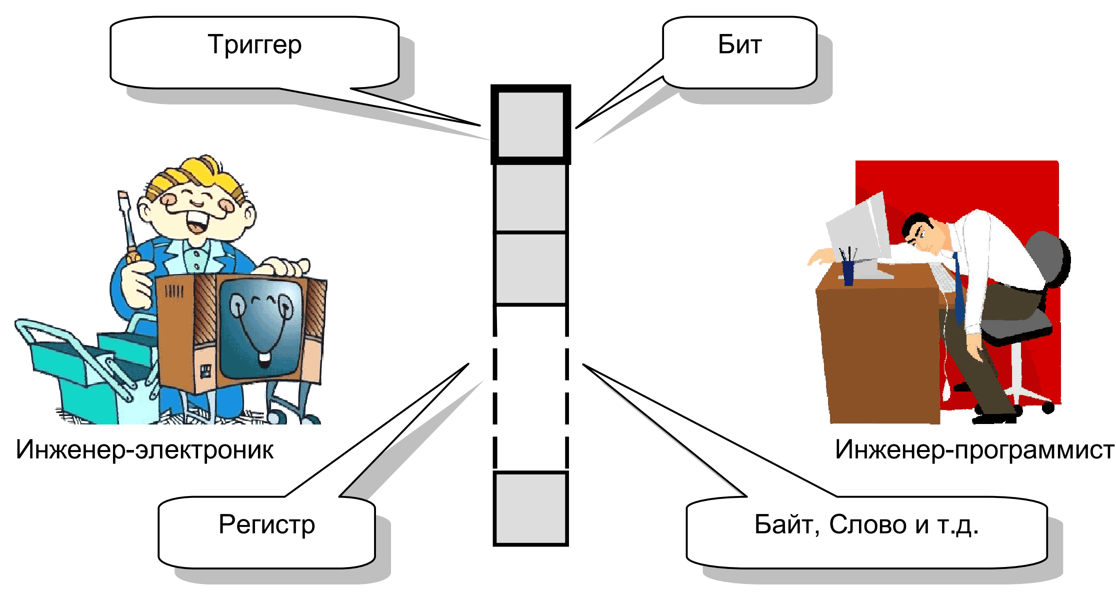


Рис. 108 – Устройство процессора глазами электроника и программиста

Логические операции в регистрах

Взгляните на эту, на первый взгляд бессмысленную программу.

```
var   A, B, C : integer;
begin
      A:= 5;     B:=16;     C:= A or B;
      Writeln( C );
end.
```

Здесь в переменную **C** заносится логическая сумма двух других числовых переменных. Но ведь логические операции применяют к булевым данным, причем здесь числа? Так вспомните о регистрах, где эти числа хранятся. Ведь это массивы битов! Содержимое битов можно трактовать и как числа 0 и 1, и как логические значения **FALSE** и **TRUE**. Именно так поступает Паскаль, выполняя логические действия с числами. В данном примере логически складываются шестнадцать независимых булевых пар с получением 16 битов результата. Похоже выполняются и другие логические операции с числами.

Известно, что переменная типа **BOOLEAN** занимает байт целиком, но использует лишь один из восьми битов, — расточительно, не так ли? Тогда как в байте можно хранить 8 булевых значений, в целом числе — 16, а в длинном целом — 32. Но экономия — не самое главное в жизни. Логические операции с числами дают интересные возможности для шифрования данных, их используют при обработке изображений и в иных случаях.

«Ладно, — скажете, — теперь бы увидеть это наяву». Легко! Наша следующая программа исследует булевы операции с числами. Самая серьезная её часть — функция преобразования байта в строку символов, то есть в двоичное представление этого числа. В программе P_47_1 нечто похожее выполняла функция **ConvertFromNumber**. Сейчас мы облегчим эту функцию, избавившись от одного параметра — основания системы счисления. К тому же теперь нам надо показать все восемь двоичных разрядов числа, включая незначащие нули. В результате этих изменений появилась на свет функция **ConvertTo2**, которую мы видим в программе P_48_1.

```
{ P_48_1 - исследование логических операций с числами }
function ConvertTo2(aNumber : integer): string;
var n, i : integer;    c : char;    S : string;
begin
  S:=''; { Накопитель цифр }
  for i:=1 to 8 do begin
    n:= aNumber mod 2;         { остаток от деления }
    c:= Char(n + Ord('0'));    { преобразуем в цифру }
    S:= c + S;                 { вставляем цифру слева }
    aNumber:= aNumber div 2;   { частное }
  end;
  ConvertTo2:= S;
end;
var A, B, C : byte;    { Операнды и результат }
begin {=== Главная программа ===}
  repeat
      Write('A= '); Readln(A);
      Write('B= '); Readln(B);
      C:= A or B;       { логическое сложение (объединение) }
      Writeln;
      Writeln('C= A OR B');
      Writeln('A= ',ConvertTo2(A), A:5);
      Writeln('B= ',ConvertTo2(B), B:5);
      Writeln('C= ',ConvertTo2(C), C:5);
      C:= A and B;      { логическое умножение (пересечение) }
      Writeln;
      Writeln('C= A AND B');
      Writeln('A= ',ConvertTo2(A), A:5);
      Writeln('B= ',ConvertTo2(B), B:5);
      Writeln('C= ',ConvertTo2(C), C:5);
      C:= not A;        { логическое отрицание (инверсия) }
      Writeln;
      Writeln('C= NOT A');
      Writeln('A= ',ConvertTo2(A), A:5);
      Writeln('C= ',ConvertTo2(C), C:5);
  until A=0;
end.
```

Главная программа не должна вызывать вопросов: после ввода пары чисел и выполнения логических операций с ними, на экран выводятся как исходные числа, так и результаты. Причем выводятся и в двоичной, и в десятичной системах счисления, например:

```
      C= A OR B
A= 00001101   13
B= 00001011   11
C= 00001111   15
      C= A AND B
A= 00001101   13
B= 00001011   11
C= 00001001    9
      C= A XOR B
A= 00001101   13
B= 00001011   11
C= 00000110    6
      C= NOT A
A= 00001101   13
C= 11110010  242
```

По результатам этих опытов выведены правила для логических операций (табл. 12). Логическое отрицание «НЕ» отличается от прочих тем, что применяется к одному операнду.

Табл. 12 – Правила выполнения логических операций с битами

Логическая операция	Пример	Правило
«**ИЛИ**» (сложение)	1010 OR 1100 = 1110	Результат единица, если **ХОТЯ БЫ ОДИН** из операндов равен единице.
«**И**» (умножение)	1010 AND 1100 = 1000	Результат единица, если **ОБА** операнда равны единице.
«**Исключающее ИЛИ**» (сравнение)	1010 XOR 1100 = 0110	Результат единица, если операнды **ОТЛИЧАЮТСЯ**.
«**НЕ**» (отрицание)	1010 NOT = 0101	Результат единица, если операнд **РАВЕН НУЛЮ**.

Заменив в этих правилах единицу на **TRUE**, а ноль на **FALSE**, вы получите правила для булевых данных.

Сдвиги влево и вправо

Сдвиг — одна из тех операций обработки регистров, которые выполняют все процессоры. В Паскале тоже предусмотрены две такие операции с числами: сдвиг влево (**SHL**) и сдвиг вправо (**SHR**).

Операция левого сдвига (рис. 109) перемещает все биты слова на заданное число позиций влево, при этом младшие биты заполняются нулями, а старшие теряются, например:

```
N:= 3;                  { 3 =  00000011 }
Writeln (N shl 1);      { 6 =  00000110 }
Writeln (N shl 2);      { 12 = 00001100 }
```

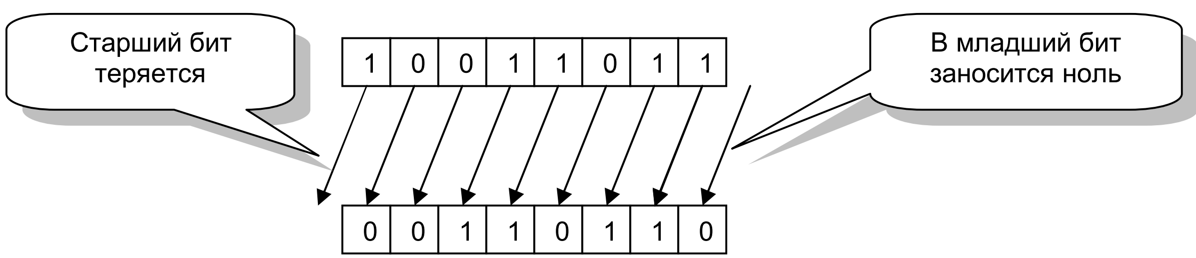


Рис. 109 – Сдвиг байта на один разряд влево

Операция правого сдвига (рис. 110) перемещает все биты слова на заданное число позиций вправо. При этом старшие биты заполняются нулями, а младшие теряются.

```
N:= 3;                  { 3 = 00000011 }
Writeln (N shr 1);      { 1 = 00000001 }
Writeln (N shr 2);      { 0 = 00000000 }
```

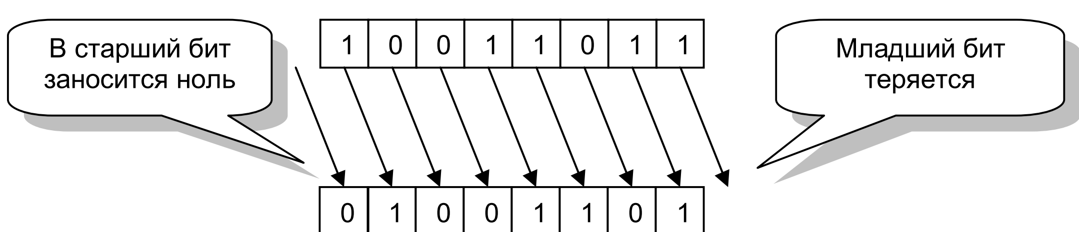


Рис. 110 – Сдвиг байта на один разряд вправо

Совместив сдвиг с логическими операциями, можно исследовать отдельные биты слова. Перед вами булева функция **TestBit**, принимающая два параметра: **ARG** — число, в котором проверяется состояние некоторого бита, и **BIT** — номер

этого бита. Функция возвращает **TRUE**, если проверяемый бит содержит единицу, и **FALSE** в противном случае.

```
function TestBit (arg: longint; bit : byte): Boolean;
begin
      TestBit := (arg and (1 shl bit)) <> 0
end;
```

Итоги

- Процессоры построены из триггеров. Триггер – это элемент с двумя устойчивыми состояниями, которые можно трактовать либо как булевы значения **TRUE** и **FALSE**, либо как числа 0 и 1.
- Для хранения чисел и других данных, триггеры соединены в регистры. Обычно регистр состоит из 8, 16, 32 или 64 битов.
- В Паскале есть средства для работы с регистрами – это логические операции и сдвиги. Они трактуют числа как массивы битов.

А слабо?

А) Напишите программу для исследования операций сдвига (подобную программе P_48_1).

Б) Наряду с рассмотренными здесь обычными сдвигами, в процессорах заложены операции циклического (кругового) сдвига. При таком сдвиге (рис. 111) выдвигаемый бит не теряется, а попадает соответственно в младший бит (при сдвиге влево) или в старший бит (при сдвиге вправо).

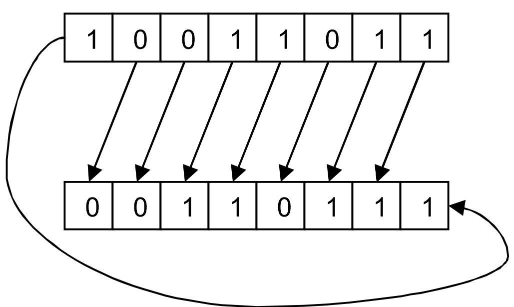


Рис. 111 – Циклический сдвиг влево

В Паскале нет операций циклического сдвига. Напишите функции для циклического сдвига слова влево и вправо. Подсказка: перед сдвигом каждого бита проверяйте состояние теряемого бита, а затем восстанавливайте его в младшем или старшем разряде.

Глава 49
Сложные массивы

Элементом массива может быть любой тип данных, и даже другой массив. Разбор следующих задач убедит вас в этом.

На поклон к Науке

Вернемся в тридевятое царство, история которого ещё далека от завершения. В 38-й главе мы узнали, что для исследования материка запустили спутник, передавший на землю номера границ тамошних стран. А программа, сработанная придворным программистом Ником, нашла по этим данным соседей царства А.

Молва о научном успехе дошла до купцов, и заронила в их души зерно надежды. Душевной болью торгашей были немалые пошлины, вносимые при каждом пересечении границы. Нет, купцы не надеялись избавиться от пошлин. Но, прежде чем везти товар, они желали знать о количестве пересекаемых границ, дабы прикинуть, стоит ли овчинка выделки? Раньше купцы ехали, куда глаза глядят и часто терпели убытки. Но теперь — иное дело, — можно предвидеть расходы. Требовалась лишь программа для определения минимального количества пересекаемых границ. С этой просьбой купцы и подкатили к придворному программисту, обещая весомое вознаграждение.

Ник выслушал купцов, и представил себе задачу так. Есть файл с номерами границ каждого государства, а также названия двух стран, назовем их условно А и Б. Надо вычислить наименьшее число пересекаемых границ на пути из страны А в страну Б. Напоминаю, что переходить границы в углах не разрешалось, поэтому соприкосновение стран углами не считается общей границей.

Имперское строительство

Ник принял заказ и погрузился в работу. Однако щедрые посулы купцов не содействовали раздумьям, — вдохновение не являлось, хоть убей! В таких случаях Ник пытался отвлечься; вот и сейчас его рука потянулась к полке и достала первую попавшуюся книгу — это была история средних веков. Книга открылась на странице с рассказом о зарождении средневековой империи. Парень увлекся чтением, забыв на время о своих неудачах. Но вскоре Ника осенило: «Ведь это то, что мне нужно, — блеснуло в его голове, — я должен построить империю!»

Вам приходилось строить империи? Тогда послушайте меня — опытного «императора». Строительство начинается с собственной страны — центра империи. Я готовлю мощную армию и накапливаю прочие ресурсы: оружие, горючее, продовольствие. Всё это нужно для «добровольного» присоединения соседей. Затем нападаю на них и покоряю поодиночке. После такой завоевательной кампании рождается новая страна с расширенными границами и другими соседями, которые ещё не ведают о своей судьбе! Дав отдых армии, и накопив ресурсы, я предпринимаю следующую завоевательную кампанию и

присоединяю соседей своих бывших соседей. Взгляните на рис. 112, — если строительство империи начать из страны D, то в ходе первой кампании будут поглощены соседи A, C и E, а в ходе второй — их соседи, страны B, I и F.

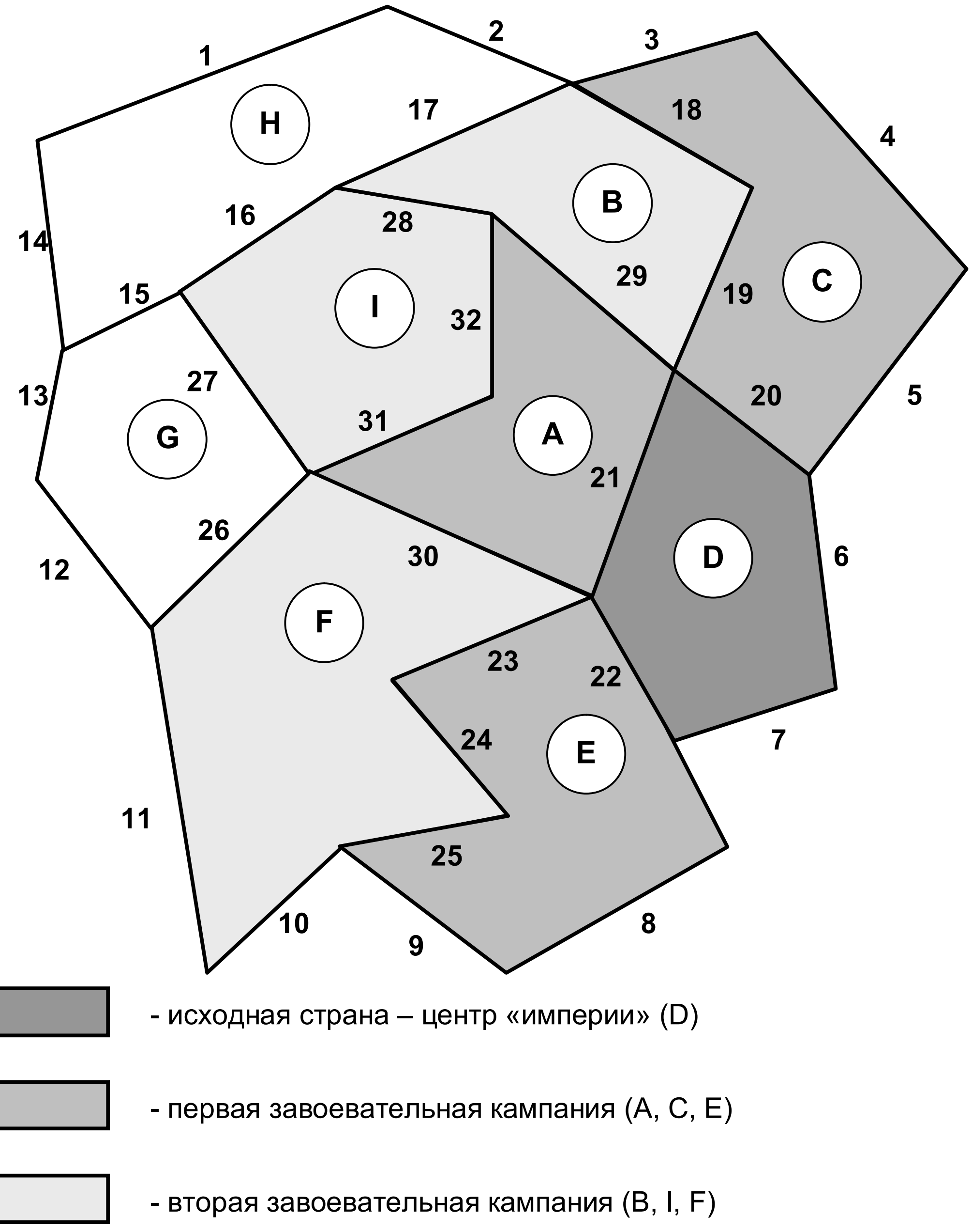


Рис. 112 – Строительство империи

Хорошо, — скажете, — но где тут связь с купеческим заказом? Сейчас объясню. Остроумный Ник догадался, что каждая завоевательная кампания уменьшает число границ между центром империи и любой другой, пока ещё независимой, страной ровно на единицу. И так будет, пока страна не поглотится империей, и граница между ними исчезнет. Стало быть, количество необходимых для поглощения страны завоевательных кампаний будет равно количеству пересечений границ, которое интересует купцов.

«Нашел, нашел!» — просветлел Ник, подвигаясь к компьютеру. Главная идея родилась, осталось обдумать детали. В первую очередь, следовало выбрать подходящий способ хранения границ. В 38-й главе для этого использовано несколько переменных-множеств. Теперь так не годится, — догадался Ник, — ведь для каждой завоевательной кампании мне надо организовать цикл. А там где циклы, там и массивы. Стало быть, мне нужен массив множеств. Ник нарек этот тип данных именем **TStates**.

```
type  TBoundSet = set of 1..255;    { множество границ одной страны }
      TStates = array ['A'..'Z'] of TBoundSet; { массив из множеств }
var   States : TStates;       { Переменная-массив }
```

Вы помните, как именовались страны в тех местах? Буквами латинского алфавита. Это надоумило Ника индексировать массив именно символами, — ведь это один из перечислимых типов, а все они пригодны для индексации. Тогда множество границ страны **B**, имя которой хранится в символьной переменной **X**, извлекается из массива множеств **States** так.

```
var   B : TBoundSet;     { множество границ одной страны }
...
      B:= States[X];     { здесь X = 'A'...'Z' - символ-название страны }
```

Но как в таком случае перебирать элементы массива? Ведь к символу не прибавишь единицу! Спасает функция **Succ**. Напомню, что она возвращает следующее по порядку значение перечислимого типа, например:

```
      X:= 'A';
      X:= Succ(X);       { X = 'B' }
      X:= Succ(X);       { X = 'C' }
```

Ещё один подводный камень, вовремя подмеченный Ником, был таков. При вводе имени несуществующей страны программа зациклится, вращаясь в замкнутом круге. Потому при вводе данных организован упрямый цикл **REPEAT-UNTIL**, вынуждающий пользователя ввести правильные названия стран.

И, наконец, последнее замечание к программе P_49_1 касается переменной **Temp** (что значит «временная»). Поскольку текущие границы империи накапливаются в переменной **EmpireB** и расширяются в ходе кампании, то определять бывших соседей по этим границам нельзя! Поэтому предыдущие границы империи перед началом цикла запоминаются в переменной **Temp**.

```
      Temp:= EmpireB;   { Запоминаем границы империи до начала кампании }
```

Теперь рассмотрите программу P_49_1, затем испытайте её.

```
{ P_49_1 – Решение «купеческой» задачи о пересечении границ }

type  TNameRange= 'A'..'Z';         { Диапазон возможных названий стран }
      TNameSet = set of TNameRange; { Множество названий стран }
      TBoundSet = set of 1..255;    { множество границ некоторой страны }
      { Массив множеств TStates – это границы всех стран }
      TStates = array ['A'..'Z'] of TBoundSet;

      { Процедура чтения  множества чисел (границ) из строки файла }
procedure ReadSet(var aFile: text; var aSet : TBoundSet);
var k : integer;
begin
      aSet:= [];
      while not Eoln(aFile) do begin
            Read(aFile, k);
            aSet:= aSet+[k];
      end;
      Readln (aFile);
end;
      { Глобальные переменные }
var   FileIn : text;          { Входной файл, полученный со спутника }
      States : TStates;       { Массив множеств границ }
      Names  : TNameSet;      { Множество имен всех стран на карте }
      C1, C2 : char;          { Имена стран "откуда" и "куда" }
      C      : char;          { Рабочая переменная для имен стран }
      EmpireB: TBoundSet;     { Границы империи }
      Temp   : TBoundSet;     { Предыдущие границы империи }
      EmpireN: TNameSet;      { страны империи }
      Counter: integer;       { Счетчик пересечений границ (результат) }

begin       {--- Главная программа ---}
  { Открываем входной файл }
  Assign(FileIn, 'P_38_3.in'); Reset(FileIn);

  { Готовим цикл чтения массива множеств }
  Names:=[ ]; { Названия стран }
  C:= 'A';    { Первый индекс в массиве стран }
```

```
  while not Eof(FileIn) do begin  { Цикл чтения массива множеств }
      ReadSet(FileIn, States[C]);   { Чтение одного множества }
      Names:= Names+[C];            { Добавляем имя страны }
      C:= Succ(C);                  { Переход к следующей букве }
  end;
  Close(FileIn);  { Теперь входной файл можно закрыть }
  repeat { «Упрямый» цикл чтения правильных имен стран }
      Write('Откуда: '); Readln(C1);
      Write('Куда  : '); Readln(C2);
      { Переводим имена стран в верхний регистр }
      C1:= UpCase(C1);    C2:= UpCase(C2);
      { Если имена не совпадают и оба достоверны, значит ввод правильный,
      в таком случае выходим из цикла, а иначе повторяем ввод }
      if (C1<>C2) and (C1 in Names) and (C2 in Names)
            then Break
            else Writeln('Ошибка! Повторите ввод имен стран');
  until False;
  { Подготовка к присоединению стран }
  EmpireB:= States[C1]; { Империя начинает расширяться от страны C1 }
  EmpireN:= [C1];       { Здесь накапливаются имена присоединенных стран }
  Counter:= 0;          { Счетчик "завоевательных кампаний" }
  { Цикл, пока не будет присоединена страна C2 }
  repeat
      { Подготовка к "завоевательной кампании" }
      C:='A';           { Первый индекс в массиве множеств границ }
      Temp:= EmpireB;   { Запоминаем предыдущие границы империи }
      { Цикл очередной "завоевательной кампании" по всем странам массива }
      while C in Names do begin
            { Если страна имеет общую границу, присоединяем её к империи }
            if (Temp * States[C]) <> [] then begin
                  EmpireB:= EmpireB + States[C];  { Добавляем границы }
                  EmpireN:= EmpireN + [C];        { Добавляем имя страны }
            end;
            C:= Succ(C);  { Следующий индекс в массиве множеств границ }
      end;  { очередная кампания завершена }
      Inc(Counter);     { Наращиваем счетчик "завоевательных кампаний" }
  until C2 in EmpireN;  { Пока не будет присоединена страна C2 }
  { Печать результата }
  Writeln('Пересечений границ: ', Counter); Readln;
end.
```

Так была решена задача о минимальной сумме пошлин. Купцы, было, обрадовались, но вскоре явились с новым поклоном. Много ль толку от знания расходов, если не знаешь пути, по которому надо двигаться? «Сделай нам, братан, что-то типа навигатора для поиска кратчайшего пути между странами. Так, чтобы нам меньше этих пошлин платить, — умоляли купцы, — мы не поскупимся!» Ник обещал подумать, и продолжение истории следует.

Крестики-нолики

Кто не любовался яркой световой рекламой? Рекламный щит составлен из лампочек, а изображение «рисуется» их включением и отключением, — этим управляет микропроцессор. Компьютер у нас под рукой, почему бы не соорудить такой рекламный щит прямо на экране? Научим компьютер выполнять с изображением на нашем щите три действия: два зеркальных отражения (относительно горизонтальной и вертикальной осей), а также инверсию изображения (рис. 113).

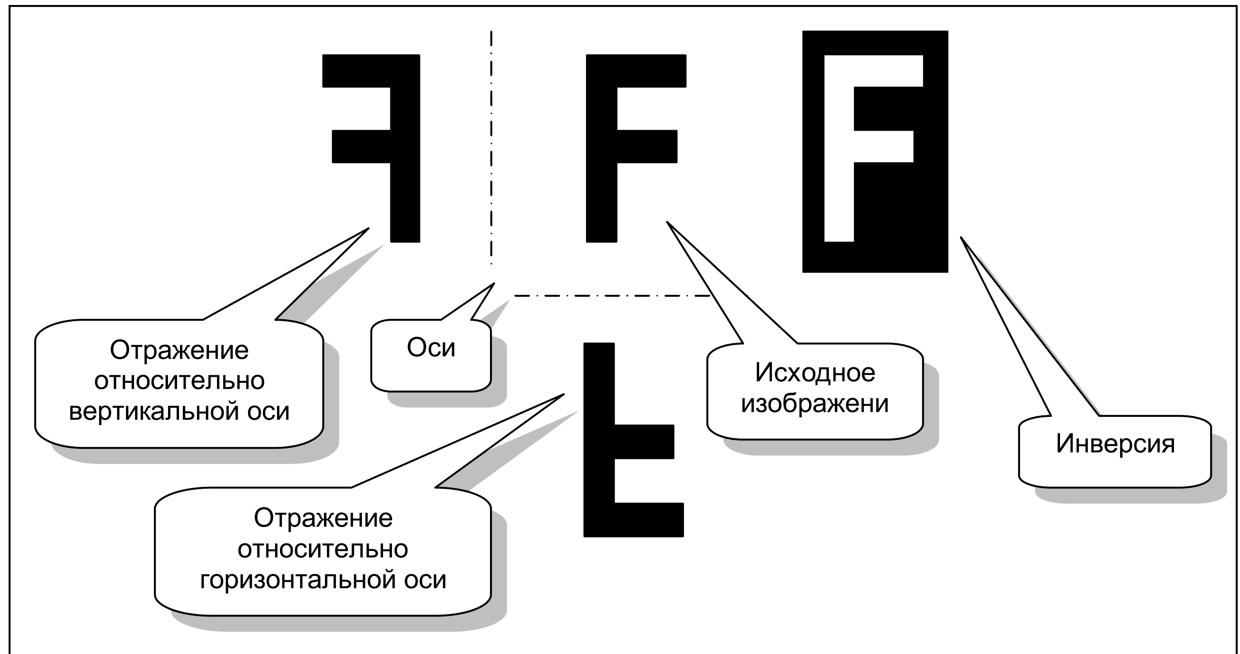


Рис. 113 – Операции с изображением на щите

Прежде всего, определим, как представить исходное изображение так, чтобы ввод его в компьютер был по возможности прост. Неплохим вариантом будет текстовый файл, где исходная картинка нарисована двумя символами, например, крестиком и ноликом. Присмотревшись к рис. 114, вы разглядите в правом верхнем углу изображенную таким способом букву «F».

```
0000000000000000000000000000000000000000
000000000000000000000000000+++++00000000
000000000000000000000000000+000000000000
000000000000000000000000000+000000000000
000000000000000000000000000+++++00000000
000000000000000000000000000+000000000000
000000000000000000000000000+000000000000
000000000000000000000000000+000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
```

Рис. 114 – Исходный файл для представления рекламного щита, крестиками нарисована буква «F»

Размер картинки выберем таким, чтобы она помещалась на экране монитора. В текстовом режиме экран содержит 25 строк по 80 символов в каждой. Мы ограничимся 20 строками по 40 символов. Тогда воображаемая вертикальная ось нашего щита пройдет между 20-м и 21-м столбцами, а горизонтальная — между 10-й и 11- строками.

Разобравшись с видимым представлением щита, обратимся к невидимому: придумаем способ хранения в памяти «лампочек» щита. Годятся ли символьные переменные, — те же крестики и нолики? Да, но мне приглянулся способ, лучше отвечающий природе рекламного щита. Ведь каждая из лампочек может быть либо включена либо погашена, — их состояние можно отразить в булевых переменных. А сколько их понадобится? Не так уж мало — 800 штук (20 строк по 40 в каждой). Разумеется, нужен массив, но каким он будет? Предположим, что тип «рекламный щит» (**Desk**) объявлен так.

```
type  TDesk = array [1..800] of Boolean;
```

Здесь разумеем, что первые 40 элементов массива хранят верхнюю строку щита, следующие 40 элементов, — вторую строку и так далее. Не очень удобно, правда?

Можно сделать иначе: сначала объявить отдельную строку щита **TLine** как массив из 40 лампочек.

```
type  TLine = array [1..40] of Boolean;
```

И тогда весь щит представится массивом из 20 таких строк, — это будет массив массивов.

```
type  TDesk = array [1..20] of TLine;
```

То же самое можно записать развернуто, вот так.

```
type  TDesk = array [1..20] of array [1..40] of boolean;
```

Подчеркнутое означает отдельную строку щита. Паскаль разрешает собрать все индексы объявления внутри одних скобок и записать всё это ещё короче.

```
type  TDesk = array [1..20, 1..40] of boolean;
```

Так мы получили структуру, которую математики называют матрицей, а программисты — двумерным массивом. Матрицы состоят из строк и столбцов. Для доступа к элементам матрицы нужны два индекса, один из которых указывает номер столбца, а другой — номер строки. Например, элемент матрицы **Desk**, стоящий в 5-м столбце 3-й строки, доступен так.

```
      Desk[3, 5]
```

Разумеется, что для индексов позволены числовые выражения, значения которых должны лежать в объявленных пределах. При обработке матриц применяют циклы, их можно организовать как по строкам, так и по столбцам. Возьмем для примера наш рекламный щит, объявим его тип, а потом заполним значением **FALSE**.

```
const Cx = 40;  { количество столбцов (ширина) }
      Cy = 20;  { количество строк (высота) }
type  TDesk = array [1..Cy, 1..Cx] of boolean; { тип «рекламный щит» }
var   Desk : TDesk;     { переменная «рекламный щит» }
```

Здесь пределы для индексов указаны через константы **Cx** и **Cy**. Заполнить матрицу значением **FALSE** можно двумя вложенными циклами:

```
      for y:=1 to Cy do
            for x:=1 to Cx do Desk[y, x]:= False;
```

То же самое делается быстрее и короче известной вам процедурой заполнения **FillChar**:

```
FillChar(Desk, SizeOf(Desk), false);
```

Здесь значение **SizeOf(Desk)** составит 800 — это количество элементов матрицы.

Можно обрабатывать и отдельные строки, и отдельные столбцы матрицы. Например, заполнить значением **TRUE** 5-й столбец.

```
for y:=1 to Cy do Desk[y, 5] := True;
```

А для заполнения 3-й строки организовать такой цикл:

```
for x:=1 to Cx do Desk[3, x] := True;
```

Если вам понятна техника работы с матрицами, перейдем к программе P_49_2.

Начнем с процедуры **ReadDesk**, что вводит матрицу из файла. Условимся считать, что крестикам в матрице **Desk** соответствует **TRUE**, а ноликам — **FALSE**. Входной файл обрабатываем построчно: сначала очередную строку читаем во вспомогательную строковую переменную **S**, а затем символы этой строки преобразуем в булевы значения оператором сравнения (вы помните, что оператор сравнения дает булев результат?).

```
Desk[y,x]:= S[x]='+';   { TRUE, если S[x] содержит крестик }
```

Следовательно, для ввода матрицы нужны два вложенных цикла: внешний — по строкам и внутренний — по столбцам.

Схоже работает и процедура **WriteDesk**, выводящая матрицу на экран. Здесь внутренний цикл формирует строку из 40 символов, каждый из которых может быть либо крестиком либо ноликом. Выбор пары символов — дело вкуса, в нашем случае пара определяется строковой константой **CSymbols**.

```
const CSymbols : string = '0+';
```

Нужный символ из этой строки выбирается по индексу.

```
S:= S + CSymbols[1+ Ord(Desk[y, x])];
```

Так, для значений **Desk[y,x]**, равных **FALSE,** будет выбран первый символ строки ('**0**'), а для **TRUE** — второй ('**+**'), что равнозначно следующему громоздкому оператору.

```
      if Desk[y, x]
            then S:= S + CSymbols[2]
            else S:= S + CSymbols[1]
```

Далее следуют две простые процедуры зеркального отражения матрицы относительно горизонтальной и вертикальной осей, — они всего лишь переставляют симметрично расположенные элементы.

Процедура инверсии рекламного щита ещё проще, — она меняет значения элементов матрицы на противоположные. Наконец, в главной программе после чтения из файла исходного изображения организован цикл ввода и обработки команд пользователя. Вводя одну из трёх команд (1, 2 или 3), пользователь крутит изображение туда-сюда, а также инвертирует его. Вот полный текст этой программы.

```
{ P_49_2 - Рекламная панель "крестики-нолики" }

const Cx = 40;  { количество столбцов (ширина) }
      Cy = 20;  { количество строк (высота) }

type  TDesk = array [1..Cy, 1..Cx] of boolean;
var   Desk : TDesk;

      { Чтение исходного состояния панели из текстового файла }
procedure ReadDesk(var F: Text);
var   x, y: integer;    { x - индекс столбца, y - индекс строки }
      S: string;
begin
  FillChar(Desk, SizeOf(Desk), false);
  y:=1;
  while not Eof(F) and (y<=Cy) do begin
      Readln(F, S);
      x:=1;
      while (x<=Length(S)) and (x<=Cx) do begin
            Desk[y,x]:= S[x]='+';
            Inc(x);     { x:= x+1 }
      end;
      Inc(y);           { y:= y+1 }
  end
end;
```

```
      { Вывод текущего состояния панели в текстовый файл }
procedure WriteDesk(var F: Text);
const CSymbols : string = '0+';
var   x, y: integer;    S: string;
begin
  for y:=1 to Cy do begin
      S:='';
      for x:=1 to Cx do S:= S + CSymbols[1+ Ord(Desk[y, x])];
      Writeln(F, S);
  end;
end;
      { Вспомогательная процедура обмена местами булевых переменных }
procedure Swap (var a, b : boolean);
var t : boolean;
begin
  t:=a;  a:=b;  b:=t;
end;
      { Отражение относительно вертикальной оси }
procedure Vert;
var x, y: integer;
begin
  for y:=1 to Cy do
    for x:=1 to Cx div 2 do Swap(Desk[y, x], Desk[y, Cx-x+1])
end;
      { Отражение относительно горизонтальной оси }
procedure Horisont;
var x, y: integer;
begin
  for y:=1 to Cy div 2 do
      for x:=1 to Cx do Swap(Desk[y, x], Desk[Cy-y+1, x])
end;

      { Инверсия рекламной панели }
procedure Invers;
var x, y: integer;
begin
  for y:=1 to Cy do
      for x:=1 to Cx do Desk[y, x]:= not Desk[y, x]
end;
```

```
var   FileIn : Text;
      cmd : integer;

begin       {=== Главная программа ===}
  Assign(FileIn, 'P_46_2.in'); Reset(FileIn);
  ReadDesk(FileIn);
  Close(FileIn);
  repeat
      WriteDesk(Output);      { вывод «щита» на экран }
      Writeln;
      Write('1- Вертикальная; 2- Горизонтальная; 3- Инверсия, 0- Выход : ');
      Readln(cmd);            { Ввод команды }
      case cmd of
            1: Vert;          { отражение относительно вертикальной оси }
            2: Horisont;      { отражение относительно горизонтальной оси }
            3: Invers;        { инверсия }
            else Break;       { выход из цикла и завершение программы }
      end;
  until cmd=0;
end.
```

Добавлю ещё два слова о константе **CSymbols**.

```
const CSymbols : string = '0+';
```

Напомню, что такие константы, сопровождаемые описанием типа, называют типизированными и применяют для размещения данных в памяти.

Теперь, говоря по школьному, мы прошли тему массивов и двинемся дальше. Но с массивами впредь не расстанемся, поскольку, ни одна мало-мальски сложная задача без них не решается. Всё только начинается!

Итоги

- Элементами массивов могут быть как простые, так и сложные типы данных, например, другие массивы или множества.
- Массив массивов называют двумерным массивом или матрицей.
- Для доступа к элементам матрицы необходимы два индекса: один – для столбца, другой – для строки.

А слабо?

А) По ходу строительства империи её бывшие границы — каналы — оказываются внутри новой страны и мешают перемещению граждан, — их лучше сровнять. Дополните программу P_49_1 с тем, чтобы она печатала эти бывшие границы. Или слабо?

Б) Измените внутреннее представление рекламного щита так, чтобы вместо булевых элементов использовать символы. Внесите необходимые изменения в программу и проверьте её.

В) В 38-й главе для нахождения простых чисел мы воспользовались множеством. К сожалению, мощность множеств в Паскале невелика (256), поэтому находить большие простые числа мы не могли. Но выход есть — это массив булевых переменных. По сути, это множество, судите сами. Объявим массив из 1000 элементов.

```
const CSize = 1000;
type  TBoolSet = array [1..CSize] of Boolean;
var   BS : TBoolSet;
```

Теперь условимся, что массив, заполненный значением **FALSE**, — это пустое множество. А если множество содержит числа **A** и **B**, то соответствующие им элементы массива **BS[A]** и **BS[B]** содержат **TRUE**. Тогда операции с этим придуманным нами типом-множеством можно выполнять так (справа показаны аналогичные операции с обычным множеством чисел **S**).

```
FillChar(BS, SizeOf(BS), false);    { S:= [] – пустое множество }
FillChar(BS, SizeOf(BS), true);     { S:= [1..1000] – полное множество }
BS[N]:= true;                       { S:= S + [N] – добавление элемента }
BS[N]:= false;                      { S:= S - [N] – удаление элемента }
if BS[N] then …                     { if N in S then … - проверка вхождения }
```

Воспользуйтесь таким массивом для поиска простых чисел в диапазоне от 1 до 1000.

Г) Садовая ограда. Вернувшись с курорта, фермер Лефт обнаружил на своем поле чудом выросший сад. Для сохранения деревьев он обнес его прямоугольной оградой. Пусть ширина и высота поля заданы константами CX и CY, пустые места обозначены точками, а деревья — звездочками. Засадите поле случайным образом и распечатайте его. Затем найдите левый верхний и правый нижний углы для ограды и постройте её символом решетки. Ограда должна охватывать деревья, но не выходить за пределы поля (то, что выходит за пределы, не строить). Распечатайте сад с оградой.

Глава 50
Неспортивные рекорды (записи)

Кушать подано!

Вообразите себя в гостях за столом, накрытым посудой и вкусностями. Только стол этот накрыт необычно: в одном углу — стопка тарелок, в другом — букет вилок, а там собраны все ножи. Неудобно, однако! Голодных гостей такие мелочи, ясно, не остановят, но согласитесь, — так накрывать не принято.

Или взять ранец со школярским добром: книгами, тетрадями, ручками и карандашами. Что, если нагрузить одного ученика всеми учебниками класса, другого — всеми тетрадями, а третьего — карандашами? Удобно им будет?

Однако ж, мы поступили именно так в одной из программ главы 41. Вспомните сортировку таблицы футбольного чемпионата. Там мы завели два массива: один — для набранных очков, другой — для названий команд. А затем в ходе сортировки меняли местами элементы этих массивов (программа P_41_3). Добавляя в таблицу чемпионата другие сведения о командах (забитые и пропущенные мячи, выигрыши, проигрыши и так далее), нам придётся заводить для них свои массивы. А потом возиться с перестановкой их элементов при сортировке, — тоска!

Нередко мы сталкиваемся с набором разнородных, но логически связанных предметов или данных, как в упомянутых выше случаях. И воспринимаем такие наборы как нечто целое, — это освобождает наш мозг от второстепенных деталей. Вот бы и в программировании найти средство логического соединения разнородных элементов!

Записи

В современных языках такое средство есть. В Паскале оно называется записью, по-английски — RECORD. «Рекорд» — знакомое словцо, не так ли? — оно имеет отношение к спорту. В самом деле, то, что мы называем спортивными рекордами, изначально было лишь записью в журнале регистрации спортивных достижений: кем, когда, где и сколько. Отсюда и пошло слово «рекорд».

Но вернемся к Паскалю. Итак, запись объединяет в единый набор логически связанные, но разнородные данные и дает этому набору имя. Такое объединение обозначают парой ключевых слов **RECORD-END** и размещают либо в секции объявления типов, либо в секции объявления переменных. Возьмем футбольную команду и соединим её название и набранные ею очки. Объявим для команды тип данных, который так и назовем — **Team** — «команда». А затем учредим две переменные этого типа, вот как это выглядит.

```
type  Team = record     { тип данных «команда» }
                 Aces  : integer;  { набранные очки }
                 Name  : string;   { название команды }
             end;

var   Team1, Team2 : Team;    { две переменных типа «команда» }
```

Может показаться, что между ключевыми словами **RECORD** и **END** объявлены переменные **Aces** и **Name**. Так ли это? И да, и нет. Нет, — потому, что в секции **TYPE** переменные не объявляют. Да, — потому, что внутри переменных **Team1** и **Team2** (объявленных чуть ниже в секции **VAR**) действительно «живут» переменные с именами **Aces** и **Name**. Только называются они теперь полями переменных **Team1** и **Team2**.

Как получить доступ к этим полям, спрятанным внутри переменных? Надо к имени переменной добавить так называемый квалификатор поля, или проще — имя этого поля. Вот так присваивают значения полям переменной **Team1**.

```
      Team1.Aces := 25;
      Team2.Name := 'Dinamo';
```

Как видите, квалификатор поля отделяют от имени переменной точкой. Поля — это «кусочки» более сложных переменных (в данном случае переменных **Team1** и **Team2**). Во всем остальном, кроме способа доступа, поля записей ничем не отличаются от обычных переменных.

Переменные, построенные на основе записей, называют структурными переменными или агрегатами. В чем их прелесть? В том, что обращаться с ними можно как с единым целым, например, копировать.

```
      Team2 := Team1;   { перенос всех полей из Team1 в Team2 }
```

Это существенно упрощает программы, в чем вы скоро убедитесь. А в сочетании с особыми типами данных — указателями — записи превращаются в «волшебные кирпичи», пригодные для возведения сложнейших структур, отражающих реальности нашего мира.

Второй тайм

Слышен свисток к началу второго тайма, вернемся на футбольное поле, точнее к программе P_41_3, сортирующей команды в порядке занятых ими мест. Сотворим новую версию этой программы P_50_1, заменив два массива (набранных очков и названий команд) одним. План игры на второй тайм таков. Объявим два типа данных: запись и массив записей. Первый из них — запись — будет содержать сведения об отдельной команде.

```
type  TTeam = record
                 mAces : integer;    { набранные очки }
                 mName : string;     { названия команд }
              end;
```

Напомню, что для улучшения читаемости программ мы условились начинать названия типов данных с буквы «Т». По этой причине тип данных «команда» (**Team**) стал называться **TTeam**. Подобное соглашение примем и для полей записей. Чтобы отличить их от прочих переменных, будем начинать имена полей с буквы «m» (от Member — «элемент», «участник»). Поэтому поля названы здесь как **mAces** и **mName**.

Второй из объявляемых типов данных — **TChamp** (Champ — «чемпионат») — представляет собой массив футбольных команд.

```
      TChamp = array [1..CSize] of TTeam;  { тип для массива команд }
```

Стало быть, каждый элемент этого типа содержит внутри себя два поля: число **mAces** и строку **mName**. Теперь можно объявить и переменную типа **TChamp**, то есть, массив команд.

```
var   Champ : TChamp;  { массив команд }
```

Доступ к элементам массива **Champ** осуществляется так.

```
Champ[i]          - i-й элемент массива, то есть i-я команда (тип TTeam)
Champ[i].mAces    - количество набранных очков i-й командой
Champ[i].mName    - название i-й команды
```

После всего сказанного рассмотрим программу P_50_1 в целом.

```
{ P_50_1 -  Футбольный чемпионат (версия 2) }
const CSize = 4; { количество команд }
      { объявление типов }
type  TTeam = record
                 mAces : integer;   { набранные очки }
                 mName : string;    { названия команд }
             end;
      TChamp = array [1..CSize] of TTeam;  { тип для массива команд }
var   Champ : TChamp;  { массив команд }
      { Процедура "пузырьковой" сортировки команд }
procedure BubbleSort(var arg: TChamp);
var   i, j : Integer;
      t : TTeam;  { для временного хранения при обмене }
begin
  for i:= 1 to CSize-1 do    { внешний цикл }
      for j:= 1 to CSize-i do  { внутренний цикл }
            { если текущий элемент меньше следующего ...}
            if arg[j].mAces < arg[j+1].mAces then begin
                  { то меняем местами соседние элементы }
                  t:= arg[j];          { временно запоминаем }
                  arg[j]:= arg[j+1];  { следующий -> в текущий }
                  arg[j+1]:= t;       { текущий -> в следующий }
            end;
end;
var i: integer;
begin       {--- Главная программа ---}
  { Вводим названия команд и набранные очки }
  for i:=1 to CSize do begin
      Write('Название команды: '); Readln(Champ[i].mName);
      Write('Набранные очки:   '); Readln(Champ[i].mAces);
  end;
  BubbleSort(Champ);  { сортируем }
  { Выводим результаты }
  Writeln('Итоги чемпионата:');
  Writeln('Место     Команда      Очки');
  for i:=1 to CSize do begin
      Writeln(i:3,' ':3, Champ[i].mName,
                          Champ[i].mAces:(20-Length(Champ[i].mName)) );
  end;
  Readln;
end.
```

Процедура сортировки заметно упростилась. Ещё бы! Ведь теперь мы работаем с одним массивом, а не с двумя. Для временного хранения элемента массива (при обмене) в процедуре объявлена переменная типа **TTeam**. А в прежнем решении для этого нужны были две переменные. Прочие изменения в программе невелики, хотя и существенны: вместо обращений к элементам массива мы обращаемся к полям этих элементов (эти места подчеркнуты).

Напомню смысл выражения для ширины поля при печати набранных очков.

```
      20-Length(Champ[i].mName)
```

Здесь учет длины названия команды обеспечивает ровную печать столбцов, стоящих правее.

Дополнительное время

Надеюсь, вы оценили приятный вкус структурных данных. Так продлим удовольствие, назначив после второго тайма дополнительное время. Соорудим третью версию программы, способную воспринимать и другие данные о командах, например, количество выигрышей и проигрышей. Для хранения этих новых сведений, очевидно, нужна память, то есть переменные. Но теперь обойдёмся без дополнительных массивов: объявим внутри записи **TTeam** ещё пару полей.

```
type  TTeam = record
                  mAces : integer;  { набранные очки }
                  mName : string;   { названия команд }
                  mWins : integer;  { количество выигрышей }
                  mFails: integer;  { количество проигрышей }
              end;
```

Внутрь каждого элемента массива **Champ** подселены ещё два поля, осталось лишь организовать ввод и вывод этих данных. Но к процедуре сортировки **BubbleSort** прикасаться уже не надо, — она не изменится! Поэтому в показанной ниже программе P_50_2 я не стал её повторять. Не стал я заниматься и обработкой поля **mFails** — количество проигрышей. Уверен, что вы и без меня справитесь с этим.

```
{ P_59_2 -  Футбольный чемпионат (версия 3) }
const CSize = 4; { количество команд }
      { объявление типов }
type  TTeam = record
                  mAces : integer;  { набранные очки }
                  mName : string;   { названия команд }
                  mWins : integer;  { количество выигрышей }
                  mFails: integer;  { количество проигрышей }
              end;
      TChamp = array [1..CSize] of TTeam;  { тип для массива команд }
var   Champ : TChamp;  { массив команд }

  { Процедура пузырьковой сортировки не изменилась }
procedure BubbleSort(var arg: TChamp);
. . .
end;
var i: integer;

begin       {--- Главная программа ---}
  { Вводим названия команд, набранные очки и прочие данные }
  for i:=1 to CSize do begin
      Write('Название команды: '); Readln(Champ[i].mName);
      Write('Набранные очки:   '); Readln(Champ[i].mAces);
      Write('Выигрышей: ');        Readln(Champ[i].mWins);
  end;
  { сортируем }
  BubbleSort(Champ);
  { Выводим результаты }
  Writeln('Итоги чемпионата:');
  Writeln('Место     Команда      Очки   Выигрышей');
  for i:=1 to CSize do begin
      Write(i:3,' ':3, Champ[i].mName,
            Champ[i].mAces:(20-Length(Champ[i].mName) ));
      Writeln(Champ[i].mWins:8);
  end;
  Readln;
end.
```

Напоследок отвечу на один вероятный вопрос. Поля записи объявлены мною в некотором порядке, существенно ли это? Ничуть! Поля могут объявляться в любой последовательности, — это не влияет на их обработку.

Итоги

- Для соединения разнородных, но связанных общим смыслом данных используют записи.
- Запись заключается в пару ключевых слов **RECORD-END**, между которыми перечисляются имена и типы полей, входящих в запись.
- На основе записей могут быть построены как одиночные переменные, так и массивы.
- Доступ к полям записей выполняется через имя переменной и имя поля, разделяемые точкой.

А слабо?

А) Дополните программу P_50_2 с тем, чтобы обработать все поля записи.

Б) Предложите структуру записи для полицейской базы данных. Какие данные следует, по вашему мнению, включить в неё?

В) Напишите программу для полицейской базы данных с применением записей. Обеспечьте ввод данных из файла, поиск по номеру и распечатку полей найденной записи.

Г) В текстовом файле тремя колонками представлены сведения о школьниках: фамилия, рост и вес. Ваша программа должна преобразовать его в три других файла, где эти же сведения отсортированы соответственно: 1) по фамилиям, 2) по росту и 3) по весу учеников.

Д) Домино. В этой игре используют 28 костяшек, каждая из которых содержит пару чисел от 0 до 6. Например: 0:0, 1:5, 6:6. Представьте костяшку записью, а игральный набор — массивом этих записей. Заполните массив костяшек и распечатайте его. «Смешайте» костяшки случайным образом и вновь распечатайте массив. Для удобства направьте распечатку в текстовый файл.

Е) Карты. Колода содержит 36 карт четырех мастей: трефы и пики — черные, а бубны и червы — красные. Относительная сила карты определяется числом от 6 до 14. Представьте карту записью, содержащей её масть, цвет и силу. Представьте колоду массивом записей, сформируйте полную колоду и распечатайте в текстовый файл. «Перетасуйте» колоду и вновь распечатайте в файл. При распечатке силу карт от 11 до 14 напечатайте их названиями: валет, дама, король, туз.

Глава 51
Указатели в море памяти

Птице в небе хорошо, а рыбе — в реке. Программы «живут» в оперативной памяти, — дайте им почуять себя там, как рыба в воде, и они обретут беспредельную мощь! Следующие главы продвинут нас к этой цели.

Погружение в оперативную память

Оперативная память содержит миллионы байтовых ячеек, — вы знаете об этом. Каждой ячейке назначен уникальный номер, иначе говоря — адрес. Уникальный — это значит, что все адреса разные, — так нумеруют дома на улицах и квартиры в домах. Первой ячейке памяти присвоен адрес 0, второй — 1 и так далее. Подобно тому, как почтальон находит дом по номеру, процессор обращается к данным по адресам ячеек, где они хранятся.

Прежде, чем «задышать», программа должна перекочевать с диска в оперативную память. Рассказать про это? С включением питания компьютера в дело вступает стартовая программа — загрузчик, прошитый в постоянной памяти материнской платы. Эта программка загружает с диска в оперативную память вашу любимую операционную систему; ритуал сопровождают загадочный скрип, мигание и попискивание. К тому моменту, когда на экране появляется знакомая картинка, часть памяти занимает операционная система. Дальше всё определяют капризы пользователя. Он волен загрузить одну или несколько программ, после чего их размещение в памяти может стать таким, как показано на рис. 115.

Распределением памяти под программы заведует операционная система. По мере запуска тех или иных приложений, система расселяет их в свободных областях памяти, а по завершении — выселяет, освобождая память для других целей. Этот механизм именуется динамическим распределением памяти.

Не применяйте слов, смысла которых не знаете. Что значит «динамический»? «Динамо» — греческое слово и означает силу, мощь. Оттого и полюбилось спортивным клубам, но к нашей теме это толкование не подходит. Сила порождает движение, потому «динамический» стали употреблять в смысле «подвижный», «быстрый». А программисты придали этому слову ещё один оттенок: «изменчивый», «непостоянный», разумея под этим изменчивое размещение в памяти данных и программ, — так будем понимать это слово и мы.

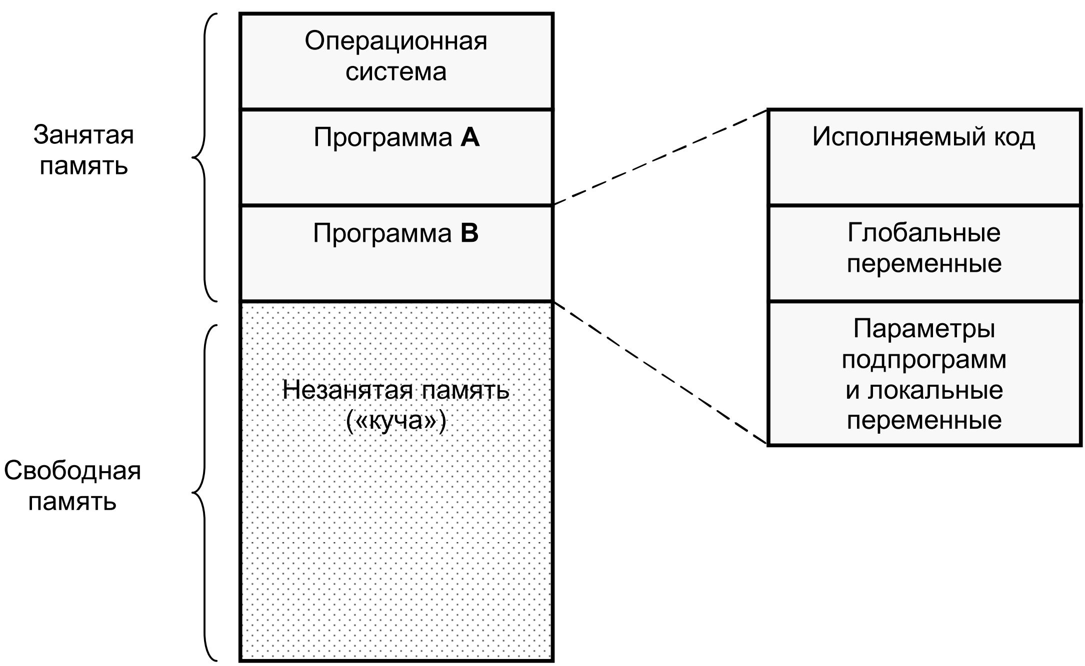


Рис. 115 – Распределение оперативной памяти

«Планировка» памяти

Обратимся к правой части рис. 115, где упрощенно показано распределение памяти внутри одной программы. Эта память делится на три части или секции (Section — «отделение»). Одна из них вмещает исполняемый код, то есть процедуры, функции и главную программу. Другая — секция данных — отведена для глобальных переменных, а третья — так называемый стек — для параметров процедур и локальных переменных. Сейчас надо усвоить лишь две простые вещи, а именно:

- все секции программы (как и программа в целом) имеют фиксированные, то есть постоянные размеры, определяемые при компиляции программы;
- все объекты программы – процедуры, функции, переменные – обладают своими «личными» адресами в оперативной памяти; эти адреса определяются при загрузке программы, и потому могут изменяться от одной загрузки к другой.

Рассмотрим пример. Пусть в программе объявлены четыре переменные.

```
var   B : Boolean;      C : char;   I : integer;      S : string;
```

После загрузки программы в оперативную память они «расселятся» в соседних ячейках памяти, начиная с некоторого начального адреса **N** так, как показано на рис. 116.

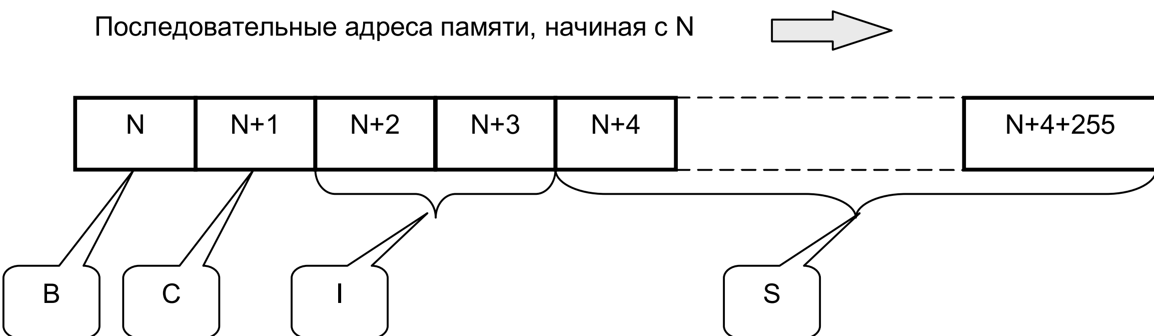


Рис. 116 – Размещение переменных в оперативной памяти

Первые ячейки этого участка памяти займут однобайтовые переменные булевого и символьного типа. В следующих двух байтах поселится целое число, а далее — в 256 байтах — строковая переменная. Подобная картина наблюдается и при размещении структурных переменных — записей; их поля занимают соседние ячейки. И хотя начальный адрес участка **N** может изменяться от загрузки к загрузке (его определяет операционная система), относительное размещение переменных в памяти остаётся тем же.

Указатели, первое знакомство

Отныне мы приступаем к освоению средств языка для работы с памятью. Овладев ими, вы откроете себе новые горизонты!

Начнем с нового для нас типа данных — указателя (по-английски — POINTER). Указатели могут хранить адреса переменных, процедур и функций. Нам интересны, прежде всего, указатели на переменные, рассмотрим пример обращения с таким указателем.

```
var   P : ^integer;     { указатель на целое }
      N : integer       { целое }
begin
      P:= @N            { указателю назначается адрес переменной N }
      P^:= 125;         { переменной присваивается значение через указатель }
      Writeln(N);       { 125 }
end.
```

В первой строчке объявлен указатель **P**. Это сделано специальным значком — стрелка вверх «**^**», — эта стрелка ставится перед именем типа, с которым будет работать указатель. В данном случае указатель **P** предназначен для хранения адресов переменных типа **INTEGER**.

Первый из исполняемых операторов

```
P:= @N
```

заносит в указатель **P** адрес переменной **N**. С этого момента указатель **P** ссылается на то место в памяти, где «живет» переменная **N**. Обратите внимание на «почтовую собачку» перед **N** — это операция взятия адреса. То же самое можно сделать функцией взятия адреса.

```
P:= Addr(N)
```

Следующий далее оператор программы

```
P^:= 125;
```

присвоит переменной **N** значение 125. Из чего это следует? Ведь переменной **N** в этом операторе нет! Всё дело в стрелочке «**^**», стоящей после указателя **P**, теперь она играет другую роль. Добавление стрелочки за указателем ведет к тому, что число 125 попадает в область памяти, на которую ссылается указатель **P**, то есть по месту жительства переменной **N**.

Таким образом, стрелочка за указателем — это операция разыменования, которая противоположна операции взятия адреса, она превращает указатель в переменную, на которую он ссылается. Поэтому следующие два оператора дают одинаковый результат.

```
Writeln(100+P^);  { 225 }
Writeln(100+N);   { 225 }
```

Введите рассмотренный пример в компьютер и проверьте его в действии.

Объявление указателей

Сколько типов данных способен придумать программист? Не сосчитать! И для каждого из них можно объявить свой тип указателя, например:

```
var   PI : ^integer;    { указатель на целое }
      PC : ^Char;       { указатель на символ }
      PS : ^String;     { указатель на строку }
```

Памятуя о нашей договоренности объявлять типы в секции **TYPE**, сделаем это для упомянутых типов-указателей.

```
type  PInt = ^Integer;        { тип указателя на целое }
      PChar = ^Char;          { тип указателя на символ }
      PString = ^String;      { тип указателя на строку }
```

Как всегда, имя объявляемого типа выбираем по вкусу. Здесь лучше придерживаться традиций, а они рекомендуют начинать названия типов-указателей с буквы «P» (от Pointer — «указатель»).

Объявив тип, можно объявить затем переменные этого типа, например:

```
var   p1, p2 : PInt;    { два указателя на целое }
      p3     : PChar;   { указатель на символ }
```

Копирование указателей, пустой указатель

Я сказал, что указатель содержит адрес переменной, стало быть, это число? Да, но не совсем обычное. Нас интересует не само это число, а лишь то, на что оно указывает.

Если нужны несколько указателей на одну и ту же переменную, указатели копируют, например:

```
      P1 := @X1;  { В указатель P1 заносится адрес переменной X1 }
      P2 := P1;   { Оба указателя содержат адрес переменной X1 }
```

Теперь оба указателя ссылаются на переменную **X1** (хотя сам по себе адрес переменной **X1** нас не интересует). Но копировать можно лишь указатели одного типа, — за этим соответствием следит компилятор.

А на что ссылается указатель, которому не присвоено значение? Как любая неинициализированная переменная, он содержит мусор и указывает «пальцем в небо». Для пометки временно не используемого указателя ему присваивают специальное значение **NIL**. Это зарезервированное слово языка — подобие нуля для чисел. Значение **NIL** можно присвоить указателю любого типа, например:

```
      p1 := nil;  { p1 – пустой указатель на целое }
      p3 := nil;  { p3 – пустой указатель на символ }
```

Указатели похожи на письма, а переменные — на дома, куда эти письма адресованы (рис. 117).

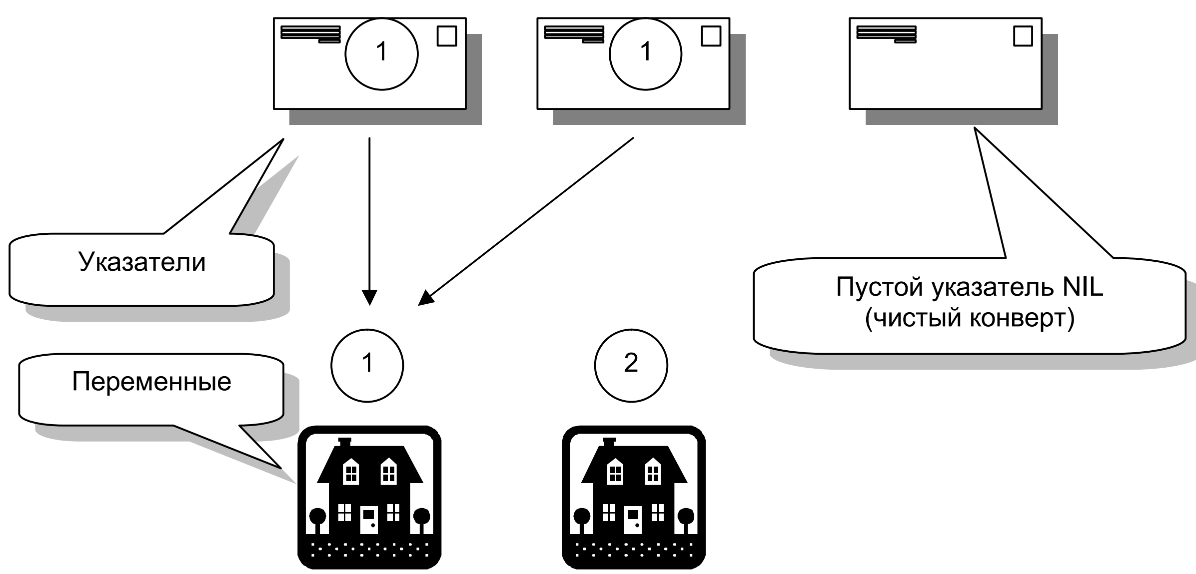


Рис. 117 – Указатели подобны письмам

Судя по рис. 117, жителям первого дома повезло, — им адресованы два письма. Третий конверт — без адреса, он пуст и подобен указателю, содержащему **NIL**.

Сравнение и проверка указателей

Поскольку указатели — это не обычные числа, их нельзя вычитать, складывать и сравнивать на «больше» или «меньше». Зато можно сравнивать на равенство и неравенство. В таком сравнении есть смысл: ведь если непустые указатели равны, то ссылаются на одну и ту же переменную. Вот примеры правильных сравнений.

```
    if p1=p2 then ...
    if p1<>p2 then ...
    if p3=nil then ...
```

Сравнением с **NIL** выясняется, свободен ли указатель или ссылается на что-то. Но значение **NIL** в указатель должен занести программист, само оно там не появится!

Проверить незанятость указателя можно как сравнением с **NIL**, так и функцией **Assigned**. Она принимает указатель любого типа, а возвращает булев результат. Вот примеры её применения.

```
      p1 := @X;   p3 := nil;
      Writeln (Assigned(p1));       { true }
      Writeln (Assigned(p3));       { false }
```

Функция **Assigned** возвращает **FALSE**, если указатель содержит **NIL**.

Разыменование указателей

Этим неуклюжим словом — разыменование — названа операция, обратная взятию адреса. Разыменование превращает указатель в переменную, на которую он ссылается. Операция обозначается следующей за указателем стрелкой вверх «^», вот пример:

```
      p1 := @X;              { назначение адреса указателю P1 }
      X := 25;
      Writeln (p1^);         { 25 }
      X := 100;
      Writeln (p1^);         { 100 }
```

Здесь показано, что с изменением переменной **X** меняется и значение **P1^**. Иначе говоря, если **P1=@X**, то **P1^=X** (а верно ли обратное?).

Итак, указатели дают ещё один способ доступа к переменным, к которым мы обращаемся по именам. В чем же выгода от указателей? — пока её не видно. Но, проявив немного терпения, вы изведаете всю их мощь.

Нетипичный указатель

Типы указателей соотносятся с типами данных, на которые они ссылаются. Но порой нужен универсальный указатель, способный ссылаться на что угодно. Такой указатель объявляют как **Pointer**, — указатели этого типа нельзя разыменовать, но можно сравнивать между собой и со значением **NIL**.

```
var   P1, P2 : pointer;       N : integer;       S : string;
begin
      P1:= @N;    P2:= @S;
      if P1=P2 then Writeln('Указатели совпадают');
      if P1<>nil then Writeln('Указатель не пустой');
end.
```

Впрочем, такой указатель можно привести к любому другому типу указателя (преобразовать тип указателя), и тогда возможно разыменование полученной конструкции, например:

```
type  PInt = ^integer;  { тип указателя на целое }
var   P : pointer;      N : integer;
      ...
      P:= @N;
      Writeln( PInt(P)^ );    { печатается значение N }
```

Примеры с указателями

Рассмотрим пару несложных программ, поясняющих работу указателей, испытайте их на своем компьютере.

```
{ P_51_1 – Указатели }
var A, B, C : integer;          { целые числа }
    p1, p2, p3 :^integer;      { указатели на целые числа }
begin
  { Присвоение значений переменным }
  A:= 10; B:= 20; C:= 30;
  { Последовательное переключение одного указателя на разные переменные }
  p1:= @A;  Writeln(p1^);
  p1:= @B;  Writeln(p1^);
  p1:= @C;  Writeln(p1^);
  { Настройка трех указателей на одну переменную }
  p1:=@B; p2:=p1; p3:=p1;
  Writeln(p1^:6, p2^:6, p3^:6);
  { Арифметические действия через указатели }
  C:= 2 * p1^;
  Writeln(C);  { C= 2 * B = 40 }
  Readln;
end.
```

Результат работы этой программы таков.

```
10
20
30
20    20    20
40
```

Здесь опять убеждаемся, что разыменованный указатель равнозначен переменной, на которую он ссылается. С ним выполняют те же действия, что и с переменной: ввод, вывод, арифметические операции и так далее.

В программе P_51_2 мы ещё раз увидим это, а вдобавок исследуем размеры указателей на переменные разных типов, — отличаются ли они?

```
{ P_51_2 - Указатели разных типов, размеры указателей }
type  PBool= ^boolean;  { Тип указателя на булевскую переменную }
      PInt = ^integer;  { Тип указателя на целое число }
      PStr = ^string;   { Тип указателя на строку }
var B : boolean;
    I : integer;
    S : string;
    pB : PBool;  { Указатель на булевскую переменную }
    pI : PInt;   { Указатель на целое число }
    pS : PStr;   { Указатель на строку }
begin
  { Настройка указателей на переменные }
  pB := @B; pI := @I; pS := @S;
  { Присвоение значений переменным через указатели }
  pB^ := true;
  pI^ := 10;
  pS^ := 'Hello!';
  { Распечатка значений переменных }
  Writeln(B:6, I:6, S:10);
  { Исследование размеров типов и указателей на них }
  Writeln('Boolean = ',SizeOf(Boolean):6, SizeOf(PBool):6);
  Writeln('Integer = ',SizeOf(integer):6, SizeOf(PInt ):6);
  Writeln('String  = ',SizeOf(String ):6, SizeOf(PStr ):6);
  Readln;
end.
```

Вот «продукция» этой программы.

```
true  10    Hello!
Boolean = 1      4
Integer = 2      4
String  = 256    4
```

Любопытны три последних строки. Они показывают, что размеры указателей на переменные всех типов одинаковы и для 32-разрядных систем составляют 4 байта (подобно тому, как размер конверта не зависит от размера дома, куда он адресован).

В следующей главе мы пожнем первые плоды от применения указателей, а пока подведем итоги.

Итоги

- Память компьютера – это последовательность ячеек, которым назначены уникальные адреса.
- Объекты программы – переменные, процедуры и функции – занимают ячейки памяти, адреса которых можно определить операцией взятия адреса @ или функцией **Addr**.
- Для хранения адресов применяют переменные особого типа – указатели. Каждому типу переменных соответствует свой тип указателя.
- Перед использованием указателя ему присваивают либо адрес переменной, либо пустое значение **NIL**.
- С указателями допустимы лишь три операции: копирование, сравнение и разыменование.
- Разыменованный указатель – это переменная, на которую он ссылается в данный момент; с ним можно поступать как с этой переменной.
- Указатели всех типов имеют одинаковый размер, который для 32-разрядных операционных систем составляет 4 байта.

А слабо?

А) Какие ошибки найдет компилятор в следующей программе? Объясните их.

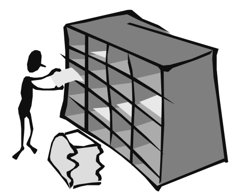
```
var   P1 : ^Integer;    P2 : ^String;
      N : Integer;      S : String;
begin
      P1 := @S;
      P2 := @N;
end.
```

Б) Будет ли работать следующая программа? В чём ошибки?

```
var   P1 : ^Integer;
begin
      P1 := 0;
      P1^ := 30;
      P1 := nil;  Writeln(P1^);
end.
```

В) Откройте программу P_51_1 и введите в окно обзора переменные **P1** и **P1^** (комбинацией *Ctrl+F7*). Выполняя программу по шагам, наблюдайте за переменными. Сделайте то же с программой P_51_2.

Глава 52
Динамические переменные



В предыдущей главе вы узнали о размещении данных в оперативной памяти и познакомились с указателями, хранящими адреса переменных. Это была присказка... Ведь самый жгучий вопрос остался без ответа — зачем нужны эти указатели?

Аппетит является к обеду

В программах для сортировки таблицы чемпионата и обработки классного журнала был заранее известен объемом данных. Действительно, количество клубов в чемпионате известно любому болельщику. Чуть сложнее с классным журналом, — ведь ученики приходят и уходят. Но, взяв размер массива учеников с некоторым разумным запасом, мы решаем и эту проблему.

Но так будет не всегда. Есть немало задач, где предугадать объем данных нельзя даже приблизительно. Вот, к примеру, полицейская база данных по угнанным автомобилям, каков должен быть её размер? Тысяча или миллион элементов? В спокойной стране достаточно будет десятка, а там, где угоняют каждый третий автомобиль… Ох! Лучше не спрашивайте! Можно, конечно, объявить массив с солидным запасом, но это породит ещё две проблемы. Во-первых, большая часть массива вероятней всего будет пустовать, — разумно ли транжирить память попусту? Второй случай ещё злее: в какой-то момент не хватит даже этого запаса, и программа «рухнет».

Безупречное решение — выделять данным ровно столько памяти, сколько нужно. То есть, создавать переменные, когда нам надо, и уничтожать их, когда потребность в них отпадает. Отличная мысль! Двинемся в этом направлении!

Одолжите памяти немножко!

Вернитесь к рисунку 51-й главы, где показано размещение программ в оперативной памяти. Большая часть этой памяти остаётся «не вспаханной», свободной. Куча — так принято её называть (по-английски — Heap). Операционная система распоряжается кучей по своему усмотрению, и всё же большие куски этой памяти простаивают без дела. Нельзя ли программе временно одолжить частичку? Оказывается, можно! Надо лишь освоить работу с указателями. В предыдущей главе мы применяли указатели на переменные, не видя в том особой пользы. Другое дело — участки памяти в куче, у которых нет имени. Здесь указатели — единственное средство для доступа к этим залежам.

Поскольку кучей заведует операционная система, за памятью обращаются к ней. Для этого в Паскале предусмотрено несколько процедур и функций, две из которых — процедуры **`New`** и **`Dispose`** — мы и рассмотрим.

Выделение памяти

Для получения кусочка памяти из кучи, вызывают процедуру **New** (что значит «новый»). Этой процедуре нужен лишь один параметр — указатель некоторого типа. Процедура бронирует в куче кусок соответствующего размера, и адрес этого куска помещает в указатель. Так рождается новая переменная в куче. Рассмотрим пример:

```
var   York : ^Integer;  { указатель на целое }
begin
      New(York);  { выделено два байта из кучи, адрес в переменной York }
      York^:=123;
      Writeln(York^);   { 123 }
end.
```

Здесь объявлен указатель на целое число по имени **York**. При выполнении процедуры **New(York)** из кучи выделяется 2 байта (для целого числа), и адрес этого кусочка попадает в указатель **York**. Чтобы воспользоваться выделенным участком как обычной переменной, указатель разыменовывают.

Спрашивается: откуда операционная система узнает объем запрашиваемой памяти (в данном случае 2 байта)? Об этом ей тихонько сообщит компилятор, которому известны размеры всех типов данных.

Освобождение памяти

Поработав с выделенным участком памяти, со временем вы можете отказаться от него — за ненадобностью. Тогда следует освободить его и вернуть в кучу, — как ни велика память, она не беспредельна, а брать взаймы ещё придется.

Освобождение кусочка памяти выполняется процедурой **Dispose** — «освободить». Ей, как и процедуре выделения памяти, нужен лишь один параметр — указатель на ранее выделенный участок памяти. Обратимся снова к примеру:

```
var   ps : ^string;     { указатель на строку }
begin
      New(ps);          { выделено 256 байтов из кучи }
      ps^:='Hello !';
      Writeln(ps^);     { Hello ! }
      Dispose(ps);      { возвращено в кучу 256 байтов }
end.
```

Здесь мы получили из кучи 256 байтов под строковую переменную, — вполне приличный кусок. А после — за ненадобностью — освободили эту память.

Переменные, порождаемые и исчезающие по мановению волшебника-программиста, называют динамическими. Но исполняет повеления программиста операционная система, — только она вправе хозяйничать в куче. Получив запрос на выделение памяти через процедуру **New**, система сообщит программе адрес выделенного участка и отметит его у себя как занятый. Теперь никто не посягнет на него, пока программа не освободит участок процедурой **Dispose**, или не завершится. По завершении программы все занятые ею участки памяти освобождаются системой автоматически.

Доступ к динамическим переменным возможен лишь через указатели, а они размещаются в секции данных или в стеке. Такие переменные (в обычном понимании, то есть с именами) называют статическими, поскольку их положение в памяти не изменяется, то есть, статично. В приведенных выше примерах статические указатели ссылались на динамические переменные.

Предупреждён – значит, вооружен

Работа с динамическими переменными таит ряд тонкостей, к ним надо привыкнуть, впитать в себя. Рассмотрим типичные ошибки.

Не инициализированный указатель

```
var   ps : ^string;
begin
  ps^:='Hello !'; { В указателе мусор – нельзя обращаться через него }
end.
```

Здесь память для переменной не выделена, и указатель содержит мусор, обращение через такой указатель к несуществующей переменной вызовет крах программы.

Обращение через пустой указатель

```
var   ps : ^string;
begin
  ps := nil;
  ps^:='Hello !'; { Указатель пуст – нельзя обращаться через него }
end.
```

Урок предыдущего примера справедлив и здесь: обращение к памяти через пустой указатель тоже вызовет крах программы.

Обращение к уничтоженной переменной (висячие ссылки)

```
var   ps : ^string;
begin
  New(ps);  ps^:='Hello !';   { Это нормально }
  Dispose(ps);
  ps^:='Bye !';   { Здесь ошибка, – переменной уже нет! }
end.
```

После освобождения памяти, занимаемой динамической переменной, обращаться к ней уже нельзя. Вот другой пример такой ошибки:

```
var   p1, p2 : ^string;
begin
  New(p1);
  p2 := p1;  { адрес динамической переменной копируется в другой указатель }
  Dispose(p2);    { Переменная освобождается через указатель p2 }
  p1^:='Hello !'; { Это ошибка, – переменной уже нет! }
  Dispose(p1);    { Это тоже ошибка ! }
end.
```

Здесь всё тоньше. Динамическая переменная создана через один указатель, и её адрес скопирован в другой. Теперь можно обращаться к переменной и освобождать её через любой из них. Но после освобождения все ссылки на переменную теряют силу! Обратиться к ней, или повторно освободить уже нельзя!

Утечка памяти

Эта ошибка преследует даже опытных программистов. Её трудно заметить, поскольку расплата наступает после исчерпания всей памяти, а этого можно и не дождаться. В действительности никакой утечки памяти не происходит, правильней назвать это захламлением памяти или потерей ссылки на переменную. Итак, смотрим:

```
var   p1, p2 : ^string;
begin
  New(p1);  New(p2);    { создаем две динамические переменные }
  p2 := p1;  { адрес первой переменной копируется во второй указатель }
  Dispose(p1); { Первую переменную освободить можно... }
  Dispose(p2); { ... а вторую – уже нельзя! }
end.
```

Здесь вторая переменная существует и после того, как указатель **p2** переключился на первую. Но связь с нею потеряна навсегда, эту переменную

нельзя даже освободить! Так объем свободной памяти в куче уменьшается, что и породило выражение «утечка памяти».

К счастью, после завершения программы операционная система сама освободит всю занятую программой память, включая потерянные участки. Не тревожьтесь — память вашего компьютера не пострадает!

В последующих главах мы построим несколько «боевых» программ с применением динамических переменных.

Итоги

- В задачах, где нельзя предсказать объем обрабатываемых данных, используют динамические переменные.
- Память для динамических переменных выделяется из кучи – области свободной памяти, которой заведует операционная система.
- У динамических переменных нет имен, поэтому доступ к ним возможен только через указатели.
- Для выделения памяти вызывают процедуру **New**, она помещает в указатель адрес созданной переменной.
- Когда надобность в динамической переменной отпадает, занятую ею память освобождают процедурой **Dispose**.
- Обращение с динамическими переменными требует аккуратности; остерегайтесь таких ошибок, как висячие ссылки и утечка памяти.

А слабо?

А) Сравните размеры переменных и указателей на них, воспользуйтесь для этого следующей программкой. Напишите нечто подобное для переменных других типов.

```
type  pb = ^byte; pw = ^word; pe = ^extended;
var   b : byte;   w : word;   e : extended;
begin
      Writeln(SizeOf(b):5, SizeOf(pb):5);
      Writeln(SizeOf(w):5, SizeOf(pw):5);
      Writeln(SizeOf(e):5, SizeOf(pe):5);
      Readln;
end.
```

Б) Найдите ошибки в следующей программе и объясните их.

```
var   p1, p2 : ^integer;
begin
      p1 := 10;
      p2^:= 20;
      New(p1);
      p2:= p1;
      p1^:= 'Привет!';
      Dispose(p1);
      Dispose(p2);
end.
```

Задачи на темы предыдущих глав

В) Ник обожал музыку. Но компьютерный музыкальный проигрыватель раздражал программиста, поскольку при случайном выборе мелодий повторял одни песни, напрочь забывая о других. Предположим, в списке 10 песен, но звучали только три из них: 3, 6, 5, 6, 3, 6, 5 и т.д.

Ник создал «справедливый» проигрыватель, выбирающий мелодии иначе. Все песни состояли в одном из двух списков: «белом» или «черном». Изначально все они были в «белом» списке, и очередная мелодия выбиралась из него случайно, а после проигрывания ставилась в конец «черного». Если в этот момент в «черном» списке состояла половина мелодий, то первая мелодия из «черного» списка возвращалась в «белый». Затем снова случайно выбиралась мелодия из «белого» списка. Так гарантировалось отсутствие повторов ранее проигранных песен в течение достаточно длительного времени. Создайте программу, генерирующую случайные числа (мелодии) в диапазоне от `1` до **`N`** представленным выше методом. Значение **`N`** не превышает 255.

Г) Распечатывая числовое множество, мы выводили все его элементы по одному, не заботясь об экономии бумаги или места на экране. Напишите экономную процедуру печати множества, учитывающую подряд идущие диапазоны чисел. Вот примеры желаемой распечатки:

```
1,5..255
0..200,210..255
0..255
2,5,7,10..20,30..40
```

Глава 53
Массив указателей

Мы научились создавать переменные по мере надобности — динамически. Эти переменные поселяются в куче, но доступны через указатели, что расположены в секции данных или в стеке — статической памяти программы. Взглянете на рис. 118, где каждая клеточка соответствует байту.

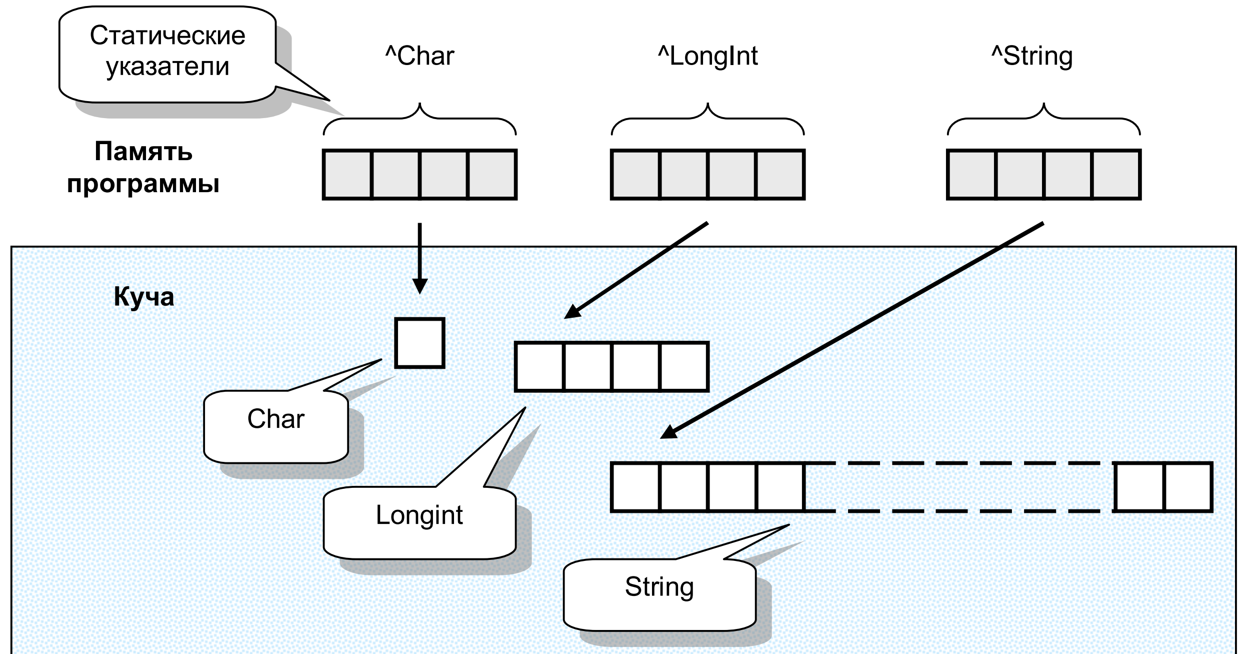


Рис. 118 – Указатели на динамические переменные

Очевидно, что размещая в куче мелкие переменные, никакой экономии статической памяти не получишь. Но чем крупнее переменная, тем выгоднее выселять её в кучу. Эта идея лежит в основе следующей переделки полицейской базы данных.

Базу данных – в кучу

Последние улучшения полицейской базы данных связаны с её сортировкой и быстрым двоичным поиском. Вдобавок мы выяснили, что номер автомобиля — не единственное, что нужно полицейским. Хорошо бы держать в базе данных фамилию владельца, его адрес, телефон и прочее. Всё это можно связать с номером автомобиля в структурном типе данных — записи.

```
type  TRec = record     { структура записи о владельце автомобиля }
        mNum : integer;         { номер авто – 2 байта }
        mFam : string[31];      { фамилия владельца – 32 байта }
        mAddr: string[63];      { адрес  – 64 байта }
        mTel : integer;         { телефон – 2 байта  }
      end;
```

Для экономии памяти отведем под фамилию и адрес укороченные строки: 31 байт — для фамилии и 63 — для адреса. Легко посчитать, что для размещения такой записи потребуется: 2+32+64+2 = 100 байтов (размер строки на единицу больше объявленной длины, вы помните об этом?).

Объявим массив из тысячи таких записей (на первое время хватит).

```
type  TBase = array [1..1000] of TRec;
```

И посчитаем размер массива, он составит 100 x 1000 = 100000 байтов. Ого! Сто тысяч — это больше, чем могут позволить вам иные компиляторы (Borland Pascal, например). Но если такой массив и поместится в памяти, немалая его часть по понятным причинам будет пустовать.

А если разместить эти увесистые записи в куче? Тогда в статической памяти останется лишь массив указателей на эти переменные, который займет в памяти 4 x 1000 = 4000 байтов, что вполне приемлемо. А куча? Её вместимость сопоставима с объемом памяти компьютера и исчисляется мегабайтами. Так мы убьем двух зайцев: сэкономим статическую память программы и выйдем за ограничения, налагаемые компилятором. Рис. 119 поясняет нашу задумку.

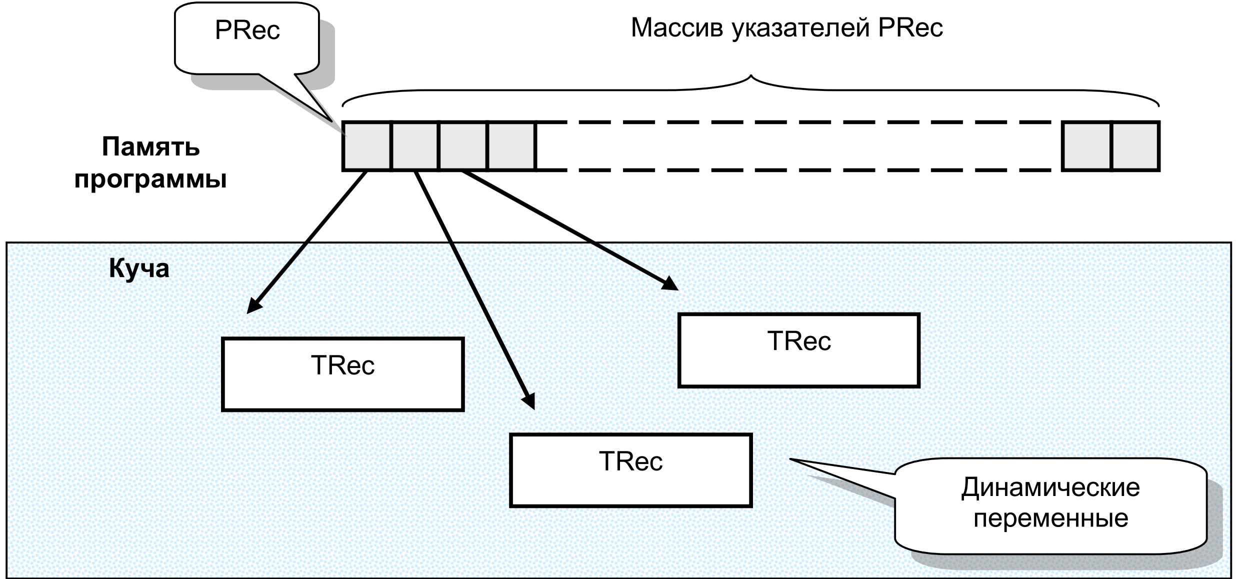


Рис. 119 – Массив указателей на переменные в куче

Кстати, вы заметили какой-либо порядок записей в куче? Переменные разбросаны там и сям без всякой системы. Кучей, как вы помните, заведует операционная система, и требовать какого-то порядка от неё неуместно.

Итак, исполняя задуманное, учредим ещё один тип данных — указатель на запись.

```
type  PRec = ^TRec;
```

Тогда база данных в статической памяти представится массивом.

```
type  TBase = array [1..1000] of PRec;
```

Уловив идею будущей программы, приступим к деталям. План дальнейших действий таков. Сначала создадим вспомогательную программу с двумя процедурами: одна из них — для ввода базы данных из текста в кучу, а другая — для распечатки этой базы. Эта программа будет фундаментом для следующей, где добавим процедуру сортировки записей.

Как обычно, данные будем вводить из текстового файла. Каждая его строка содержит номер автомобиля и фамилию владельца (прочие данные вы добавите позднее). Вот пример входного файла, где номер автомобиля и фамилия разделяются несколькими пробелами.

```
      6723  Иванов
      2199  Петров
```

Первый вариант программы P_53_1 перед вами, рассмотрим его.

```
{ P_53_1 - Ввод и вывод полицейской базы данных }
const CSize = 1000;     { Емкость базы данных }
type  TRec = record                 { Тип записи для базы данных }
            mNumber : integer;      { Номер авто }
            mFam    : string[31];   { Фамилия владельца }
      end;
      PRec = ^TRec;    { Тип указатель на запись }
      TBase = array[1..CSize] of PRec;    { Тип массив указателей }
var   DataBase : TBase;    { База данных - это массив указателей }
      Count:  integer;     { Фактическое количество записей в базе }
```

```
{ Чтение данных из текстового файла в базу данных }
function ReadData(var F : text): integer;
var   N : integer;      { номер авто }
      S : string;       { фамилия }
      P : PRec;         { временный указатель на запись }
      i : integer;      { счетчик записей }
begin
  Reset(F);  i:=0;
  while not Eof(F) and (i<CSize) do begin
    Inc(i);     { i+1 }
    { Читаем строку с номером и фамилией }
    Read(F, N);   Readln(F, S);
    { Удаляем пробелы в начале строки с фамилией }
    while (S[1]=' ') do Delete(S,1,1);
    New(P);    { Создаем новую запись в куче }
    { Заполняем поля записи }
    P^.mNumber := N;    P^.mFam := S;
    { Указатель на запись помещаем в массив }
    DataBase[i]:= P;
  end;
  Close(F);  ReadData:= i;
end;
procedure ExpoDataBase;      { Распечатка базы данных }
var  i : integer;
begin
  i:=1;
  { Пока индекс в пределах массива и указатель не пуст }
  while (i<=CSize) and Assigned(DataBase[i]) do begin
    { Печатаем номер, четыре пробела и фамилию }
    Writeln(DataBase[i]^.mNumber :6, '':4, DataBase[i]^.mFam);
    Inc(i);     { i+1 }
  end;
end;
var   F : text;    i : integer;
begin       {--- Главная программа ---}
  for i:= 1 to CSize do DataBase[i]:= nil;
  Assign(F,'P_50_1.in');
  Count:= ReadData(F);
  Writeln ('Всего записей: ',Count);
  ExpoDataBase;   Readln;
end.
```

Типы данных объявлены так, как уговорено выше. Предельный размер базы данных задан константой **CSize=1000**.

Функция **ReadData** читает строки текстового файла и помещает данные в кучу. После ввода номера автомобиля оператором **Read(F,N)** указатель чтения в файле остановится на первом пробеле за числом. Следующий оператор **Readln(F,S)** дочитает остаток строки. Так в переменной **S** окажется фамилия с пробелами в начале строки, — они потом удаляются.

Последующие операторы внутри функции **ReadData** создают динамическую переменную (запись), адрес которой содержится в указателе **P**. Затем поля записи заполняем номером автомобиля и фамилией владельца, после чего указатель **P** копируем в очередной элемент массива указателей. Эти действия можно записать короче — без вспомогательного указателя **P**, вот так.

```
New(DataBase[i]); { создаем переменную-запись, указатель в массиве }
DataBase[i]^.mNumber := N;    { копируем номер }
DataBase[i]^.mFam := S;       { и фамилию }
```

Но при пошаговой отладке удобнее пользоваться промежуточными переменными, что мы и сделали.

Теперь обратимся к процедуре **ExpoDataBase** — она распечатывает данные, размещенные в куче. Выражение **Assigned(DataBase[i])** в условии цикла **WHILE** равнозначно выражению **DataBase[i]<>NIL** и проверяет, ссылается ли указатель на динамическую переменную. Такая проверка исключает ошибку обращения через пустой указатель.

В главной программе заслуживает внимание строка, заполняющая пустым значением **NIL** все указатели массива. Ведь пока динамические переменные не созданы, указатели на них следует «заглушить» константой **NIL**.

```
      for i:= 1 to CSize do DataBase[i]:= nil;
```

То же самое делается проще и быстрее процедурой **FillChar**.

```
      FillChar(DataBase, SizeOf(DataBase), 0);
```

Но указатели — не обычные числа, возможно ли заполнять их нулями? Здесь проявляется универсальность процедуры **FillChar**, которая способна работать с данными любого типа. А ноль как раз и соответствует внутреннему представлению константы **NIL**.

Прежде, чем двинуться дальше, подготовьте файл с исходными данными и хорошенько проверьте работу программы P_53_1.

Сортировка массива указателей

Переходим ко второму этапу, где мы добавим процедуру сортировки массива указателей. Напомню, что в сортированном массиве работает быстрый двоичный поиск, — этим и привлекает нас сортировка. Вот программа P_53_2, где процедуры чтения и распечатки базы данных пропущены, — их следует взять из программы P_53_1.

```
{ P_53_2 - Сортировка полицейской базы данных }
const CSize = 1000;     { Максимальное количество записей в базе данных }
type  TRec = record                 { Тип записи для базы данных }
            mNumber : integer;      { Номер авто }
            mFam    : string[31];   { Фамилия владельца }
      end;
      PRec = ^TRec;    { Тип указатель на запись }
      TBase = array[1..CSize] of PRec;   { Тип массив указателей }
var   DataBase : TBase;    { База данных - это массив указателей }
      Count:  integer;     { Количество записей в базе }
      { Чтение данных из файла БД }
function ReadData(var F : text): integer;
{ Взять из P_53_1 }
end;
      { Распечатка БД }
procedure ExpoDataBase;
{ Взять из P_53_1 }
end;
      { FarmSort - «фермерская» сортировка }
procedure FarmSort(var arg: TBase; Right : integer);
var   L, R : Integer;        T : PRec;
begin
  for L := 1 to Right-1 do
    { Сдвигаем правый индекс влево }
    for R := Right downto L+1 do begin
     { Если левый элемент оказался больше правого,
       то меняем элементы местами }
      if arg[L]^.mNumber > arg[R]^.mNumber then begin
        { Перестановка элементов массива }
        T:= arg[L];  arg[L]:= arg[R]; arg[R]:= T;
      end;
    end;
end;
```

```
var   F : text;
begin      {--- Главная программа ---}
  FillChar(DataBase, SizeOf(DataBase), 0);
  Assign(F,'P_53_1.in');
  Count:= ReadData(F);       { Ввод данных и подсчет числа записей }
  Writeln('До сортировки: ');
  ExpoDataBase;  Readln;
  FarmSort(DataBase, Count);  { Сортировка }
  Writeln('После сортировки: ');
  ExpoDataBase;  Readln;
end.
```

Теперь направим внимание на процедуру сортировки. Для простоты я взял «фермерскую» сортировку — не самый быстрый алгоритм (смотрите главу 43). Отличия нынешнего варианта сортировки от первого невелики, их всего два.

Во-первых, сортируются не записи, а указатели на них. В 49-й главе нам довелось сортировать массив записей (помните футбольный чемпионат?), теперь сравним то решение с этим. Сортировка массива перемещает его элементы. Чем крупнее элемент, тем больше байтов передвигается. Очевидно, что четыре байта указателя двигаются быстрее, чем сотня байтов записи.

Здесь приходит на ум почтальон с пачкой открыток. Если открытки в пачке сложены случайно, почтальон станет бестолково бегать от дома к дому. Для ускорения дела надо что-то отсортировать: то ли дома в порядке сложенных открыток, то ли открытки по номерам домов. Как поступит почтальон? — догадайтесь сами. В нашем случае дома — это записи, а открытки — указатели на них.

Вторая особенность сортировки указателей в том, что обрабатывается не весь массив, а лишь непустые указатели. Обращаться к данным через пустые указатели недопустимо, — это верный способ «уронить» программу. Поэтому в процедуру передается граница сортируемой части через параметр **Right** — это правый индекс массива, равный фактическому количеству элементов в базе данных.

Итоги

- Используя массив указателей на динамические переменные, в памяти кучи можно поместить большой объем данных.
- При инициализации массив указателей заполняют значением **NIL**. Это предотвращает обращение к несуществующим динамическим переменным.
- Сортировка массива указателей похожа на сортировку других типов данных, но выполняется, как правило, быстрее. При этом сортируют лишь непустые указатели.

А слабо?

А) Дополните запись **TRec** полями с адресом владельца и его телефоном. Организуйте ввод и вывод этих данных (наряду с другими). Подготовьте надлежащий текстовый файл с исходными данными и проверьте работу этой версии программы.

Б) Напишите процедуру линейного поиска номера автомобиля в несортированном массиве указателей.

В) Напишите процедуру быстрого двоичного поиска в сортированном массиве указателей. Дополните вывод найденных данных, включая фамилию владельца и прочее.

Задачи на темы предыдущих глав

Г) «Глупый» винчестер (об «умном» вы узнаете в задаче 54-Д). Рассмотрим очень упрощенную модель винчестера, «шустрость» которого в основном определяется частотой вращения диска и скоростью перемещения головки чтения-записи. Время одного оборота диска примем за единицу - квант. За это время головка полностью читает или записывает одну дорожку. Количество дорожек на диске — 256, а нумерация идет с нуля (0...255). Время, необходимое для перемещения головки на соседнюю дорожку, тоже примем равным одному кванту.

Винчестером управляет контроллер, работающий куда быстрее механических узлов - диска и головки, поэтому издержками времени на его работу пренебрежем. Через некоторый интервал времени (таймаут) контроллер просматривает входную очередь, содержащую запросы на чтение или запись дорожек. Эта очередь формируется всеми запущенными программами. У нас это будет текстовый файл, каждая строка которого содержит по несколько чисел в диапазоне от 0 до 255 — это номера запрашиваемых дорожек. Пустая строка говорит об отсутствии запросов в текущий момент времени. Для первой строки файла сделаем исключение, поместив там лишь одно число - период (таймаут) просмотра этой очереди контроллером в квантах.

Контроллер «рулит» так. Прочитав список запросов (очередную строку файла), он перемещает их в свою внутреннюю очередь и далее обрабатывает её в том же порядке: смещает головку в нужную позицию и выполняет чтение-запись. Одновременно он следит за таймаутом, и, по истечении оного, читает следующую порцию входной очереди (то есть, строку файла). Ваша программа должна подсчитать общее время обработки запросов, заданных во входном файле (время измеряется в квантах).

Глава 54
Односвязные списки

Создав массив указателей на динамические переменные, мы сбросили в кучу уйму данных. Результат неплох, но где пределы совершенству? Как ни крути, размер массива ограничен, сколько указателей нам запасти? Побольше? Но тогда значительная часть массива будет пустовать. Одним словом, постоянный размер массива вяжет нас. Идеальное решение в том, чтобы отвести под данные столько памяти, сколько им требуется, — не больше и не меньше. Мы стремимся к этому решению и сейчас в шаге от цели.

Чудесное сочетание

Ключ к безупречному решению — в сочетании записи с указателем. Запись может содержать в своей обёртке разнородные элементы: числа, символы, строки, массивы и так далее. А если внедрить в неё указатель на другую запись этого же типа? Тогда мы сможем погрузить в кучу цепочку элементов так, как показано на рис. 120.

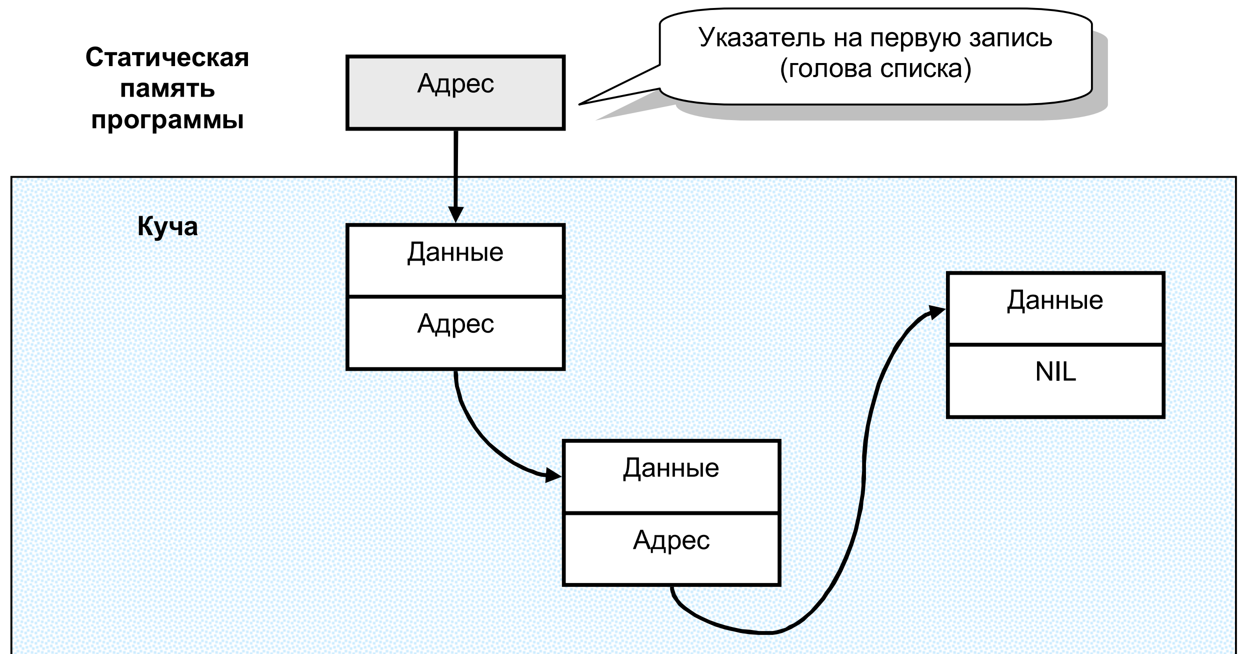


Рис. 120 – Связывание записей указателями

Здесь в кучу погружены три элемента данных (три записи), а в статической памяти программы торчит лишь «поплавок» — указатель на первый элемент этой цепочки. Все последующие записи сцеплены друг с другом указателями, встроенными в них же. Последняя запись содержит **NIL**, — как говорится, сколько веревочке не виться… Наряду с указателями, записи содержат, разумеется, и данные, необходимые для решаемой задачи. Такую структуру называют односвязным списком. Односвязным, — потому что элементы сцеплены лишь одной связью (при желании таких связей можно сделать больше). Мы будем называть эту динамическую структуру просто списком, по-английски — List.

В отличие от массива, где любой элемент доступен по индексу, в списке всё иначе. Поскольку указатели ведут в одном направлении, то возможен лишь последовательный доступ к элементам. Начав движение с головы списка, мы достигнем первого элемента. Затем по указателю, хранящемуся в нём, перескочим на второй элемент, со второго — на третий и так далее, пока не найдем нужный элемент или не уткнемся в конец списка.

Проблема курицы и яйца

Построим элемент односвязного списка для полицейской базы данных. Вот объявления нужных нам типов данных: записи и указателя на неё.

```
type
      PRec = ^TRec;     { Указатель на запись, - здесь TRec ещё не объявлен! }

      TRec = record                  { Тип записи для базы данных }
            mNumber : integer;       { Номер авто }
            mFam    : string[31];    { Фамилия владельца }
            mNext   : PRec;          { Указатель на следующую запись }
      end;
```

Тип **TRec** содержит, наряду с полезными полями **mNumber** и **mFam**, ещё одно служебное поле — указатель на следующую запись **mNext** (Next — «следующий»).

Друзья мои, обратите внимание на подчеркнутую строку, где нарушено важнейшее правило Паскаля! Вы знаете, что любой объект программы объявляют до его применения. Внутри записи объявлен указатель на неё же — поле **mNext**. Поэтому тип этого указателя **PRec** надо объявить раньше типа записи **TRec**, что и сделано в первой строке. Однако в этой первой строке применён тип записи **TRec**, который объявлен ниже! Круг замкнулся, как в задаче о курице и яйце, — что появилось раньше?

К счастью, в Паскале эта задача решена: указатель на любой тип данных можно объявлять раньше того типа, на который он ссылается. Это единственное исключение из правила. Значит, представленное выше объявление типов вполне законно.

Вяжем список

Возьмемся за постройку списка для полицейской БД. Начнем с простого: вставим в список несколько элементов данных с номерами автомобилей и фамилиями владельцев, а затем распечатаем список. Это делает программа P_54_1, рассмотрим её.

```
{ P_54_1 - Размещение данных в несортированном списке }
type  PRec = ^TRec;     { Тип указатель на запись }
      TRec = record                  { Тип записи для базы данных }
            mNumber : integer;       { Номер авто }
            mFam    : string[31];    { Фамилия владельца }
            mNext   : PRec;          { Указатель на следующую запись }
      end;
var   List : PRec;  { Указатель на начало списка (голова) }

      { Добавление в список записи об автомобиле }
procedure AddToList(aNumber: integer; const aFam : string);
var P : PRec;
begin
  New(P); { Создаем динамическую переменную-запись }
  { Размещаем данные в полях записи }
  P^.mNumber:= aNumber; P^.mFam:= aFam;
  P^.mNext:= List; { Цепляем предыдущую запись к новой записи }
  List:= P;       { Теперь голова указывает на новую запись }
end;
      { Распечатка списка }
procedure PrintList;
var P : PRec;
begin
  P:= List;
  while Assigned(P) do begin
      Writeln(P^.mNumber, '':3, P^.mFam);
      P:= P^.mNext;
  end;
end;

begin   { Главная программа }
  List:= nil;
  AddToList(10, 'Иванов');
  AddToList(20, 'Петров');
  AddToList(30, 'Сидоров');
  PrintList;
  Readln;
end.
```

Основу программы составляют процедуры вставки в список и его распечатки. В процедуре **AddToList** — добавить в список — первые строки вам знакомы: после создания динамической переменной **P^** данные копируются в её поля.

Обратите внимание на то, что указатель **P** — это локальная переменная, которая исчезнет после выхода из процедуры. И тогда, если не сохранить этот указатель, адрес новой переменной будет утерян, что приведет к утечке памяти. Поэтому после создания новой переменной **P^** адрес из головы списка **List** копируется в поле **mNext** вновь созданной записи, и адрес новой записи **P** помещается в голову списка. Вот эти операторы.

```
P^.mNext:= List;      { Цепляем предыдущую запись к новой записи }
List:= P;             { Теперь голова указывает на новую запись }
```

Всё просто, но если эти строки поменять местами, катастрофа неминуема!

Следующие рисунки показывают порядок наращивания списка. Здесь локальная переменная **P** обведена пунктиром, что отмечает её временный характер. Пунктирные стрелки с цифрами отмечают порядок выполняемых действий.

На рис. 121 и рис. 122 выполняется вставка первого элемента. В начальный момент, когда список ещё пуст, его голова — глобальная переменная **List** — содержит заглушку **NIL**, помещенную туда главной программой.

В результате присваивания оператором

```
P^.mNext:= List;
```

значение **NIL** попадает в поле новой переменной **P^.mNext** (после создания переменной это поле было не определено и содержало мусор).

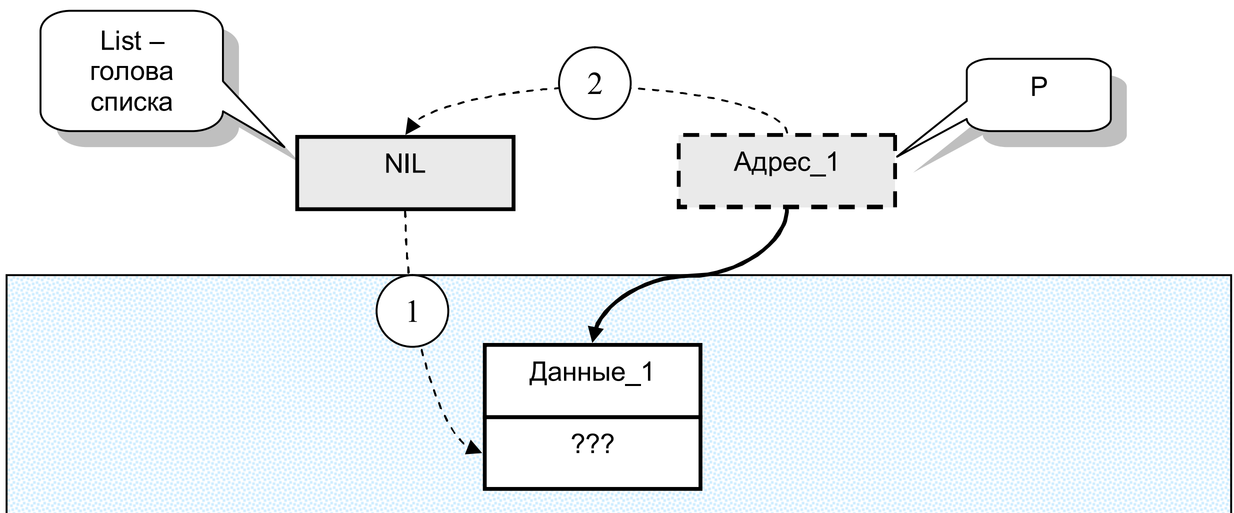


Рис. 121 – Состояние списка перед вставкой первого элемента

Следующий затем оператор

```
    List:= P;
```

сохраняет указатель на вновь созданную переменную в голове списка **List**. Теперь, перед выходом из процедуры, на эту переменную ссылаются уже два указателя (рис. 122). Но поскольку **P** — это локальная переменная, которая вскоре исчезнет, то после выхода из процедуры единственной ссылкой на первый элемент останется голова списка **List**.

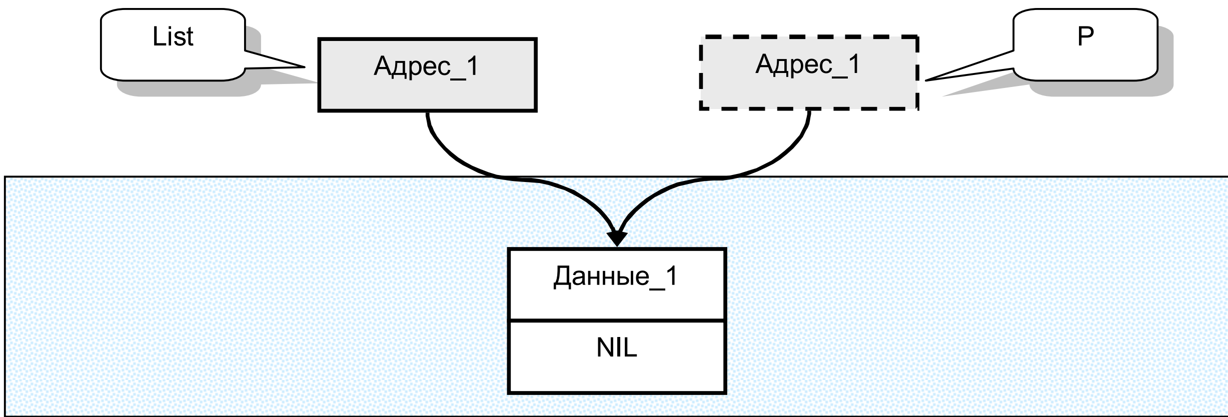


Рис. 122 – Состояние списка после вставки первого элемента

Следующая пара рисунков показывает вставку второго и последующих элементов. Сначала адрес бывшего первого элемента копируется из головы списка **List** в поле **mNext** вновь созданной переменной. Теперь доступ ко всем последующим элементам возможен через это поле. А следующий оператор ставит во главе списка вновь созданную переменную. Так новичок возглавляет список, оттесняя старожилов на последующие места.

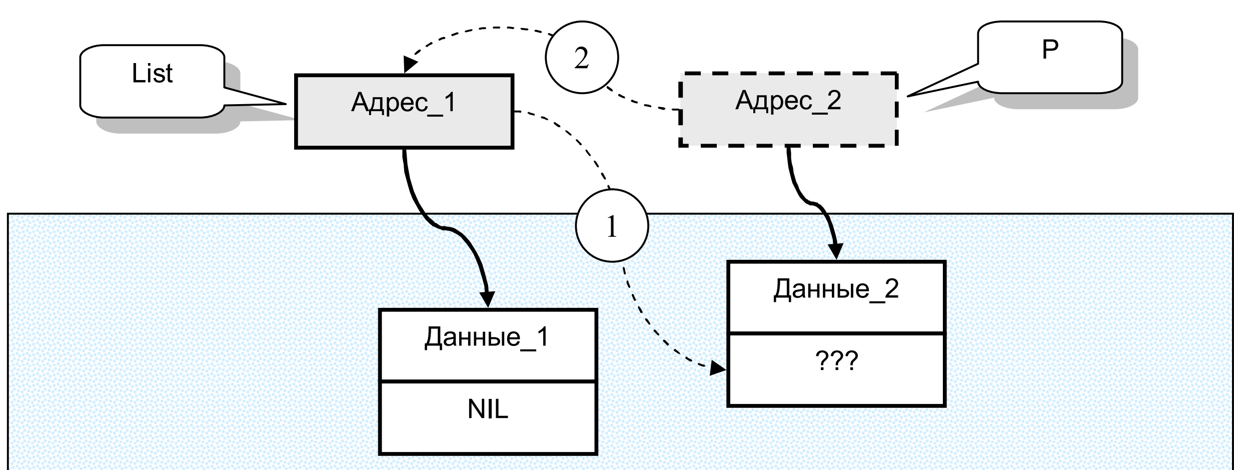


Рис. 123 – Состояние списка перед вставкой второго элемента

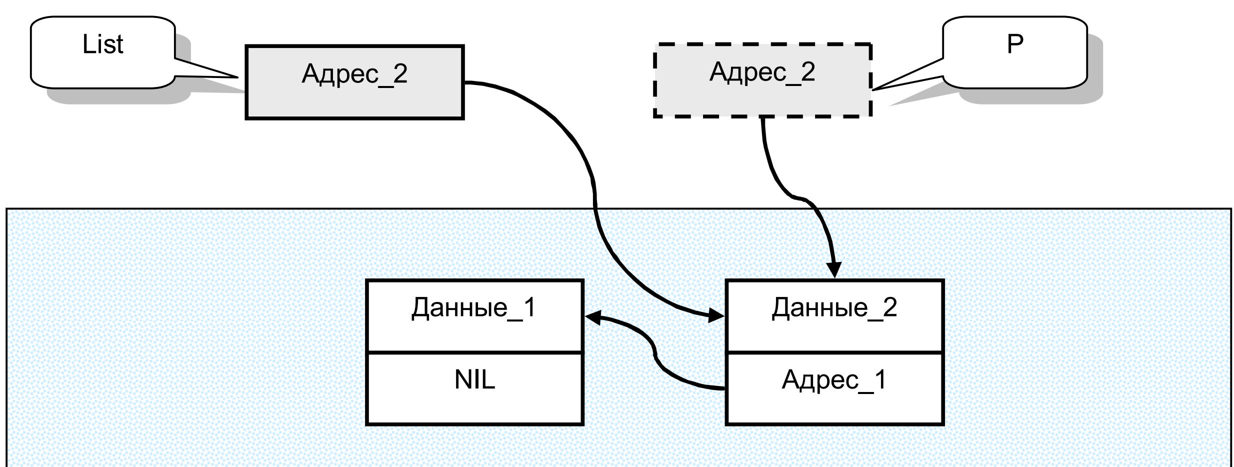


Рис. 124 – Состояние списка после вставки второго элемента

Распечатка списка

Теперь рассмотрим процедуру распечатки списка **PrintList**. Кто скажет, что она проста? Не проста, а очень проста! Подобные ей процедуры обработки списков строятся на основе цикла **While**. Скелет такой типовой процедуры показан ниже.

```
P:= List;
while Assigned(P) do begin
      { Здесь обрабатывается элемент списка }
      P:= P^.mNext;     { переход к следующему элементу }
end;
```

На входе в цикл указатель **P** загружается из головы списка. Внутри цикла, после обработки очередного элемента, переходим по указателю **mNext** к следующему. Напомню, что функция **Assigned(P)** равнозначна выражению **P<>NIL**. Таким образом, цикл завершится после обработки последнего элемента списка, поскольку его поле **mNext** содержит **NIL**. А если список пуст (голова содержит **NIL**), цикл не выполнится ни разу.

Программа P_54_1 покажет на экране следующий результат.

```
30 Сидоров
20 Петров
10 Иванов
```

Как и следовало ожидать, порядок элементов в списке оказался обратным порядку вставки в него. Этим можно воспользоваться для построения стека, что мы и сделаем в свое время.

Поиск в несортированном списке

Углубившись в списки, мы чуть не забыли о нашем пользователе — полицейском. А ведь ему придется искать в базе данных информацию об угнанных автомобилях. Дополнив предыдущую программу функцией поиска, получим представленную ниже программу P_54_2. Для экономии бумаги я не показал здесь процедуры вставки и распечатки списка.

```
{ P_54_2 - Размещение и поиск данных в несортированном списке }
type  PRec = ^TRec;     { Тип указатель на запись }
      TRec = record                 { Тип записи для базы данных }
            mNumber : integer;      { Номер авто }
            mFam    : string[31];   { Фамилия владельца }
            mNext   : PRec;         { Указатель на следующую запись }
      end;

var   List : PRec;  { Указатель на начало списка (голова) }

procedure AddToList(aNumber: integer; const aFam : string);
{--- взять из P_54_1 ---}
end;

procedure PrintList;
{--- взять из P_54_1 ---}
end;

      { Поиск в несортированном списке }
function Find(aNumber: integer): PRec;
var p : PRec;
begin
  p:= List; { Поиск начинаем с головы списка }
  { Продвигаемся по списку, пока не найдем нужный номер
    или не "упремся" в конец списка }
  while Assigned(p) and (P^.mNumber <> aNumber)  do p:= p^.mNext;
  Find:= p;  { результат поиска }
end;
```

```
var i, N : integer;   P : PRec;

begin   {--- Главная программа ---}
  List:= nil;
      { Заполним список случайными значениями номеров }
  for i:=1 to 20 do AddToList(100+Random(100), 'Деточкин');
  PrintList;      { Распечатка списка }
  repeat          { Цикл попыток поиска в списке }
      Write('Укажите номер авто = '); Readln(N);
      if N>0 then begin
            P:= Find(N);
            if Assigned(P)
                  then Writeln(P^.mNumber, '':3, P^.mFam)
                  else Writeln ('Не найдено!');
      end;
  until N=0
end.
```

Простенькая функция **Find** ищет в списке элемент с нужным номером автомобиля, и возвращает указатель на этот элемент списка. При неудачном исходе функция возвращает **NIL**.

В главной программе список заполняется записями со случайными номерами автомобилей. Так же «случайно» все владельцы оказались однофамильцами (придумайте тут что-нибудь!). Далее организован цикл с запросом искомого номера, поиском и печатью результата.

Сортированные списки

Итак, мы построили список и организовали в нём поиск данных. Довольны ли мы результатом? Для начала — неплохо, но если копнуть глубже…

Войдите в положение полицейского, перед которым мелькают сотни машин. Сколько из них числятся в угоне? Мизер! Но обработка каждого автомобиля вынуждает программу пройти по всему списку, — ведь он не сортирован! А опыт показал, что там, где порядок, всё ищется быстрее. Действительно, если расположить записи в порядке возрастания номеров, то просмотр списка можно прекращать при достижении номера больше искомого. При этом среднее число шагов поиска сократится вдвое.

Есть и другая причина для сортировки списка — желание распечатать данные в удобном порядке, например, по алфавиту. Честно говоря, список — это не лучшая структура для поиска и сортировки. Для этого лучше подходят другие динамические структуры — деревья, о которых можно узнать из литературы по алгоритмам. Но сейчас мы заняты списком и будем сортировать его.

Здесь не годятся алгоритмы для массивов. Ведь в списке нет индексов, к его элементам можно добраться лишь по цепочке ссылок. Зато у списка есть другое достоинство: для перестановки элементов не надо перемещать записи в памяти, — достаточно перенацелить указатели. Это легко сделать при вводе данных из файла, — так совмещается ввод списка с его сортировкой.

Вот пример с тремя записями (на рис. 125 и рис. 126 показаны лишь номера автомобилей). Предположим, что список содержит записи с номерами 20, 30 и 40, и мы вставляем запись с номером 35. После создания переменной **p^** надо найти предыдущий элемент, чтобы подцепить к нему новый. Обозначим указатель на такой элемент буквой **q**. Найдя элемент **q** (об этом чуть позже), вставку делаем на «раз и два» (рис. 125).

```
p^.mNext:=q^.mNext;     { связываем текущий со следующим }
q^.mNext:=p;            { связываем предыдущий с текущим }
```

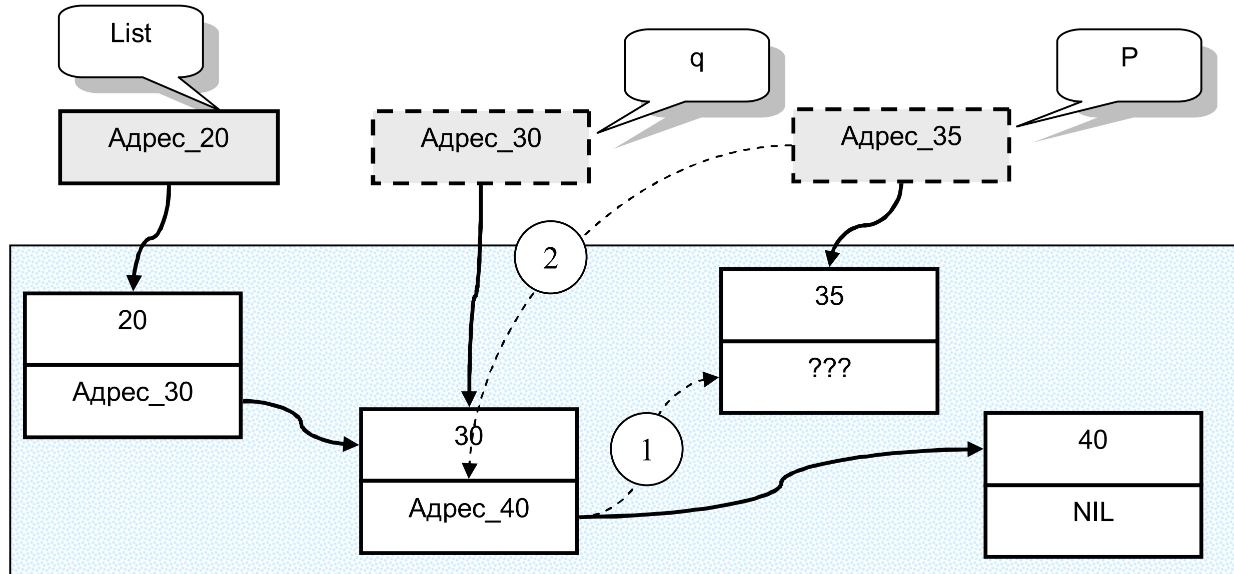


Рис. 125 – Состояние списка перед вставкой записи с номером 35

Состояние списка после вставки элемента показано на рис. 126.

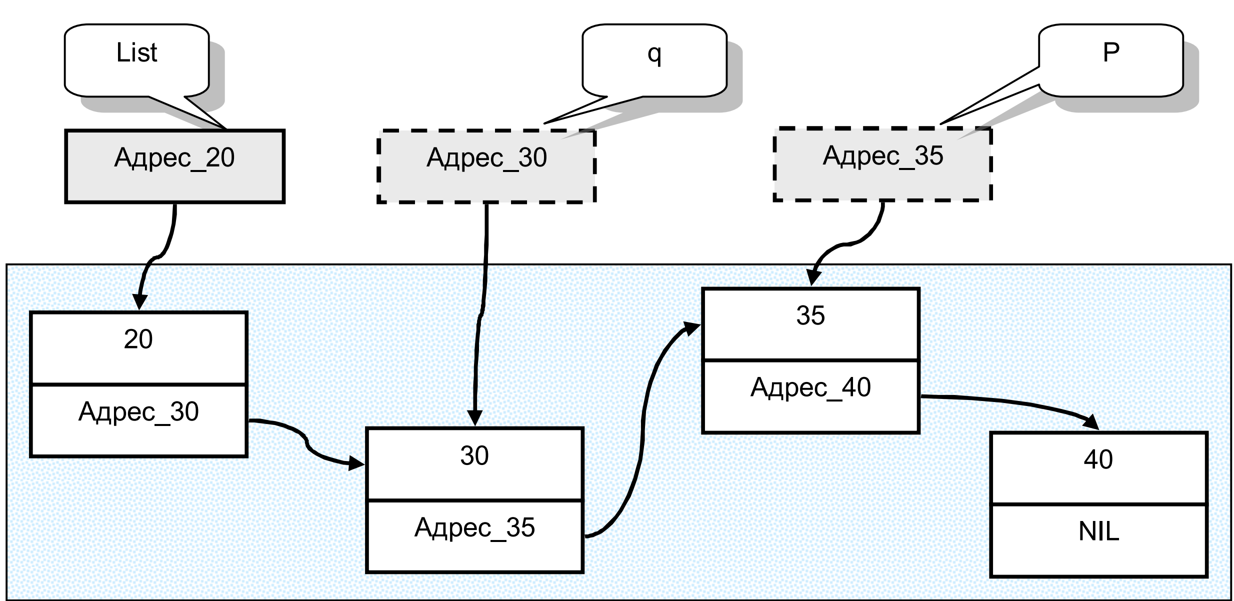


Рис. 126 Состояние списка после вставки записи с номером 35

Теперь рассмотрим два особых случая. Первый — когда список ещё пуст, и вставляемая запись будет первой и последней, здесь вставка делается так.

```
List:= p;  { если список пуст, голова указывает на новую запись }
```

Второй случай, — когда номер в первой записи окажется больше вставляемого. Тогда новую запись вставим в начало списка.

```
p^.mNext:=List;   { первый становится вторым }
List:=p;          { а текущий- первым }
```

Разобрав все три возможные ситуации при вставке в сортированный список, рассмотрим поиск указателя на предыдущий элемент **q**. Условие поиска таково: номер в элементе **q** должен быть меньше вставляемого, а следующий за ним элемент должен иметь номер, больше вставляемого, а иначе элемент **q** будет последним в списке. Поиск начнем, как всегда, с головы.

```
q:= List;  { Поиск места вставки начинаем с головы, здесь List<>nil }
while Assigned(q^.mNext) and (q^.mNext^.mNumber < aNumber)
      do q:=q^.mNext;
```

Здесь выражение **q^.mNext^.mNumber** соответствует номеру автомобиля в следующей за **q** записи.

Разобрав тонкости программы **P_54_3**, предъявлю её во всей красе.

```
{ P_54_3 - Размещение данных в сортированном списке }
type  PRec = ^TRec;     { Тип указатель на запись }
      TRec = record                 { Тип записи для базы данных }
            mNumber : integer;      { Номер авто }
            mFam    : string[31];   { Фамилия владельца }
            mNext   : PRec;         { Указатель на следующую запись }
      end;
var   List : PRec;  { Указатель на начало списка (голова) }
      { Размещение нового элемента в сортированном списке }
procedure AddToSortList(aNumber: integer; const aFam : string);
var p, q : PRec;
begin
  New(p); { Создаем динамическую переменную-запись }
  { Размещаем данные в полях записи }
  p^.mNumber:= aNumber; p^.mFam:= aFam;
  p^.mNext:=nil;
  { Если список пуст... }
  if not Assigned(List)
      then List:= p  { если список пуст, голова указывает на новую запись }
      else begin  { если список не пуст… }
            q:= List;  { Поиск места вставки начинаем с головы }
            { Двигаемся по списку, пока следующий элемент существует
              и его номер меньше вставляемого }
            while Assigned(q^.mNext) and (q^.mNext^.mNumber < aNumber)
                  do q:=q^.mNext;
            if q^.mNumber > aNumber then begin
                  { вставка на первое место }
                  p^.mNext:=List;   { первый становится вторым }
                  List:=p;          { а текущий - первым }
            end else begin
                  { вставка в середине или в конце списка }
                  p^.mNext:=q^.mNext;     { связываем текущий со следующим }
                  q^.mNext:=p;            { связываем предыдущий с текущим }
            end
      end
end;
```

```
  { Распечатка списка }
procedure PrintList;
{--- взять из P_54_1 ---}
end;

var i: integer;
begin   {--- Главная программа ---}
  List:= nil; { инициализация}
  { Заполнение списка }
  for i:=1 to 20 do AddToSortList(100+Random(100), 'Деточкин');
  { Распечатка списка }
  PrintList;
  Readln;
end.
```

Разумеется, что проверку этой программы я возлагаю на вас.

Поиск в сортированном списке

Создав функцию поиска номера в сортированном списке, поставим победную точку. Будем искать запись, для которой номер в следующей за ней записи больше искомого (если следующая запись существует). Это условие совпадает с условием поиска при вставке в сортированный список. Найдя такую запись и сравнив её номер с искомым, сформируем результат: если номер найден, возвращаем указатель на эту запись, а иначе — **NIL**. Всё это относится к программе P_54_4.

```
{ P_54_4 - Поиск данных в сортированном списке }
type  PRec = ^TRec;     { Тип указатель на запись }
      TRec = record                 { Тип записи для базы данных }
            mNumber : integer;      { Номер авто }
            mFam    : string[31];   { Фамилия владельца }
            mNext   : PRec;         { Указатель на следующую запись }
      end;
var   List : PRec;  { Указатель на начало списка (голова) }

      { Размещение нового элемента в сортированном списке }
procedure AddToSortList(aNumber: integer; const aFam : string);
{--- взять из P_54_1 ---}
end;
```

```
      { Распечатка списка }
procedure PrintList;
{--- взять из P_54_1 ---}
end;
      { Поиск в сортированном списке }
function Find(aNumber: integer): PRec;
var p : PRec;
begin
  p:= List;  { Поиск начинаем с головы }
  { Двигаемся по списку, пока следующий элемент существует
     и его номер не больше искомого }
  while Assigned(p) and Assigned(p^.mNext) and (p^.mNext^.mNumber <= aNumber)
      do p:=p^.mNext;
  { Если конец списка не достигнут и номер совпадает... }
  if Assigned(p) and (p^.mNumber = aNumber)
      then Find:= p     { то успешно! }
      else Find:= nil;  { а иначе не нашли }
end;

var   i, N : integer;   P : PRec;

begin   {--- Главная программа ---}
  List:= nil;
  for i:=1 to 20 do AddToSortList(100+Random(100), 'Фамилия, Имя');
  PrintList;      { Просмотр списка }
  repeat          { Цикл экспериментов по поиску в списке }
      Write('Укажите номер авто = '); Readln(N);
      if N>0 then begin
            P:= Find(N);
            if Assigned(P)
                  then Writeln(P^.mNumber, '':3, P^.mFam)
                  else Writeln ('Не найдено!');
      end;
  until N=0
end.
```

Итоги

- Указатель на любой тип данных можно объявлять раньше типа, на который он ссылается.
- Односвязный список – это простейшая динамическая структура, отводящая под данные столько памяти, сколько им требуется.

- Элементы списка – это записи, содержащие в числе прочих данных указатель на следующую запись в списке.
- Элементы списка помещают в кучу и связывают между собой внедренными в них указателями.
- Первый элемент доступен через голову списка (указатель в статической памяти программы). Остальные элементы доступны по цепочке указателей, встроенных в записи.
- Сортировку списка можно совместить с его вводом.

А слабо?

А) Напишите функцию для подсчета элементов списка; она должна принимать указатель на голову списка, а возвращать целое число.

Б) Начертите блок-схему вставки записи в сортированный список.

В) Напишите процедуру для удаления первого элемента списка. Или слабо?

Г) Напишите процедуру сортировки уже готового списка. Подсказка: последовательно извлекайте элементы из несортированного списка и вставляйте в сортированный (потребуется две головы для двух списков).

Задачи на темы предыдущих глав

Д) В задаче 53-Г была представлена модель «глупого» винчестера. «Умный» винчестер отличается организацией внутренней очереди и челночным движением головки, которая следует попеременно то от внутренней дорожки к внешней, то обратно, попутно выполняя все накопившиеся в очереди запросы. Направление движения переключается, когда в текущем направлении не остаётся запросов, поэтому головка редко достигает крайних дорожек.

Ваша программа должна подсчитать общее время обработки запросов «умным» контроллером для набора данных из входного файла, составленного по правилам для задачи 53-Г. Создайте несколько наборов таких данных и сравните время их обработки двумя типами контроллеров: «умным» и «глупым».

Подсказка: для организации внутренней очереди контроллера примените массив чисел (счётчиков). Каждый счётчик будет хранить текущее количество запросов для своей дорожки. При постановке запроса в очередь счётчик наращивается, а при извлечении (обработке) уменьшается.

Глава 55
Слова, слова, слова…

Односвязные списки подоспели как раз вовремя, — сейчас они поработают в необычном проекте.

Частотный анализ текста

Однажды разгорелся спор об известном романе «Тихий Дон», — некоторые литераторы усомнились в авторстве Михаила Шолохова. Их сомнения развеяли программисты, вычислившие частотные характеристики нескольких его произведений. Что это за характеристики такие?

Предположим, вы подсчитали, что слово «Паскаль» упомянуто в этой книге 150 раз, а всего в книге 10000 слов. Тогда относительная частота слова «Паскаль» в книге составит 150 / 10000 = 0,015 или 1,5%. Если найти частоту употребления других слов книги, и расположить эти результаты в некотором порядке, то получится картина, подобная отпечатку пальца. У разных авторов эти «отпечатки» разные, зато у одного автора в разных произведениях — очень похожи! Обработав таким частотным анализатором несколько книг Михаила Шолохова, специалисты сравнили результаты и обнаружили на романе «Тихий Дон» «пальчики» донского писателя.

Слово за слово

Итак, мы беремся за разработку слегка упрощенного частотного анализатора. Это опять тот случай, где заранее неизвестен объем обрабатываемых данных. В самом деле, определить приблизительное количество слов в тексте не так уж сложно: посчитаем их на одной странице и умножим на число страниц. Но сколько из этих слов несовпадающих, разных? Не слышу ответа!

Наша программа будет читать не романы, а текстовые файлы, — возьмем файл какой-либо из наших программ, и посчитаем в нём слова, составленные из латинских букв. Для упрощения программы русские слова считать не будем, и пропустим слова, состоящие из одной буквы. Зато примем в расчет слова с цифрами и знаками подчеркивания, например, такие.

```
Begin, NIL, P1, q2, Words_Count, _1_
```

Нам предстоит выудить из текста подходящие слова, перевести их в верхний регистр, отсортировать по алфавиту и пересчитать.

Структура записи

Накапливать слова будем в списке, а потому разработку программы начнем с конструирования надлежащей записи. Очевидно, что в ней надо предусмотреть

строку для слова и числовое поле для счетчика. Стало быть, структура элемента списка будет такой.

```
TRec = record     { Тип записи для подсчета слов }
      mWord  : string;        { Слово из текста – 256 байт }
      mCount : Longint;       { Счетчик слов – 4 байта }
      mNext  : PRec;          { Указатель на следующий – 4 байта }
end;
```

Сколько памяти займет один такой элемент? Сейчас посчитаем: **256+4+4=264** байта, — не так уж мало! Полагаю, что для слова достаточно и тридцати символов. Но, прежде, чем окончательно выбрать длину строки, открою небольшой секрет, — он касается выделения динамической памяти. Сколько бы памяти ни запросила программа, операционная система выделит кусочек, кратный восьми байтам. То есть, часть байтов в выделяемой порции может быть лишней. Значит, предпочтительный размер записи для динамических переменных кратен восьми байтам. В нашем случае размер записи можно уменьшить до 40 байтов, если объявить её так.

```
TRec = record     { Тип записи для подсчета слов }
      mWord  : string[31];    { Слово из текста – 32 байта }
      mCount : Longint;       { Счетчик слов – 4 байта }
      mNext  : PRec;          { Указатель на следующий – 4 байта }
end;
```

С одной стороны, число 40 кратно 8, а с другой стороны, 31-го символа для слова вполне достаточно.

Алгоритм

Теперь обсудим алгоритм обнаружения и обработки слов. В чем состоит эта обработка? Найдя выделенное слово в списке, нарастим его счетчик — поле **mCount**, а если слова в списке ещё нет, добавим запись с этим словом и счетчиком, равным единице.

Можно придумать много способов выборки слов из файла. Один из них — построчная обработка, когда каждую строку можно обработать так.

1. Перекодировать все символы строки в верхний регистр.
2. Удалить из начала строки все символы, которые не являются латинской буквой или подчеркиванием, и, если строка стала пустой, то завершить процедуру.
3. Выделить из строки очередное слово и удалить его из строки.
4. Искать слово в списке.
5. Если слово найдено, нарастить его счетчик, а иначе вставить в список запись со счетчиком, равным единице.

6. Прейти к пункту 2.

В перечисленных действиях нет ничего нового. В самом деле, обработка строк — дело привычное, так же, как поиск в сортированном списке и вставка в него данных. Таким образом, нам остаётся лишь собрать всё это воедино, что и сделано в программе P_55_1.

Процедуры этой программы сходны с аналогичными из полицейской базы данных, их отличает лишь порядок сортировки. Если там сортировка выполнялась по номерам автомобилей, то здесь — по словам.

```
{ P_55_1 - Частотный анализатор текста }

type
      PRec = ^TRec;      { Тип указатель на запись }
      TRec = record      { Тип записи для подсчета слов }
            mWord  : string[31];    { Слово из текста }
            mCount : Longint;       { Счетчик слов }
            mNext  : PRec;          { Указатель на следующий элемент }
      end;

var   List : PRec;  { Указатель на начало списка (голова) }

      { Поиск в сортированном списке }
function Find(const aWord: string): PRec;
var p: PRec;
begin
  p:= List;  { Поиск начинаем с головы }
  { Двигаемся по списку, пока следующий элемент существует
     и слово в нём меньше искомого }
  while Assigned(p) and Assigned(p^.mNext) and (p^.mNext^.mWord <= aWord)
      do p:=p^.mNext;
  { Если конец списка не достигнут и слово совпадает... }
  if Assigned(p) and (p^.mWord = aWord)
      then Find:= p    { ... то успешно! }
      else Find:= nil; { ... а иначе не нашли }
end;
```

```
      { Размещение нового элемента в сортированном списке слов }
procedure AddToSortList(const aWord : string);
var p, q : PRec;
begin
  New(p); { Создаем динамическую переменную-запись }
  { Размещаем данные в полях записи }
  p^.mCount:= 1; p^.mWord:= aWord; p^.mNext:=nil;
  { Если список пуст... }
  if not Assigned(List)
    then List:= p  { ...голова указывает на первую запись }
    else begin
           q:= List;  { Поиск места вставки начинаем с головы }
           { Двигаемся по списку, пока следующий элемент существует
             и его номер меньше вставляемого }
           while Assigned(q^.mNext) and (q^.mNext^.mWord < aWord)
              do q:=q^.mNext;
           if q^.mWord > aWord then begin
              { вставка на первое место }
              p^.mNext:=List;      { первый становится вторым }
              List:=p;             { а текущий- первым }
           end else begin
              { вставка в середине или в конце списка }
              p^.mNext:=q^.mNext;  { связываем текущий со следующим }
              q^.mNext:=p;         { связываем предыдущий с текущим }
           end
         end
end;
      { Добавление слова либо увеличение его счетчика }
procedure AddWord(const aWord : string);
var P : PRec;
begin
    P:= Find(aWord);
    if Assigned(p)
      then Inc(P^.mCount)
      else AddToSortList(aWord);
end;
      { Выделение и добавление слов из прочитанной строки }
procedure AddLine(S: string);
const CLetter = ['A'..'Z','_'];
      CDigits = ['0'..'9'];
var W : string;     i : integer;
```

```
begin
  { переводим все буквы строки в верхний регистр }
  for i:=1 to Length(S) do S[i]:= UpCase(S[i]);
  while Length(S)>0 do begin
    { удаляем все небуквы в начале строки }
    while (Length(S)>0) and not (S[1] in CLetter) do Delete(S,1,1);
    if Length(S)>0 then begin
      W:='';
      { копируем все буквы и цифры в слово W и удаляем из строки }
      while (Length(S)>0) and (S[1] in CLetter+CDigits) do begin
        W:= W+S[1];
        Delete(S,1,1);
      end;
      if Length(W)>1 then AddWord(W); { Если не буква, вставляем в список }
    end;
  end;
end;
      { Распечатка списка }
procedure PrintList(var F: text);
var P : PRec;
begin
  Rewrite(F);  P:= List;
  while Assigned(P) do begin
    Writeln(F, P^.mWord, '':(20-Length(P^.mWord)), P^.mCount:5 );
    P:= P^.mNext;
  end;
  Close(F);
end;
var S: string;  F: text;
begin   {--- Главная программа ---}
  List:= nil;
  Assign(F, 'P_55_1.pas');
  Reset(F);
  while not Eof(F) do begin
    Readln(F, S);
    AddLine(S);
  end;
  Close(F);   Assign(F, 'P_55_1.OUT');
  PrintList(F);   { Распечатка списка }
end.
```

Полагаю, что моих комментариев и вашего опыта хватит для понимания программы. Обязательно проверьте её. Вот результат исследования программой своего собственного текста (приведена только часть слов).

```
ADDLINE                 2
ADDTOSORTLIST           2
ADDWORD                 2
AND                     6
APPTYPE                 1
ASSIGN                  2
ASSIGNED                7
AWORD                  10
BEGIN                  14
CLETTER                 3
```

А слабо?

А) Дополните программу средствами для подсчета:

- общего количества слов в файле;
- общего количества разных слов.

Напишите для этого подобающие функции.

Б) Измените программу так, чтобы при распечатке списка выводилась относительная частота слов в процентах от общего их количества с двумя знаками после точки.

В) Создайте программу для подсчета в файле слов, составленных из одних и тех же символов: цифр и букв, например: «**end**» и «**deen», «121**» и «**221**». Для каждой такой группы слов программа должна напечатать перечень входящих в них символов (каждый символ — по разу) и количество обнаруженных слов этой группы, например, для приведенных выше слов будет напечатано:

```
1 2 – 2
d e n – 2
```

Подсказка: воспользуйтесь списком, записи которого включают в себя множество символов. Для слов, составленных из одинаковых символов, эти множества совпадают.

Глава 56
И снова очереди, и снова стеки…

Новейшая версия полицейской базы данных и частотный анализатор текста укрепили ваш опыт по части динамических переменных. И всё-таки, один момент нами упущен. Пока мы лишь добавляли данные в кучу, не утруждая себя её очисткой, — в решаемых задачах этого не требовалось. Но так будет не вечно, когда-то придется освобождать кучу от ненужных переменных. В этой главе мы рассмотрим два таких случая на примере знакомых нам очередей и стеков.

Шутить изволите?

Однажды, это было 1-го апреля, придворный программист Ник получил от приятеля странную «электрошку». Письмо содержало загадочный текст, очень похожий на программу, вот несколько его первых строк.

```
end.
  Close(F);
  while Pop(S) do Writeln(F, S);
  { Пока стек не пуст, извлекаем и печатаем строки }
  Assign(F, 'P_56_1.out'); Rewrite(F);
  { Открываем выходной файл }
  Close(F);
  end;
```

Приятель умолял Ника найти здесь ошибку.

Приняв во внимание 1-е апреля, Ник заподозрил розыгрыш и всмотрелся в эту абракадабру внимательней. Вскоре он сообразил, что строки в файле следуют в обратном порядке: последняя стоит первой, а первая — последней. Достойным ответом приятелю, — рассудил Ник, — будет правильный текст этой же программы. Но как переставить строки? Вручную, редактором текста? «Не царское это дело! — возмутился его разум, — пусть этим займется компьютер». И Ник написал программу для преобразования файла. Последуем за Ником.

Вы уже знакомы со стеком — временным хранилищем данных, из которого последний вставленный элемент извлекается первым (сообразно с дисциплиной LIFO). Стек — отличное средство для перестановки данных шиворот навыворот и задом наперед. Хранилищем данных в нашем первом стеке была строка, а хранимыми элементами — символы (загляните в главу 45). Скромные возможности того стека не помешали нам решить задачу о сортировочной горке.

Но чаще в стеке надо сохранять не символы, а крупные и сложные элементы данных. Так будет и в программе Ника, где элементом данных является строка. Как организовать стек из строк?

Вспомним порядок элементов при вставке их в список: последний элемент оттесняет соседей, становясь на первое место. А это значит, что, извлекая элементы от головы к хвосту списка, мы получим их в обратном порядке (рис. 127).

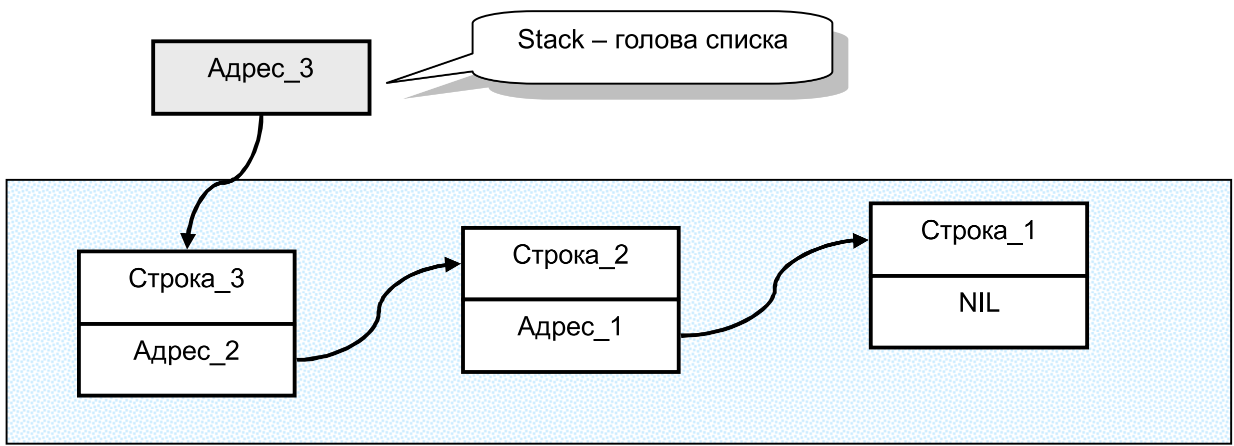


Рис. 127 – Порядок следования в стеке первых трех строк

После извлечения элемента из стека (в данном случае — строки) отпадет нужда в хранившей его динамической переменной. К чему засорять память? Ведь этот ценный ресурс нам ещё пригодится. Так давайте удалять из кучи ненужные динамические переменные.

Работу начнем, как обычно, с конструирования элемента списка. Этим элементом будет запись, включающая строку и указатель на следующую запись.

```
type  PRec = ^TRec;        { Тип указатель на запись }
      TRec = record        { Тип запись для хранения связанных строк }
        mStr  : string;    { хранимая строка }
        mNext : PRec;      { указатель на следующую запись }
      end;
```

Напомню, что со стеком выполняются, по меньшей мере, две операции: помещение в стек **PUSH**, и извлечение из стека **POP**. В нашем случае процедура записи в стек будет объявлена так.

```
procedure Push(const arg : string);
```

Аргументом процедуры является ссылка на строку, прочитанную из файла.

Теперь об извлечении из стека. Здесь надо получить не только строку, но и сигнал о состоянии стека: пуст он, или в нём ещё валяется что-то. Поэтому операцию извлечения из стека оформим булевой функцией.

```
function Pop(var arg : string): boolean;
```

Строка будет возвращаться через параметр **arg**, — это ссылка на переменную. Но, если функция вернет **FALSE**, это будет сигналом того, что стек пуст и строка не возвращена.

На этом закончим рассуждения и обратимся к программе P_56_1.

```
{ P_56_1 - перестановка строк файла }
type  PRec = ^TRec;        { Тип указатель на запись }
      TRec = record        { Тип запись для хранения связанных строк }
        mStr  : string;    { хранимая строка }
        mNext : PRec;      { указатель на следующую запись }
      end;
var Stack : PRec;  { Голова (вершина) стека }

      { Процедура размещения строки в стеке }
procedure Push(const arg : string);
var p : PRec;
begin
  New(p);               { создаем новую переменную-запись }
  p^.mStr:= arg;        { размещаем строку }
  { размещаем в голове стека }
  p^.mNext:= Stack;     { указатель на предыдущую запись }
  Stack:=p;             { текущая запись в голове стека }
end;

      { Процедура извлечения строки из стека }
function Pop(var arg : string): boolean;
var p : PRec;
begin
  Pop:= Assigned(Stack);  { Если стек не пуст, то TRUE }
   if Assigned(Stack) then begin { Если стек не пуст... }
    arg:= Stack^.mStr;       { извлекаем данные из головы стека }
    p:= Stack;               { временно копируем указатель на голову }
    Stack:= Stack^.mNext;    { переключаем голову на следующий элемент }
    Dispose(p);              { удаляем ненужный элемент }
  end
end;
```

```
var F : text;   S : string;
begin       {--- Главная программа ---}
  Stack:= nil;  { Инициализация стека пустым значением }
  { Открываем входной файл }
  Assign(F, 'P_56_1.pas'); Reset(F);

  { Пока не конец файла, читаем строки и помещаем в стек }
  while not Eof(F) do begin
      Readln(F, S);     Push(S);
  end;
  Close(F);
  { Открываем выходной файл }
  Assign(F, 'P_56_1.out'); Rewrite(F);
  { Пока стек не пуст, извлекаем и печатаем строки }
  while Pop(S) do Writeln(F, S);
  Close(F);
end.
```

Процедура **Push** повторяет процедуру вставки элемента в список. А в теле функции **Pop** посмотрите на подчеркнутые операторы. После извлечения строки ненужный теперь элемент стека уничтожается процедурой **Dispose(p)**, — так освобождается память. Но перед этим указатель на следующий элемент надо сохранить в голове списка, иначе мы потеряем ссылку на цепочку оставшихся элементов.

Изваяв программу, Ник испытал её волшебное действие на её собственном тексте. Каково же было его удивление, когда результат совпал с абракадаброй, полученной от приятеля! Вот такое чудесное совпадение!

Танцуют все!

Ох уж эти танцы… Задачу о танцевальном кружке мы решили в 45-й главе. Освежите её в памяти, поскольку новый вариант решения будет похож на первый.

Только теперь мы изменим имена мальчиков и девочек. В том варианте, как вы помните, дети носили однобуквенные имена, и мы представили их символами. Теперь же мы дадим им настоящие человеческие имена, но для этого и очередь организуем иначе. Как? Вы уже догадались: посредством односвязного списка (рис. 128).

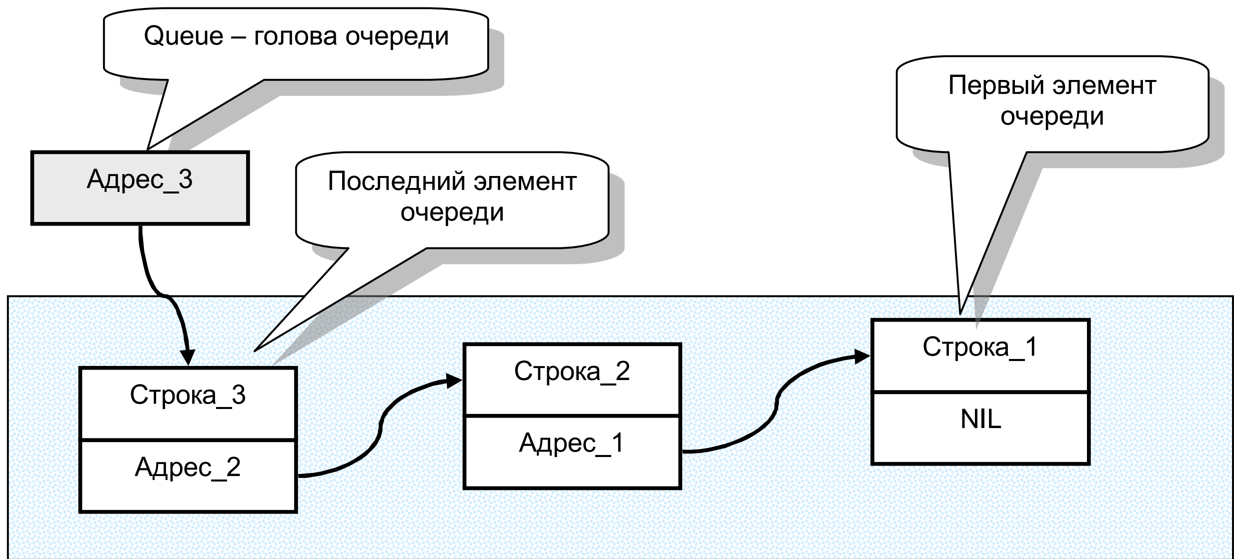


Рис. 128 – Размещение в очереди трех строк

Кажется, что этот рисунок совпадает с рисунком для стека. Так оно и есть. Только элементы теперь извлекаются в ином порядке. Первым элементом в очереди назначим тот, что в хвосте списка. Значит, по сравнению со стеком, в очереди всё наоборот: первым элементом очереди является последний элемент списка, и для доступа к нему придется пробежать по всей цепочке ссылок.

Это обстоятельство вынудит нас изменить процедуру удаления первого элемента очереди. Теперь перед удалением надо заполучить указатель на предпоследний элемент списка. В нём надлежит поставить заглушку **`NIL`**, чтобы отцепить первый элемент очереди (рис. 129). В этом состоит главная премудрость обращения с очередью строк.

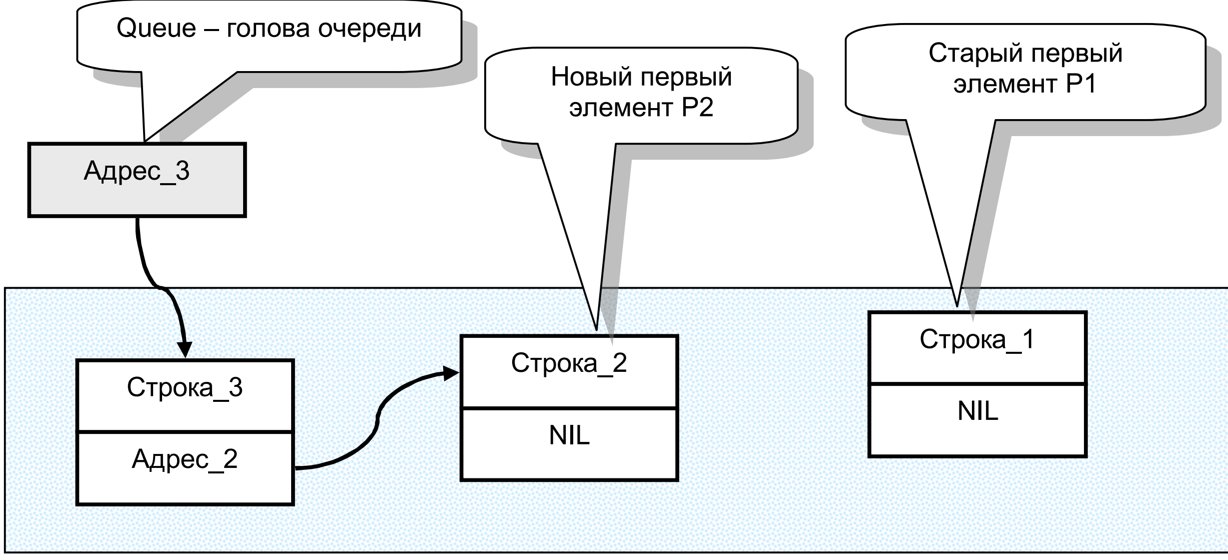


Рис. 129 – Отцепление первого элемента очереди

Так же как и для стека, для очереди надо запрограммировать, по меньшей мере, две операции: установку и извлечение из нее.

Процедуру установки в очередь **PutInQue** объявим так.

```
procedure PutInQue(var Que: PRec; const arg: string);
```

Два её параметра — это ссылка на очередь (на голову списка) и помещаемая в очередь строка.

Для извлечения из очереди потребуется уже не процедура, а функция, назовем её **GetFromQue**, и объявим так.

```
function GetFromQue(var Que: PRec; var arg: string): boolean;
```

Здесь опять заметно сходство со стеком: как только очередь окажется пустой, функция сообщит об этом, вернув значение **FALSE**. И тогда мы отвергнем возвращаемый через ссылку **arg** результат.

Осталось обсудить ещё одну мелочь: организацию входных данных с тем, чтобы отличать мальчиков от девочек. Имена детей поместим в файл «P_56_2.IN», а для различения мальчиков и девочек, впечатаем имена девочек с некоторым отступом (с одним или несколькими пробелами в начале строки). Вот пример такого входного файла.

```
Ваня
Петя
Гриша
   Маша
   Наташа
Коля
Семен
   Света
```

Теперь вы готовы рассмотреть программу P_56_2.

```
{ P_56_2 -  Запись в танцевальный кружок, версия 2 }

type  PRec = ^TRec;          { Тип указатель на запись }
      TRec = record          { Тип запись для хранения связанных строк }
        mStr  : string[31]; { хранимая строка (имя) }
        mNext : PRec;        { указатель на следующую запись }
      end;
  { Процедура размещения строки в очереди }
procedure PutInQue(var Que: PRec; const arg: string);
var p: PRec;
begin
  New(p);               { создаем новую переменную-запись }
  p^.mStr:= arg;        { размещаем строку }
  { размещаем указатель в голове очереди }
  p^.mNext:= Que;       { указатель на предыдущую запись }
  Que:=p;               { текущая запись в голове очереди }
end;
      { Извлечение строки из начала очереди (из конца списка) }
function GetFromQue(var Que: PRec; var arg: string): boolean;
var p1, p2: PRec;
begin
  GetFromQue:= Assigned(Que);
  if Assigned(Que) then begin
    { Поиск первого элемента очереди }
    p1:= Que;  p2:=p1;
    { если в очереди только один элемент, цикл не выполнится ни разу! }
    while Assigned(p1^.mNext) do begin
      p2:=p1;           { текущий }
      p1:=p1^.mNext;    { следующий }
    end;
    { теперь p1 указывает на первый элемент очереди,  а p2 – на второй
      (или на тот-же самый, если в очереди всего один элемент) }
    arg:= p1^.mStr;            { извлекаем данные }
    if p1=p2                   { если в очереди был один элемент... }
      then Que:= nil           { очередь стала пустой }
      else p2^.mNext:= nil;    { а иначе отцепляем первый элемент }
    Dispose(p1);               { освобождаем память первого элемента }
  end;
end;
```

```
var
    Boys   : PRec;     { очередь мальчиков }
    Girls  : PRec;     { очередь девочек }
    S1, S2 : String;   { строки с именами }
    Boy: boolean;      { признак чтения имени мальчика }
    F_In, F_Out : Text;  { входной и выходной файла }

begin       {--- Главная программа ---}
  { Очищаем очереди мальчиков и девочек }
  Boys  := nil ;    { очередь мальчиков }
  Girls := nil;     { очередь девочек }

  Assign(F_In, 'P_56_2.in');  Reset(F_In);
  Assign(F_Out,'P_56_2.out'); Rewrite(F_Out);

  { Цикл обработки входного потока }

  while not Eof(F_In) do begin
    Readln(F_In, S1);   { выбираем имя из входного потока }
    Boy:= S1[1]<>' ';   { строки с именами девочек начинаются с пробела! }
    while S1[1]=' ' do Delete(S1,1,1);
    if Boy
      then begin { если это мальчик...}
            if GetFromQue(Girls, S2)      { если в очереди есть девочка }
              then Writeln(F_Out,S1+' + '+S2) { пару -> в выходной поток }
              else PutInQue(Boys, S1);  { а иначе мальчика в очередь }
           end
      else begin { а если это девочка...}
            if GetFromQue(Boys, S2)      { если в очереди есть мальчик }
              then Writeln(F_Out,S2+' + '+S1) { пару -> в выходной поток }
              else PutInQue(Girls, S1);  { а иначе девочку в очередь }
           end
  end;
  Close(F_In);  Close(F_Out);
end.
```

Вот результат обработки входного файла:

```
Ваня + Маша
Петя + Наташа
Гриша + Света
```

Как видите, из 8 детей сформированы лишь три пары, и кто-то ожидает в сторонке.

Итоги

- Односвязные списки – это основа для построения разнообразных структур данных, в том числе очередей и стеков.
- Очереди и стеки, построенные на списках, могут хранить данные любых типов, при этом общий объем хранимых данных ограничивается лишь размером кучи.
- Не засоряйте кучу ненужными переменными, удаляйте их процедурой **Dispose**.

А слабо?

А) В Паскале есть встроенная функция по имени **MemAvail** (от Memory — «память», Available — «доступный»). Функция возвращает свободный на текущий момент объем памяти в куче.

Вставьте в процедуру **Push** и функцию **Pop** следующие операторы печати:

```
Writeln('Push :', MemAvail);
```

и

```
Writeln('Pop :', MemAvail);
```

Проследите таким образом за изменением объема свободной памяти в куче.

Б) В главе 45 было высказано предположение, что для записи в танцевальный кружок достаточно одной очереди. Покажите это, создав соответствующую программу. Чем потребуется дополнить механизм работы с очередью?

Глава 57
Графомания

Я чуть не забыл о придворном программисте Нике! В 49-й главе он решил задачу о минимальной сумме пошлин. Тогда же купцы уговорили его взяться за программу для поиска кратчайшего маршрута между двумя странами. Купцы страдали от пошлин и хотели сократить свои расходы на границах. Ник принял заказ и впал в размышления.

На рис. 130 показан вид из космоса на континент, где проживал Ник. Тамошние страны именовались, как вы помните, латинскими буквами.

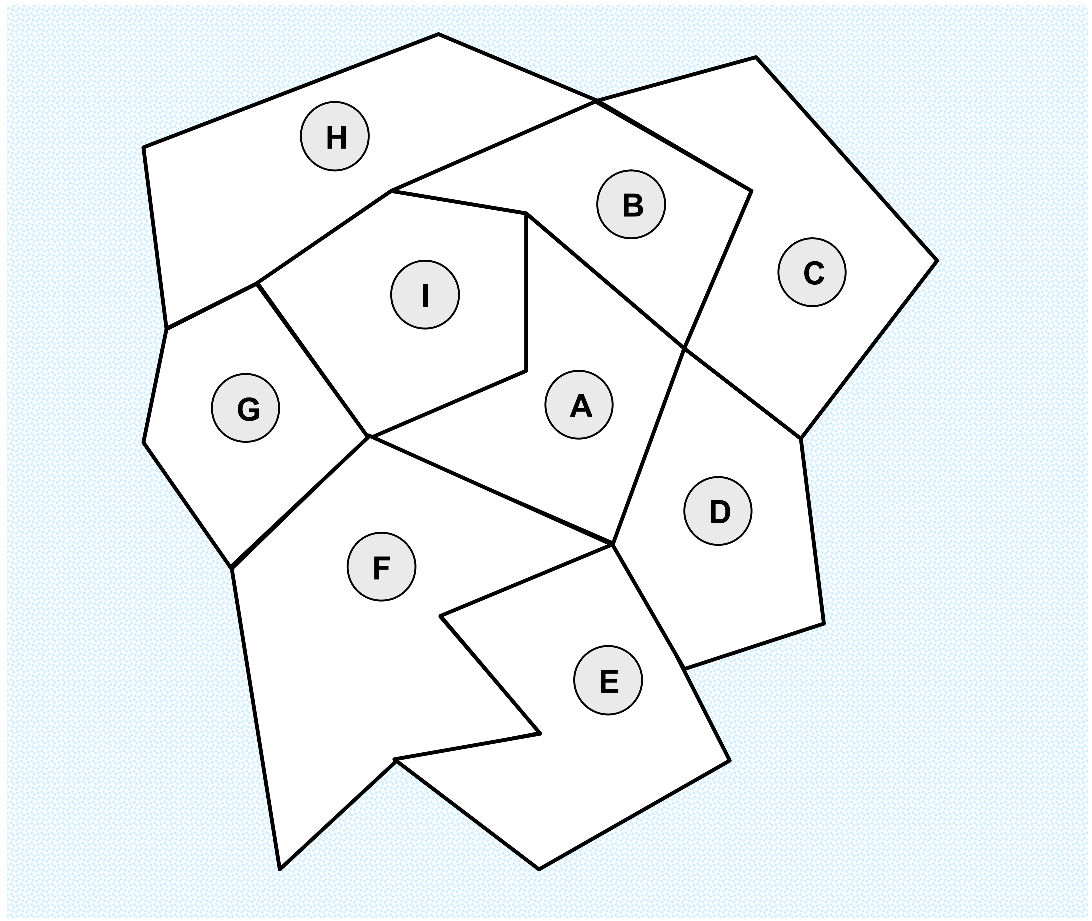


Рис. 130 – Карта континента

Программа, что создал Ник в 38-й главе, превратила эту карту в следующий файл.

```
A B D F I
B A C I H
C B D
D A C E
E D F
F A E G
G H I F
H G I B
I A B G H
```

В каждой строке файла представлены соседи одной страны: первый символ — это название самой страны, а последующие — её соседи, перечисленные в произвольном порядке. В любом порядке могут следовать и сами строки, — от этого карта не изменится, согласны? Итак, этот файл содержал данные для поиска кратчайшего маршрута.

Данные были, только решение куда-то ускользало. Вот берег озера, где спрятался Ник. Его рука в который раз царапает на мокром песке одну и ту же картинку (рис. 131).

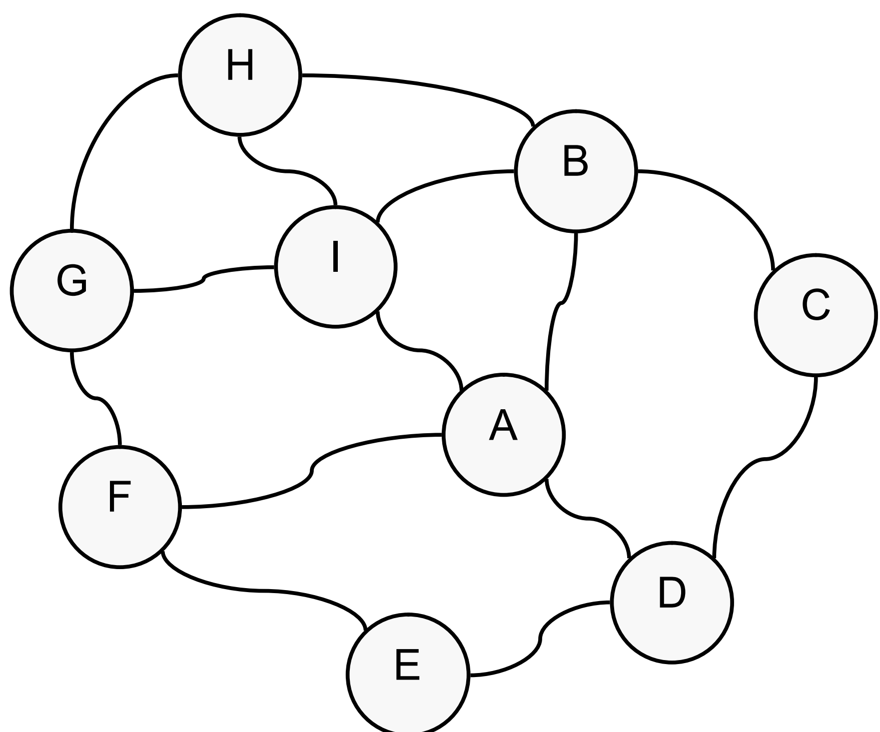


Рис. 131 – Картинка на мокром песке

Здесь вместо разделяющих царства границ, Ник нацарапал соединяющие их дороги. «Вот по этим дорогам поедут купцы, — размышлял он, — но как именно?». Озарение явилось внезапно. «Постой-ка, мне знакома эта картинка! Неужто граф? Я что-то читал о них, надо бы вспомнить!». Оставим ненадолго озаренного Ника, и выясним, что это за штука такая — граф?

Видимое представление графа

Слово «граф» намекает на рисование, графику. Но программисты и математики признают графом не любую картинку. Граф для них — это сеть связанных между собой объектов. Объекты называют вершинами или узлами графа, а связи между ними — ребрами или дугами. В англоязычной литературе используют термины Node — узел, и Link — связь.

Вот знакомая картинка — схема московского метро (Рис. 132), это пример графа. Здесь станции являются узлами графа, а пути между ними — ребрами. Соседние узлы графа называют смежными. Кстати, нырнувший в метро пассажир решает ту же задачу, что и Ник: ищет кратчайший путь между двумя станциями.

Рис. 132 – Схема московского метрополитена – это граф

А вот ещё примеры графов: карта автомобильных дорог, дерево родственных связей, электрическая схема. Вы можете придумать свои примеры. Или взять нацарапанный Ником рисунок, где узлами являются страны, а ребрами — дороги, их соединяющие.

Мы рассмотрели внешнее, видимое представлении графа, теперь обратимся к его внутреннему представлению в памяти компьютера.

Внутреннее представление графа

С внутренним представлением графа вы отчасти знакомы. Не удивляйтесь, ведь односвязный список — это тоже граф. Элементы списка — это узлы графа, а связи между элементами — это ребра. И хотя связь между узлами списка однонаправленная, такие графы тоже имеют право на жизнь. Разве нет дорог с односторонним движением?

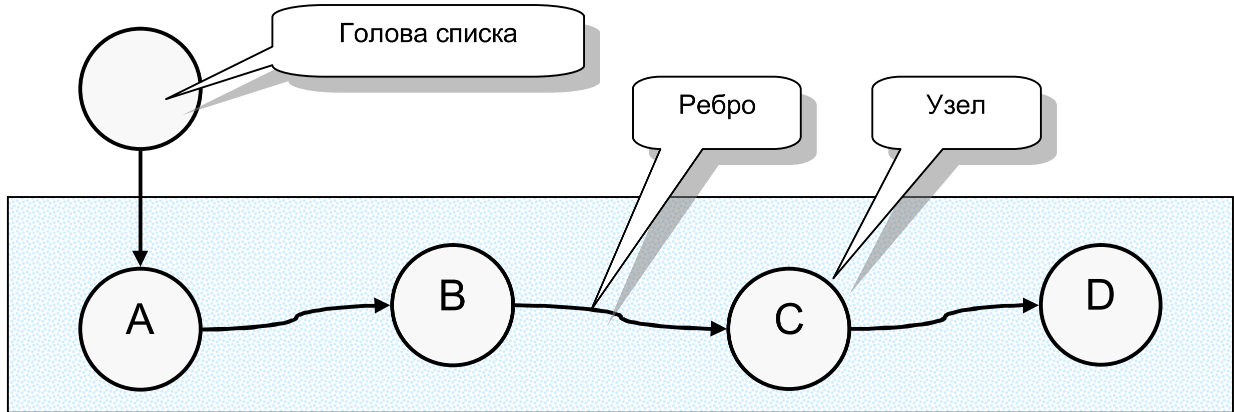


Рис. 133 – Односвязный список – это разновидность графа

Годится ли такой список для представления графа, нацарапанного Ником? Рисунок на песке очевидно сложнее списка, — в нём много связей между узлами. К тому же связи на схеме Ника двунаправленные, ведь по дорогам можно ехать в обе стороны. Для представления такого графа требуется что-то похитрее списка. Но в этой замысловатой конструкции найдется место и односвязным спискам.

Приступим к постройке нужного нам графа, и начнем с узла. Представим его, как обычно, записью. Что будет полезной нагрузкой узла? Пока достаточно хранить в записи лишь имя страны, то есть один символ. По мере необходимости, мы добавим в запись и другие поля.

Теперь о связях. Очевидно, что их представим указателями. Но сколько их потребуется? Ведь из разных узлов исходит разное количество связей (рис. 131). Я предлагаю поместить в каждом узле список его связей с соседями. Неслабый получается узелок — с собственным списком внутри! Устройство этого списка связей мы обсудим чуть позже.

Но и это не всё. Поскольку узлы графа погружаются в кучу, нужно средство для доступа к ним. Вы знаете его — это односвязный список. Значит, внутри каждого узла нужен указатель **`mNext`** для включения узла в этот вспомогательный список. В итоге наших размышлений проясняется внутреннее представление графа, показанное на рис. 134.

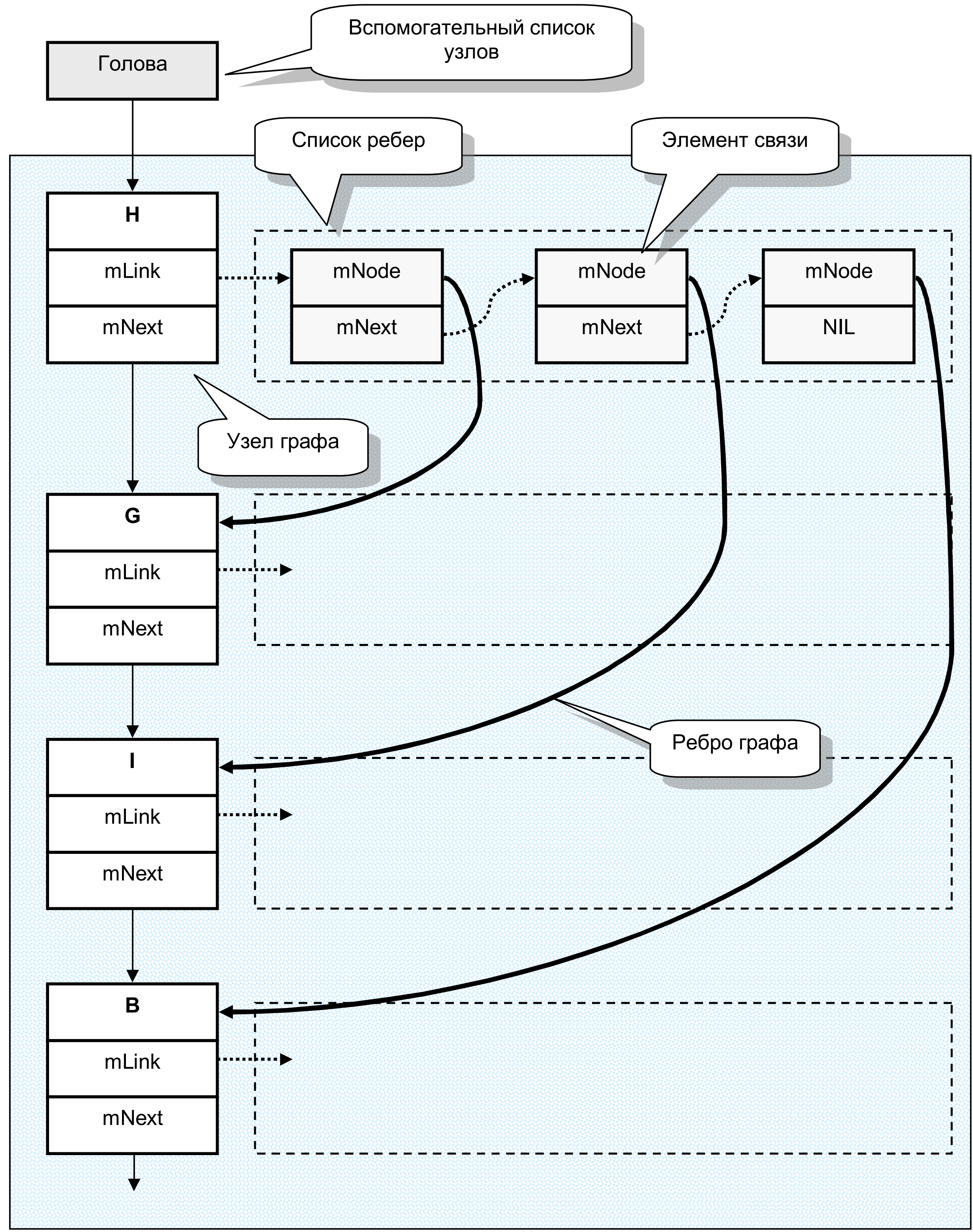


Рис. 134 – Организация связей графа на примере узла H

Слева видны тонкие стрелки, ведущие сверху вниз — это вспомогательный список, на который нанизаны узлы графа. Порядок следования узлов в этом списке не важен, важно лишь то, что двигаясь от головы списка по ссылкам **mNext**, можно достать любой узел. Этот список не определяет зримых связей между узлами.

Видимые нам ребра графа формируются списками, что вставлены внутрь каждого узла. Головы этих списков — это поля **mLink**. Чтобы не загромождать схему, я показал лишь список для узла H. Элементы списка связей вытянулись на схеме слева направо, они сцеплены полями **mNext**, — не путайте их с полями **mNext** в узлах графа. Полезной нагрузкой элементов списка связей будут указатели **mNode**, ссылающиеся на соседние узлы. Именно эти ссылки, показанные на схеме жирными стрелками, определяют видимую форму графа, то есть его ребра. На рис. 135 показана часть графа, соответствующая схеме рис. 134.

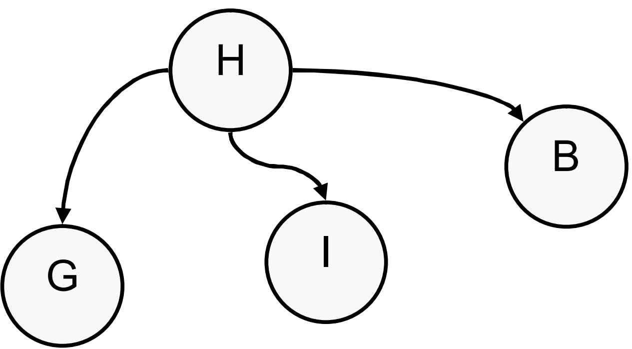


Рис. 135 – Часть графа, соответствующая схеме рис. 134

Здесь показаны лишь ребра, идущие от узла H, но подобные списки содержатся и в других узлах. Например, в списке связей узла G есть ссылка на узел H, поскольку узлы взаимно связаны. Так парами указателей создаётся двусторонняя связь узлов G ↔ H (рис. 136).

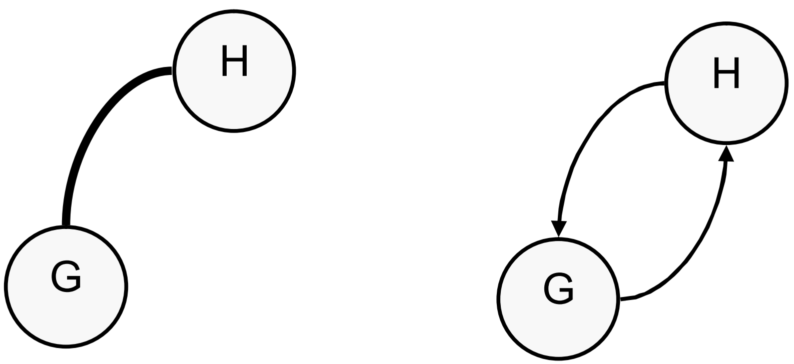


Рис. 136 – Ребро графа (слева) и внутреннее его представление (справа)

Прежде, чем выразить эту мудреную структуру на Паскале, повторю основные идеи.

- Узлы графа представлены записями.
- Каждая запись узла содержит: 1) полезные поля, 2) голову списка ребер и 3) указатель на следующий узел во вспомогательном списке.
- Полезной нагрузкой в списке ребер являются указатели на смежные узлы графа.

Всё кажется, запутано, словно паутина (а паутина — это тоже граф!). Однако выраженное на Паскале это описание выглядит не таким уж страшным.

```
type  PNode = ^TNode;      { Указатель на запись-узел }
      PLink = ^TLink;      { Указатель на список связей }

      TLink = record       { Элемент списка связей }
        mLink : PNode;     { указатель на смежный узел }
        mNext : PLink;     { указатель на следующую запись в списке }
      end;

      TNode = record       { Узел графа (страна) }
        mName : Char;      { Название страны (одна буква) }
        mLinks: PLink;     { список связей с соседями (ребра) }
        mNext : PNode;     { указатель на следующую запись в списке }
      end;

var List : PNode;   { список всех стран континента (узлов графа) }
```

Здесь определены два типа записей: элемент для списка узлов (**TNode**) и элемент для списка связей (**TLink**). Соответственно объявлены и два типа указателей на них. Для доступа к графу нужна всего одна глобальная переменная **List** — указатель на первый элемент во вспомогательном списке. И это всё! Как видите, пока ничего сложного.

Ввод и вывод графа

Мы обрисовали граф в памяти, а это уже полдела. Или ещё полдела. Следующая забота — организовать ввод и вывод графа. Так мы поступали и раньше, изучая множества, массивы и другие сложные типы данных.

Напомню ещё раз кусочек входного файла, с которым мы будем иметь дело.

```
A B D F I
B A C I H
C B D
```

Здесь первый символ строки — это имя страны, а последующие — её соседи. Например, в третьей строчке показано, что страна C соседствует со странами B и D.

Сначала обсудим алгоритм ввода графа в общих чертах.

Разумеется, что файл будем обрабатывать построчно. Взяв первый символ строки, проверим, нет ли во вспомогательном списке узла с таким именем? Возможно, что узел для этой страны уже создан при обработке предыдущих строк (что будет ясно из следующего абзаца). Если узел ещё не создан, создаем его и вставляем во вспомогательный список (обозначим этот узел буквой **P**).

Далее просматриваем оставшиеся символы строки. Для каждого из них тоже проверяем наличие готового узла. Если его нет, создаем этот узел (назовем его **q)**, вставляем во вспомогательный список, и устанавливаем связь между узлами **P** и **q**. Эта связь будет односторонней: от **P** к **q**. Но, поскольку связь для каждой пары узлов устанавливается дважды, то, в конце концов, мы получим двусторонние связи. Например, при обработке второй строки файла будет установлена связь B → C, а при обработке третьей — связь B ← C.

Теперь всё сказанное изобразим блок-схемой (рис. 137).

Чтобы облегчить себе дальнейший труд, заготовим две функции и процедуру, а именно:

- функцию для поиска узла по его имени;
- функцию для создания нового узла;
- процедуру для установки связи между двумя узлами.

Функцию поиска узла по его имени объявим так.

```
function GetPtr(aName : char): PNode;
```

Она ищет во вспомогательном списке узел по заданному в параметре имени. В случае успеха, функция вернет указатель на узел, а иначе — **NIL**.

Функция **MakeNode** создает новый узел графа с заданным именем, вставляет его во вспомогательный список узлов и возвращает указатель на этот узел.

```
function MakeNode(aName : Char): PNode;
```

И, наконец, процедура установки связей **Link** добавляет в список связей первого узла элемент связи со вторым узлом.

```
procedure Link(p1, p2 : PNode);
```

Все три подпрограммы очень просты, поскольку работают со списками.

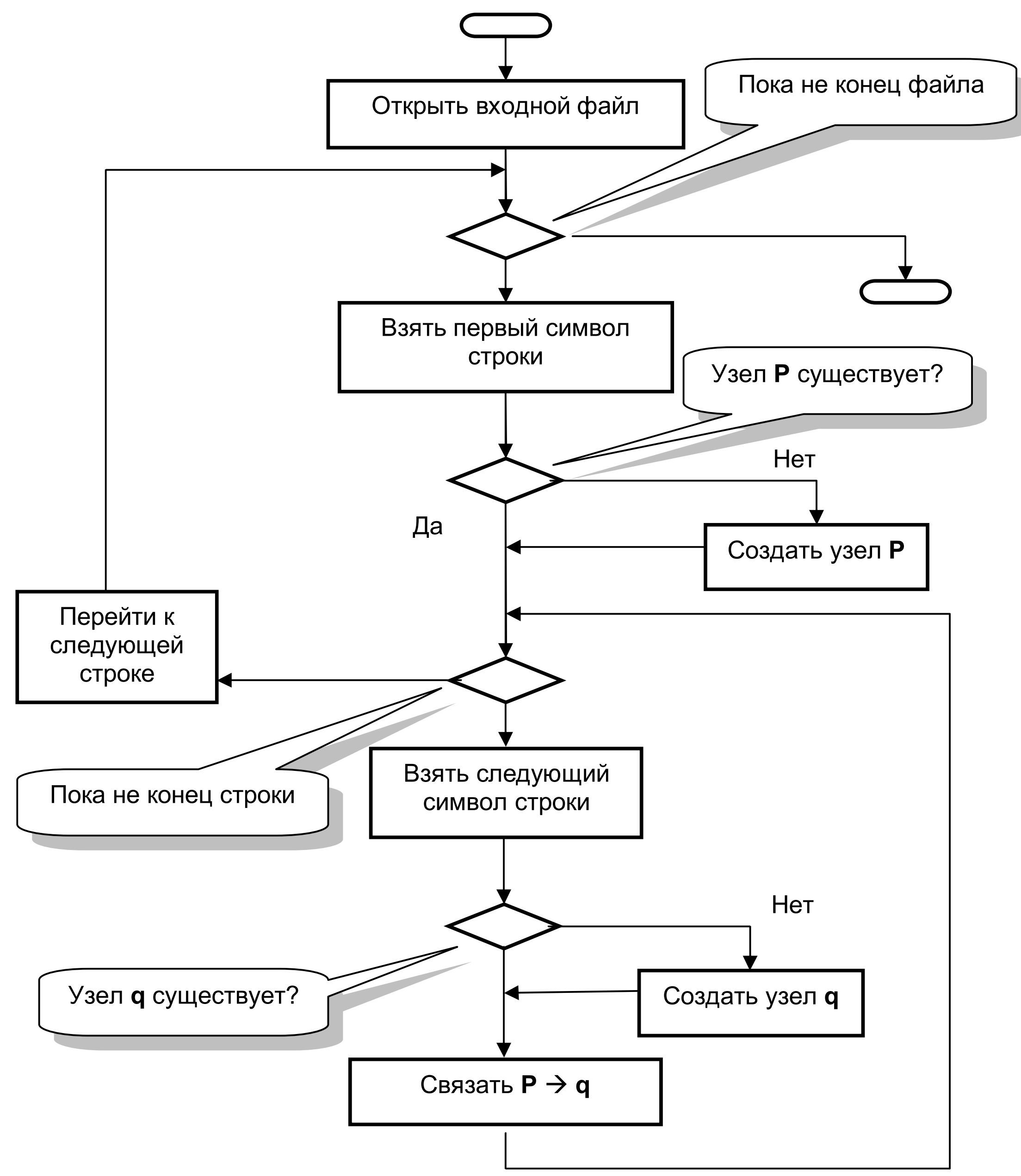


Рис. 137 – Алгоритм чтения графа

Немногим сложнее будет процедура распечатки графа, она объявлена так.

```
procedure ExpoData(var F: Text);
```

Процедура пробегает по вспомогательному списку узлов и спискам связей, распечатывая имена стран и их соседей.

Остальные детали алгоритма пояснены в программе **P_57_1**.

```
{ P_57_1 – Ввод и вывод графа }
type  PNode = ^TNode;      { Указатель на запись-узел }
      PLink = ^TLink;      { Указатель на список связей }
      TLink = record       { Тип список связей }
        mLink : PNode;     { указатель на смежный узел }
        mNext : PLink;     { указатель на следующую запись в списке }
      end;
      TNode = record       { Тип запись для хранения страны (узла графа) }
        mName : Char;      { Название страны (одна буква) }
        mLinks: PLink;     { список связей с соседями (смежными узлами) }
        mNext : PNode;     { указатель на следующую запись в списке }
      end;
var List : PNode;   { список всех стран континента (узлов графа) }

      { Функция поиска страны (узла графа) по имени страны }
function GetPtr(aName : char): PNode;
var p : PNode;
begin
  p:= List; { поиск начинается с головы списка }
  { проходим по элементам списка }
  while Assigned(p) do begin
    if p^.mName= aName
      then break              { нашли! }
      else p:= p^.mNext;      { а иначе следующий }
  end;
  GetPtr:= p;
end;

  { Функция создает новую страну (узел), вставляет в глобальный список List
    и возвращает указатель на новый узел }
function MakeNode(aName : Char): PNode;
var  p : PNode;
begin
  New(p);              { создаем переменную }
  p^.mName:= aName;    { копируем имя }
  p^.mLinks:=nil;      { список связей пока пуст }
  p^.mNext:= List;     { указатель на следующий берем из заголовка }
  List:= p;            { заголовок указывает на новый узел }
  MakeNode:= p;        { результат выполнения функции }
end;
```

```
      { Процедура установки связи узла p1 с узлом p2 }
procedure Link(p1, p2 : PNode);
var  p : PLink;
begin
  New(p);                  { создаем переменную-связь }
  p^.mLink:= p2;           { поле mLink должно указывать на p2 }
  p^.mNext:= p1^.mLinks;   { указатель на следующий берем из заголовка }
  p1^.mLinks:= p;          { заголовок указывает на новый узел }
end;

      { Процедура чтения графа из текстового файла }
procedure ReadData(var F: Text);
var C : Char;
    p, q : PNode;
begin
  Reset(F);
  while not Eof(F) do begin
    if not Eoln(F) then begin     { если строка не пуста }
      Read(F, C);                 { читаем имя страны }
      C:=UpCase(C);               { перевод в верхний регистр }
      p:= GetPtr(C);              { а может эта страна уже существует? }
      if not Assigned(p)
        then p:= MakeNode(C);     { если нет, - создаем }
      while not Eoln(F) do begin  { чтение стран-соседей до конца строки }
        Read(F, C);
        C:= UpCase(C);
        if C in ['A'..'Z'] then begin   { если это имя страны, а не пробел }
          q:= GetPtr(C);                { проверяем существование страны }
          if not Assigned(q)            { если не существует, - создаем }
            then q:= MakeNode(C);
          Link(p, q);                   { связываем страну p с q }
        end
      end
    end;
    Readln(F);   { переход на следующую строку файла }
  end;
end;
```

```
      { Процедура распечатки графа }
procedure ExpoData(var F: Text);
var p : PNode;
    q : PLink;
begin
  Rewrite(F);
  p:= List;   { начало просмотра списка стран (узлов) }
  while Assigned(p) do begin
    Write (F, p^.mName);          { название страны }
    q:= p^.mLinks;                { начало просмотра списка соседей }
    while Assigned(q) do begin
      Write(F, ' ', q^.mLink^.mName); { название соседа }
      q:= q^.mNext;                   { следующий сосед }
    end;
    Writeln(F);      { конец строки }
    p:= p^.mNext;    { следующая страна }
  end;
  Close(F);
end;
var    F_In, F_Out : Text;  { входной и выходной файла }
begin {--- Главная программа ---}
  List:= nil;
  Assign(F_In, 'P_57_1.in');
  ReadData(F_In);                 { читаем граф из входного файла }
  Assign(F_Out,'P_57_1.out');
  ExpoData(F_Out);                { печатаем в выходной файл }
end.
```

Запустив эту программу, я обнаружил на выходе такой результат:

```
G I H F
E F D
H I G B
C D B
I H G B A
F G E A
D E C A
B H I C A
A I F D B
```

Это явно отличается от входных данных, разница налицо, неужели ошибка? Да, порядок следования узлов не совпадает. И порядок перечисления связей в

строках тоже. Но нарисованный по этим данным граф оказался копией исходного! Всё потому, что порядок перечисления узлов и ребер графа не важен, главное — связи между узлами.

Ознакомившись с графами, мы готовы теперь последовать за придворным программистом Ником. Так айда в следующую главу!

Итоги

- Граф – это структура, состоящая из узлов и соединяющих их ребер.
- В памяти компьютера граф можно представить списком узлов и списками связей.
- Двунаправленные ребра графа представляются парой указателей.
- Порядок перечисления узлов и связей графа не имеет значения, поскольку не влияет на форму графа.

А слабо?

А) Когда-то страны континента (рис. 130) не поддерживали дипломатических связей. Изобразите отвечающий этой эпохе граф, отражая ребрами дипломатические отношения. Кстати, такой граф без ребер называют лесом.

Б) В пору расцвета континента все страны установили между собой дипломатические отношения. Нарисуйте подобающий граф.

В) В период политического кризиса соседние страны перессорились между собой и разорвали дипломатические отношения. Какие ребра графа уцелели? Нарисуйте его.

Г) Пусть названия стран представляются не буквами, а словами. Возьмите карту Европы и создайте входной файл для нескольких соседних стран, например:

```
Франция Испания Италия Бельгия Швейцария
Италия Франция Швейцария Словения
```

и так далее, перечисляя страны-соседи и отделяя их одним или несколькими пробелами. Напишите программу для ввода и вывода такого графа. Что придется изменить в структуре узла?

Глава 58
По графу шагом марш!

Ознакомившись с графами, вернемся к программисту Нику, который всё ещё царапает прибрежный песочек. «Если бы, — бормочет Ник, — мне надо было попасть из страны E в страну H, то я бы поехал так». И он прочертил жирные стрелки, ведущие к цели через узлы F и G (рис. 138). «Но это я сообразил, глядя на карту, а без карты можно блуждать вот так», — и нацарапал стрелки, показанные пунктиром.

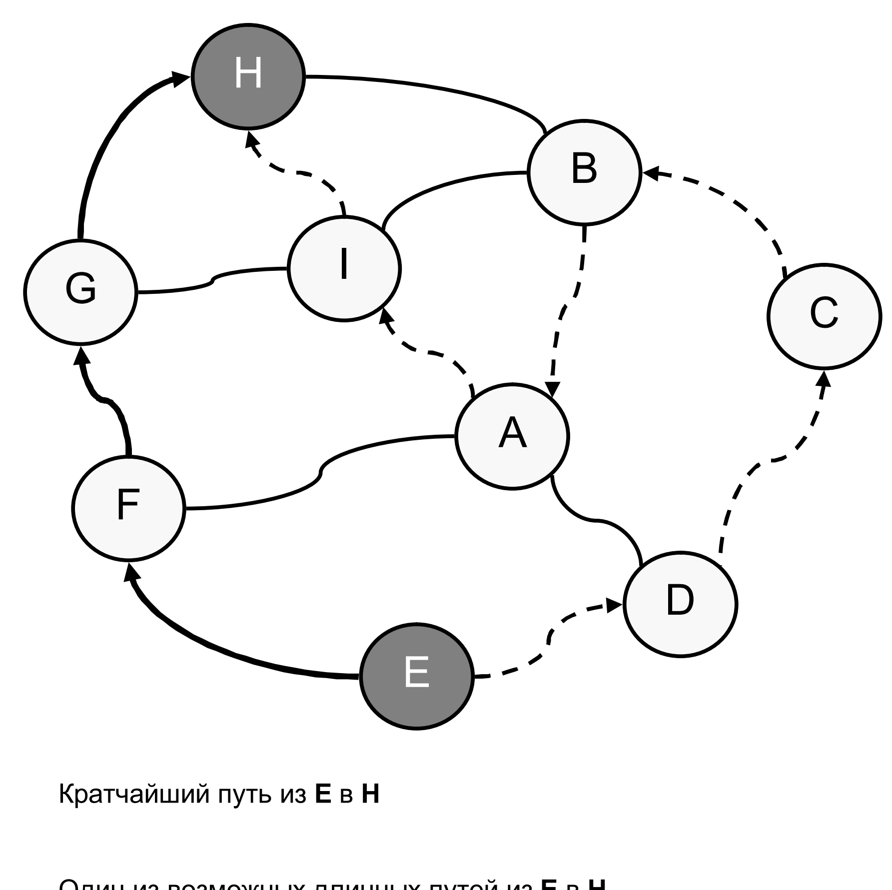


Рис. 138 – Возможные пути из E в H

Как растолковать компьютеру верный путь? Нужна свежая идея! Новое — это всего лишь забытое старое — почему-то вспомнилось ему. «А не построить ли тебе здесь империю, как ты сделал это в 49-й главе?» — шепнул Нику внутренний голос. И мысли программиста двинулись в этом направлении.

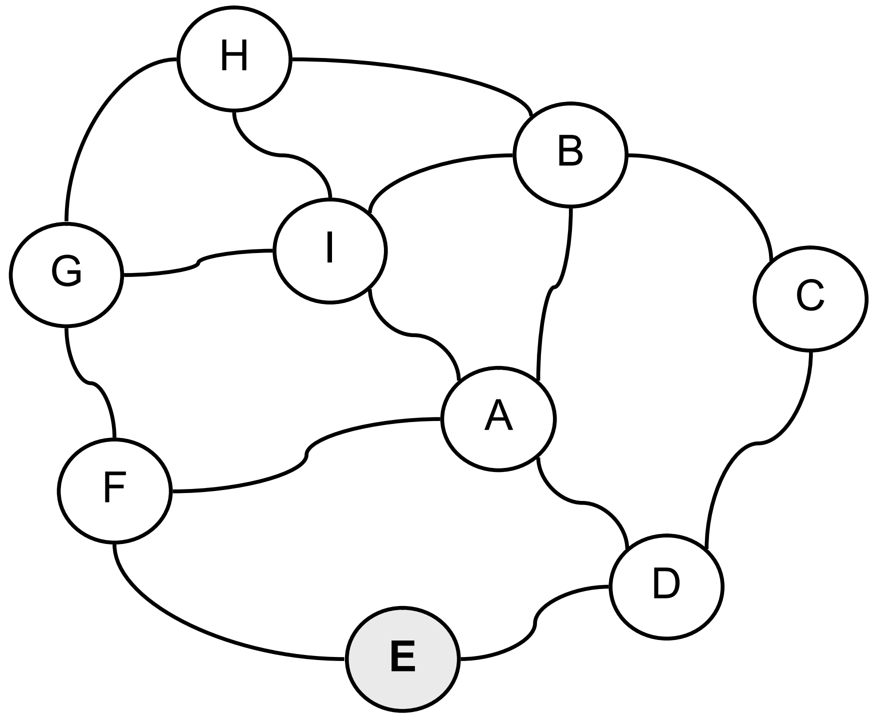
Империя номер два

Друзья, что вы слышали о постройке нынешних империй? Ведь на дворе не лютое средневековье! К чему проливать кровь, если от желающих нет отбоя, и очередь на присоединение к империям не пустует? Очередь упомянута мною не

зря, — она играет важную роль в будущем алгоритме. Кстати, алгоритм этот придумали не программисты, а политики. Судите сами, сейчас вместе с Ником мы последуем их примеру.

На рис. 139 показан граф в начале строительства «империи» (дальше я пишу это слово без кавычек). Условимся об окраске его узлов. Все страны континента (узлы) отнесем к трем категориям: 1) независимые страны, 2) страны, желающие присоединиться к империи и 3) страны, вошедшие в её состав. Независимые страны окрасим белым цветом, желающие присоединиться — серым, а присоединенные к империи — черным.

Откуда начать строительство? Пусть центром империи будет страна E. Окрасим её серым цветом и поставим в очередь на присоединение. Можно сказать, что страна E — первый кандидат на включение в несуществующую пока империю.



В очереди на присоединение страна E

Рис. 139 – Начало строительства империи из страны E

Серому кандидату поставим жесткое условие: хочешь быть принятым в империю и почернеть? Тогда уговори своих белых соседей тоже стать в очередь на присоединение и перекраситься в серый цвет. Так, страну E примут в империю, когда кандидатами на присоединение станут царства D и F, что и показано на рис. 140. Кандидат, выполнивший это условие, удаляется из очереди на присоединение и включается в империю — чернеет.

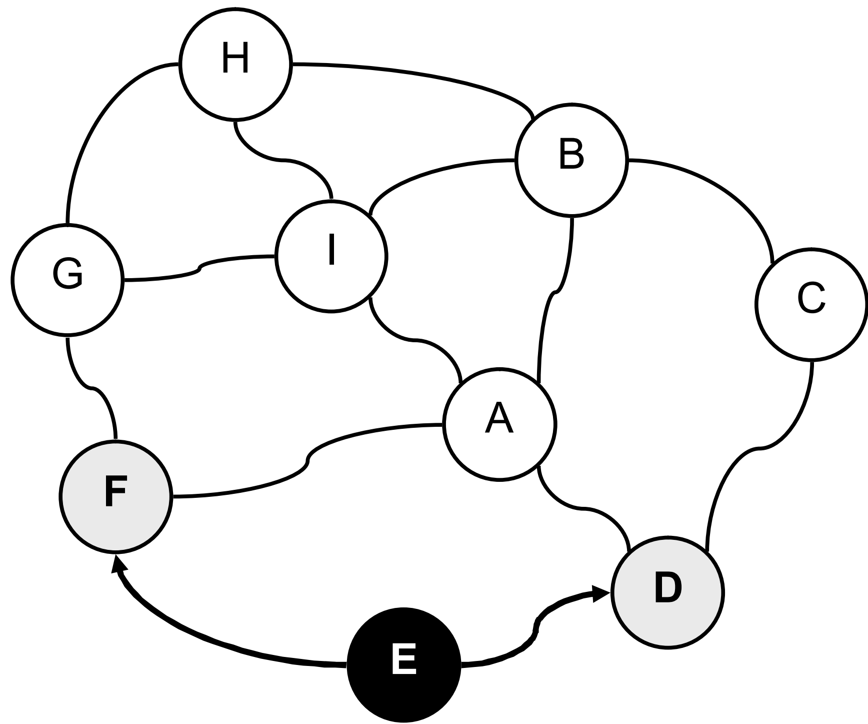
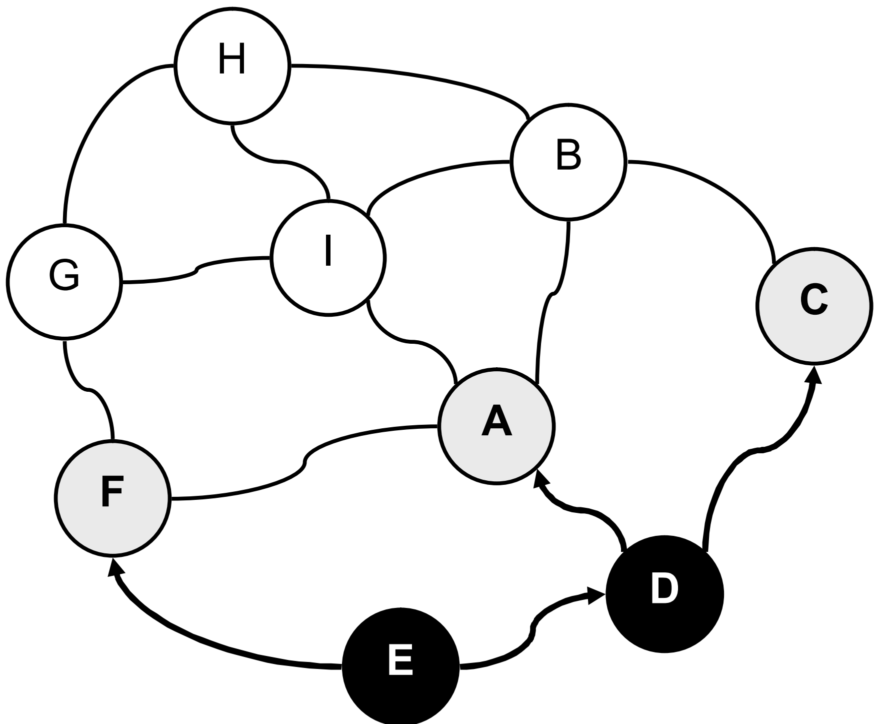


Рис. 140 – Состояние империи после присоединения первой страны

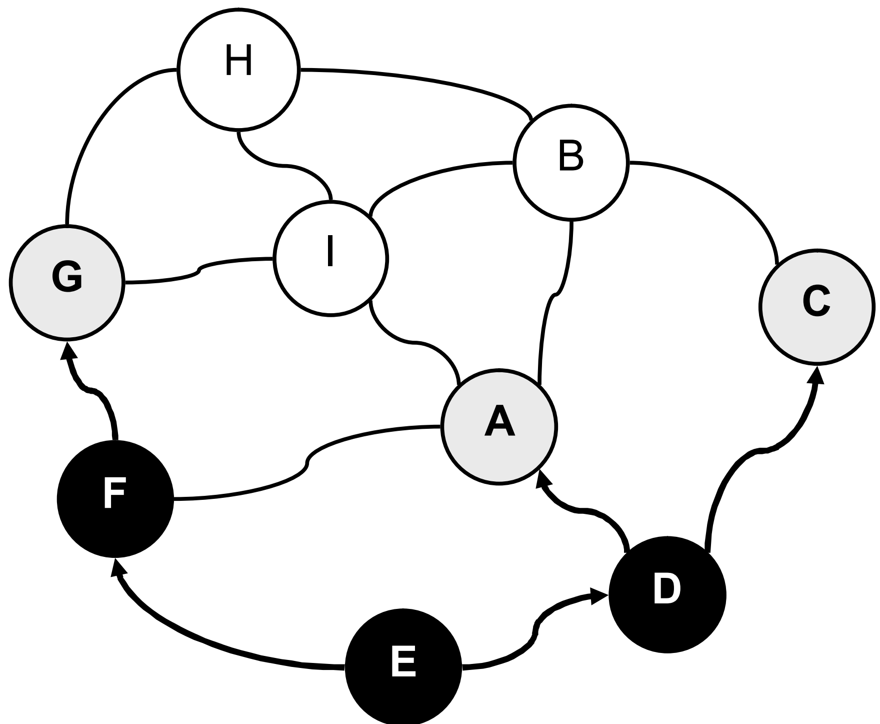
К слову сказать, строя империю, Ник постоянно думал о купцах. Жирными стрелками на графе он помечал их воображаемое движение, как если бы купцы шли вослед завоевателям.

Итак, страна E вошла в империю, а два её соседа — D и F — стали в очередь на присоединение (в каком именно порядке — D, затем F или наоборот — неважно). От них требуют то же самое — уговорить своих белых соседей. Так, для присоединения страны D ей надо убедить стать в очередь страны A и C. По мере выполнения этого условия страны-кандидаты чернеют и удаляются из очереди. После двух следующих присоединений (стран D и F) граф и очередь изменятся так, как показано на рис. 141 и рис. 142. Здесь же стрелками показано и воображаемое продвижение купцов.



В очереди на присоединение: **F, A, C**

Рис. 141 – Состояние империи после присоединения страны D



В очереди на присоединение: **A, C, G**

Рис. 142 – Состояние империи после присоединения страны F

Итак, строительство двинулось, но когда оно закончится? Очевидно, что страны с окраин империи рано или поздно войдут в число желающих, то есть, станут серыми, и тогда не останется белых соседей. А раз так, то очередь на присоединение постепенно опустеет, все страны почернеют, и строительство империи прекратится.

«Хорошо, — скажете, — только, причем тут поиск кратчайшего пути?». Но мы ведь не зря пустили купцов вослед завоевателям! Если купец потянет за собой ниточку, исходящую из начального узла E, то из любого узла империи сможет вернуться к началу, следуя по нити в обратном направлении (Рис. 143).

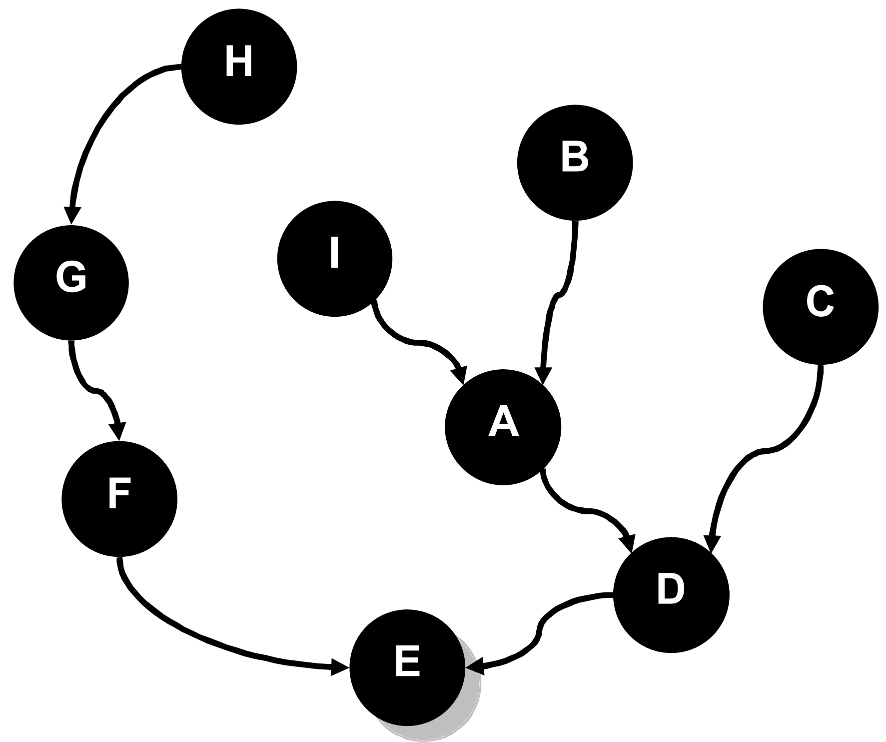


Рис. 143 – Порядок возврата в исходный узел E по цепочке обратных связей

Ник догадался, что путь из любого узла графа вдоль этих ниточек к исходной точке будет кратчайшим. Это следует из того, что империя расширялась присоединением ближайших соседей. Действительно, узлы D и F — ближайшие к исходному узлу E, ведь они его соседи. Точно так же узел G — ближайший к узлу F, а узел H — ближайший к узлу G. Эти рассуждения справедливы для любых нитей обратных связей.

Цепочки обратных связей тоже образуют граф, называемый деревом. Программисты часто применяют деревья, основное свойство которых состоит в наличии единственного пути между любыми узлами. Узел, из которого начато строительство дерева, является его корнем — это центр построенной нами империи (не географический, а политический центр).

Итак, строительство империи породило дерево обратных связей. Но как организовать эти нити? Введем в структуру узла ещё одно поле — указатель на

узел, из которого мы пришли сюда по ходу расширения империи. Назовем это поле **mPrev** — предыдущий. Например, для узлов F и D предыдущим будет узел E.

Остроумный Ник додумался по ходу строительства империи убить ещё одного зайца: определить длину пути от любого узла до центра. Так одновременно решается задача о минимальном количестве пересекаемых границ, которую в 49-й главе он решал через массив множеств. В самом деле, к чему плодить две программы, если можно обойтись одной? Ведь в ходе постройки дерева обратных связей определить расстояния несложно. Достаточно при переходе к очередному узлу отмечать в нём расстояние к центру империи, — оно будет на единицу больше того, что хранится в предыдущем узле. В центре империи это расстояние равно нулю, а в остальных узлах будет таким, как показано на рис. 144.

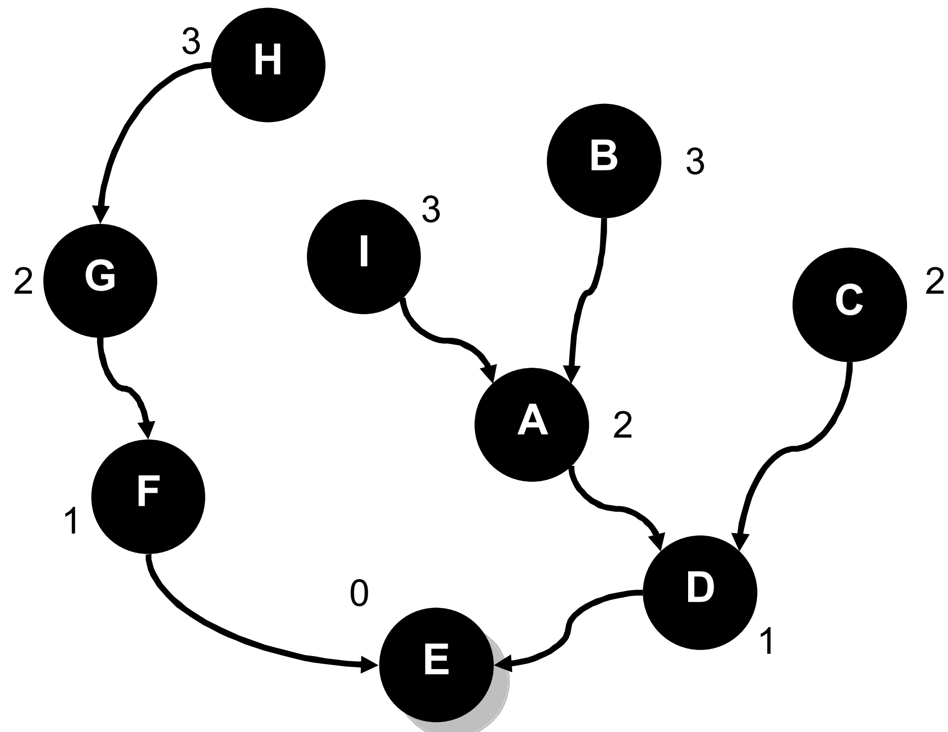


Рис. 144 – Расстояния и пути от узлов графа к центру империи

Подведем итог размышлениям Ника. Для поиска кратчайшего пути между двумя узлами графа, а заодно и определения расстояния между ними, сначала построим империю, центром которой будет один из этих двух узлов. Алгоритм этот называют обходом графа в ширину, он служит основой для решения многих задач на графах. Обход графа — не пустая прогулка. Двигаясь по нему, мы разместим в узлах информацию, необходимую для второго этапа решения — формирования кратчайшего пути.

Структура узла

Теперь уточним полезную нагрузку узла, что добавится в него? Во-первых, это упомянутый выше указатель на предыдущий узел **mPrev** — ниточка обратной связи. Во-вторых, надо застолбить поле для расстояния к центру империи, назовем

его **mDist** — «дистанция». Не забыть бы поле для окраски узла одним из трех цветов: белым, серым или черным. Назовем это поле **mColor** — «цвет», и будем хранить в нём одно из перечислимых значений цвета: **White**, **Gray**, **Black** (о перечислениях сказано в главе 32). В итоге проясняется следующая структура для узла графа:

```
type  TColor = (White, Gray, Black); { Перечисление: белый, серый, черный }
      TNode = record       { Запись для страны (узел графа) }
        mName : Char;      { Название страны (одна буква) }
        mColor: TColor;    { цвет узла, изначально белый }
        mDist : integer;   { длина пути к узлу, изначально -1 }
        mPrev : PNode;     { узел, из которого пришли в текущий }
        mLinks: PLink;     { список смежных узлов (ребер) }
        mNext : PNode;     { связь во вспомогательном списке }
      end;
```

В рассыпную!

Приступаем к постройке империи. Эта версия программы пока не найдет кратчайших путей между узлами, но подготовит почву для этого. Мы пройдем по всем узлам графа в ширину, начиная с исходного узла — центра империи. И по ходу движения разместим в этих узлах нужную информацию — обратные ссылки и расстояния к центру империи.

Программу P_58_1 построим на основе программы P_57_1, — из неё возьмем процедуру ввода графа и добавим ещё несколько подпрограмм. Две из них нужны для очереди, элементами которой будут узлы графа.

```
procedure PutInQue(arg: PNode);
function GetFromQue(var arg: Pnode): boolean;
```

Впрочем, для вас эти подпрограммы тоже не новы, — вспомните запись в танцевальный кружок в программе P_56_2. Там похожие процедуры применялись для очереди строк, а здесь организуется очередь узлов.

В начальный момент все вершины графа надо окрасить белым, — об этом позаботится простенькая процедура **InitList**. По-настоящему новой будет лишь процедура постройки империи **Expand**, вот её объявление.

```
procedure Expand(arg : PNode);
```

Она расширяет империю, начиная с заданного параметром **arg** узла. Алгоритм процедуры отвечает рассуждениям Ника, рассмотрим её подробней.

Перед входом в цикл заполняем поля стартового узла: в поле расстояния **mDist** заносим ноль, красим узел в серый цвет и ставим в очередь на присоединение. Теперь очередь содержит один элемент — исходный узел, то есть, центр империи.

Далее следует цикл **WHILE**, он выполняется, пока очередь желающих войти в империю не опустеет. Выбрав из очереди функцией **GetFromQue** первый узел (в этот момент очередь опустеет, но ненадолго), пробегаем по списку его белых соседей, располагая там нужную информацию, перекрашивая соседей в серый цвет и помещая их в очередь. После этого извлеченный из очереди узел **P** очерняем и возвращаемся к началу цикла **WHILE**. Поскольку очередь узлов уже не пуста (добавились соседние узлы), функция **GetFromQue** выберет из неё следующий узел, и цикл **WHILE** выполнится вновь. В конце концов, белые узлы когда-то иссякнут. Тогда пополнение очереди прекратится, серые узлы постепенно будут выбраны из неё, очередь опустеет, и цикл **WHILE** завершится.

Вот, собственно и всё. Для наблюдения за экспансией империи в процедуру вставлены операторы печати, не влияющие на её работу (они подчеркнуты).

```
{ P_58_1 - Обход графа в ширину }

type  PNode = ^TNode;      { Указатель на запись-узел }
      PLink = ^TLink;      { Указатель на список связей }
      TColor = (White, Gray, Black);  { Перечисление для цветов узла }

      TLink = record       { Список связей }
        mLink : PNode;     { указатель на смежный узел }
        mNext : PLink;     { указатель на следующую запись в списке }
      end;

      TNode = record       { Запись для хранения страны (узел графа) }
        mName : Char;      { Название страны (одна буква) }
        mColor: TColor;    { цвет узла, изначально белый }
        mDist : integer;   { длина пути к узлу, изначально -1 }
        mPrev : PNode;     { узел, из которого пришли в данный }
        mLinks: PLink;     { список смежных узлов (указатели на соседей ) }
        mNext : PNode;     { указатель на следующую запись в списке }
      end;
```

```
var List : PNode;   { список всех стран континента }
    Que  : PLink;   { очередь присоединяемых узлов }

      { Функция поиска страны (узла графа) по имени страны }
function GetPtr(aName : char): PNode;
{ Взять из P_57_1 }
end;
      { Функция создает новую страну (узел) }
function MakeNode(aName : Char): PNode;
{ Взять из P_57_1 }
end;

      { Процедура установки связи узла p1 с узлом p2 }
procedure Link(p1, p2 : PNode);
{ Взять из P_57_1 }
end;
      { Процедура чтения графа из текстового файла.}
procedure ReadData(var F: Text);
{ Взять из P_57_1 }
end;

      { Помещение указателя на узел  в глобальную очередь Que }
procedure PutInQue(arg: PNode);
var p: PLink;
begin
  New(p);             { создаем новую переменную-связь }
  p^.mLink:= arg;     { размещаем указатель на узел }
  { размещаем указатель в голове очереди }
  p^.mNext:= Que;     { указатель на предыдущую запись }
  Que:=p;             { текущая запись в голове очереди }
end;
```

```
      { Извлечение из очереди указателя на узел }
function GetFromQue(var arg: Pnode): boolean;
var p, q: PLink;
begin
  GetFromQue:= Assigned(Que);
  if Assigned(Que) then begin
    { Поиск последнего элемента (хвоста) очереди }
    p:= Que;  q:=p;
    { если в очереди только один элемент, цикл не выполнится ни разу! }
    while Assigned(p^.mNext) do begin
      q:=p;           { текущий }
      p:=p^.mNext;    { следующий }
    end;
    { p и q указывают на последний и предпоследний элементы }
    arg:= p^.mLink;
    if p=q                      { если в очереди был один элемент... }
      then Que:= nil            { очередь стала пустой }
      else q^.mNext:= nil;      { а иначе отцепляем последний элемент }
    Dispose(p);                 { освобождаем память последнего элемента }
  end;
end;
{ Процедура расширения (экспансии) империи, начиная с заданного узла arg }
procedure Expand(arg : PNode);
var p : PNode;
    q : PLink;
begin
  arg^.mDist:= 0;            { расстояние до центра империи = 0 }
  arg^.mColor:= Gray;        { метим серым цветом }
  PutInQue(arg);             { и помещаем в очередь обработки }
  while GetFromQue(p) do begin    { извлекаем очередной узел }
    Write(p^.mName, ' ->');       { печатаем название узла – для отладки }
    q:= p^.mLinks;                { начинаем просмотр соседей }
    while Assigned(q) do begin
      if q^.mLink^.mColor = White then begin  { если сосед ещё белый }
        q^.mLink^.mColor:= Gray;          { метим его серым }
        q^.mLink^.mDist:= p^.mDist +1;    { расстояние до центра }
        q^.mLink^.mPrev:= p;              { метим, откуда пришли }
        PutInQue(q^.mLink);               { и помещаем в очередь обработки }
        Write(q^.mLink^.mName:2);         { имя соседа – это для отладки }
      end;
```

```
      q:= q^.mNext;    { переход к следующему соседу }
    end;
    p^.mColor:= Black; { после обработки узла метим его черным }
    Writeln;           { новая строка – это для отладки }
  end;
end;
      { Инициализация списка узлов перед постройкой империи }
procedure InitList;
var p : PNode;
begin
  p:= List; { начинаем с головы списка узлов }
  { проходим по всем элементам списка }
  while Assigned(p) do begin
    p^.mColor:= White;  { цвет узла изначально белый }
    p^.mDist := -1;     { длина пути к узлу изначально -1 }
    p^.mPrev := nil;    { узел, из которого пришли в данный }
    p:= p^.mNext;       { следующий узел }
  end;
end;

var    F_In {, F_Out} : Text;  { входной и выходной файла }
       C : Char;           { название страны }
       Start : PNode;      { узел, с которого начинается расширение империи }

begin   {--- Главная программа ---}
  { Инициализация списка узлов и очереди узлов }
  List:= nil;  Que:= nil;
  Assign(F_In, 'P_57_1.in');
  ReadData(F_In);              { чтение графа }
  { Цикл ввода названий стран }
  repeat
    Write('Центр империи = '); Readln(C);
    C:= UpCase(C);
    if not (C in ['A'..'Z']) then break;
    Start:= GetPtr(C);              { указатель на центр империи }
    if Assigned(Start) then begin { если такая страна существует, }
      InitList;           { устанавливаем начальные значения в полях узлов }
      Expand(Start);      { расширяем империю от центра Start }
    end;
  until false
end.
```

В главной программе организован цикл, принимающий от пользователя исходную страну, из которой строится империя. Программа завершается при вводе любого символа, отличного от латинской буквы. Запустив программу, я ввел символ «E» и увидел на экране вот что:

```
E -> F D
F -> G A
D -> C
G -> I H
A -> B
C ->
I ->
H ->
B ->
```

Эти строки напечатаны операторами трассировки в процедуре **`Expand`**. Согласно первой строке из узла E мы попадаем в узлы F и D. Согласно второй — из узла F движемся в узлы G и A, и так далее. Последние четыре строки показывают, что узлы C, I, H и B оказались на окраинах империи, и продвижений оттуда нет. По этой трассировке нетрудно нарисовать дерево воображаемого продвижения купцов (рис. 145).

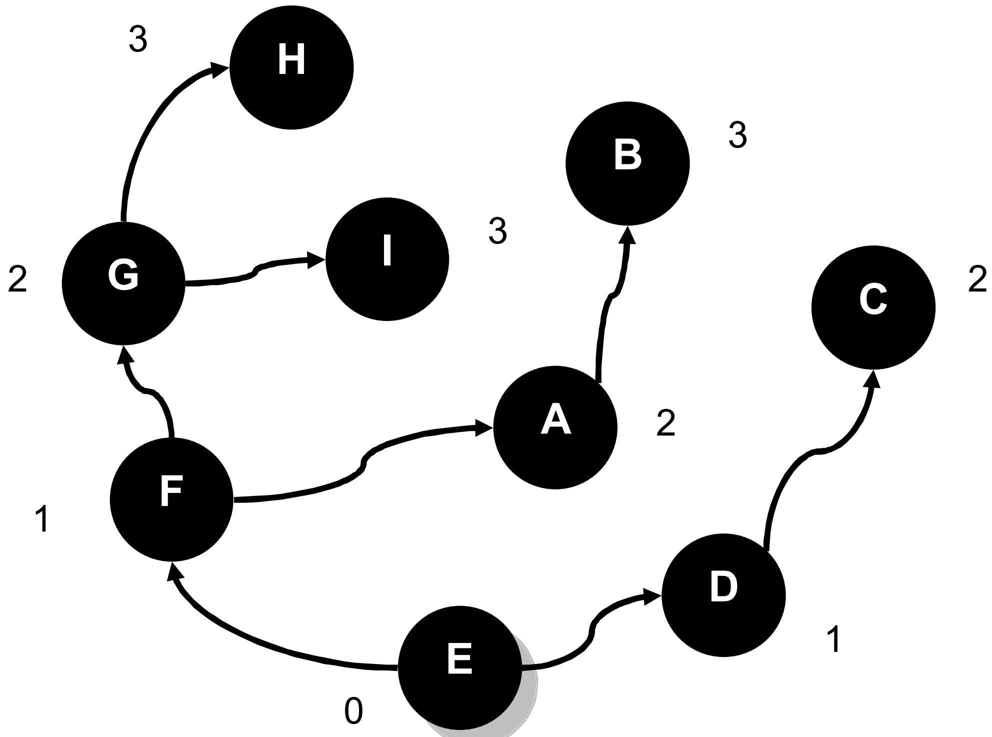


Рис. 145 – Воображаемое продвижение купцов

Сопоставьте это дерево с тем, что нацарапал на песке придворный программист (рис. 144). Разницы не заметит только слепой. В чем дело? Неужели вкралась ошибка?

Но, прежде чем огорчаться, сравните расстояния между центром империи и другими узлами — на обоих рисунках они совпадают. А это значит, что можно найти разные варианты кратчайших путей. Какой из них выберет программа — дело случая. Точнее, это определяется порядком ввода узлов. Мы знаем, что порядок строк входного файла не влияет на форму графа, но он влияет на выбор одного из кратчайших путей между узлами. Правда, купцам до этого дела нет, — ведь расстояния по таким путям будут одинаковыми.

Аты-баты

Теперь всё готово для создания полной версии программы. Пройдясь по графу вширь, мы разместили в узлах необходимые данные: расстояния от корня и обратные ссылки на пройденные узлы. Пора ставить победную точку — напечатать кратчайший путь между двумя узлами и длину этого пути.

Для постройки кратчайшего пути надо указать узел, из которого мы хотим попасть в центр империи. Двигаясь из него по цепочке обратных ссылок в направлении центра, мы, в конце концов, попадем в него. Значение обратной ссылки в центре империи равно **NIL**, что будет признаком окончания пути. С этой работой справится несложная функция **MakePath** — «создать путь».

```
function MakePath(arg : PNode): string;
```

В функцию передается узел, от которого надо вернуться к корню дерева, то есть к центру империи. Результатом будет строка пути вида «A → B → C».

Главную программу слегка дополним. Теперь пользователь введет названия двух стран, между которыми ищется кратчайший путь: «откуда» и «куда». Страну «откуда» назначим центром империи, а из страны «куда» будем возвращаться по цепочке обратных ссылок, — она составит параметр функции **MakePath**. Поскольку вводятся названия стран, а не указатели на них, преобразование имен в указатели тоже сделаем в главной программе.

Итак, в главной программе выполняются:

- ввод графа из текстового файла;
- ввод имен двух стран и преобразование их в указатели;
- подготовка узлов графа – заполнение полей начальными значениями;
- обход графа в ширину из заданного корня;
- распечатка кратчайшего пути по цепочке обратных ссылок.

Все действия, кроме первого, зациклим, — тогда пользователь сможет задавать для этого графа разные сочетания стран. Признаком выхода из цикла будет ввод любого символа, отличного от латинской буквы. Надеюсь, что сказанного достаточно, чтобы разобраться в программе P_58_2. Эта программа включает части программ P_57_1 и P_58_1, которые я лишь обозначил.

```
{ P_58_2 -  Поиск кратчайшего пути и определение расстояний в графе }
type  { Описания типов взять из P_58_1 }

var List : PNode;    { список всех стран континента }
    Que  : PLink;    { очередь присоединяемых узлов }

      { Функция поиска страны (узла графа) по имени страны }
function GetPtr(aName : char): PNode;
{ Взять из P_57_1 }
end;
      { Функция создает новую страну (узел) }
function MakeNode(aName : Char): PNode;
{ Взять из P_57_1 }
end;
      { Процедура установки связи узла p1 с узлом p2 }
procedure Link(p1, p2 : PNode);
{ Взять из P_57_1 }
end;
      { Процедура чтения графа из текстового файла }
procedure ReadData(var F: Text);
{ Взять из P_57_1 }
end;

      { Помещение указателя на узел  в глобальную очередь Que }
procedure PutInQue(arg: PNode);
{ Взять из P_58_1 }
end;

      { Извлечение из очереди указателя на узел }
function GetFromQue(var arg: Pnode): boolean;
{ Взять из P_58_1 }
end;
```

```
      { Инициализация списка узлов перед постройкой империи }
procedure InitList;
{ Взять из P_58_1 }
end;
{ Процедура расширения (экспансии) империи, начиная с заданного узла arg }
procedure Expand(arg : PNode);
{ Взять из P_58_1,
 подчеркнутые там операторы для трассировочной распечатки удалить }
end;

      { Функция для формирования пути от центра империи к заданному узлу }
function MakePath(arg : PNode): string;
var p : PNode;
    S : string;
begin
  S:= arg^.mName;       { имя конечного узла }
  p:= arg^.mPrev;       { указатель на предыдущий узел }
  while Assigned(p) do begin        { пока не достигли корня }
    S:= p^.mName +' -> '+ S;        { добавляем к пути имя узла }
    p:= p^.mPrev;                   { переход к следующему узлу }
  end;
  MakePath:= S;
end;

var    F_In {, F_Out} : Text; { входной и выходной файла }
       C1, C2 : Char;         { названия стран "откуда" и "куда" }
       Start, Stop : PNode;   { узлы  "откуда" и "куда" }

begin   {--- Главная программа ---}
  { Инициализация списка узлов и очереди узлов }
  List:= nil;  Que:= nil;
  Assign(F_In, 'P_57_1.in');
  ReadData(F_In);              { чтение графа }
  { Цикл ввода названий стран }
  repeat
    Write('Откуда= '); Readln(C1);
    C1:= UpCase(C1);
    if not (C1 in ['A'..'Z']) then break;
    Write('Куда  = '); Readln(C2);
    C2:= UpCase(C2);
    if not (C2 in ['A'..'Z']) then break;
```

```
    Start:= GetPtr(C1);        { начальный узел }
    Stop:= GetPtr(C2);         { конечный узел }
    if Assigned(Start) and Assigned(Stop) then begin
     { если такие страны существуют, }
      InitList;           { устанавливаем начальные значения в полях узлов }
      Expand(Start);      { расширяем империю от узла Start }
      Writeln (Stop^.mDist:3, '':3, MakePath(Stop));
    end;
  until false
end.
```

И вот настал час испытаний, вводим данные и получаем это:

```
Откуда:     E
Куда:       H
3   E -> F -> G -> H
```

Ура! Заработало! Сколько труда за этой короткой строчкой! Оправданы ли наши усилия? Конечно! Истина дорого дается, но теперь мы не заблудимся даже в многотысячном графе!

Графы — это мощный инструмент для решения широкого круга инженерных задач. Их применяют при сортировке и поиске данных (здесь используют деревья), в расчетах электрических схем и потоков жидкостей, — всё не перечислить. Мы отщипнули лишь краешек этого каравая, вы можете узнать о графах больше по книгам [9] и [17] из списка рекомендуемой литературы.

Итоги

- Обход графа в ширину — это один из базовых алгоритмов обработки графов. На нём основано решение многих задач, в том числе – поиск кратчайшего пути между узлами.
- При решении задач на графах в его узлах размещают информацию, нужную для решения данной задачи. Иногда такую информацию размещают и в ребрах, для этого в структуру ребер вводят необходимые поля.

А слабо?

А) Изобразите граф, содержащий не менее 20 вершин, обозначьте вершины латинскими буквами и поищите в этом графе кратчайшие пути программой P_58_2.

Б) Пусть узлы графа — это города, а ребра — дороги между ними. Расстояния между городами разные и они известны. Как отразить в структуре графа эти расстояния? Предложите что-нибудь.

В) Пусть расстояния между городами указаны в поле **mDist** записи **TLink**.

```
TLink = record        { Тип список связей }
  mLink : PNode;      { указатель на смежный узел }
  mDist : integer;    { расстояние между городами }
  mNext : PLink;      { указатель на следующую запись в списке }
end;
```

Предложите формат входного файла, содержащего в числе прочего расстояния между городами.

Г) Пусть выбран следующий формат входного файла, содержащий расстояния между городами (приведена одна строка).

```
A C 20 E 40
```

Здесь первый символ, как и ранее, обозначает текущий узел. Затем перечисляются его соседи с указанием расстояний до них. Например, между узлами A и C 20 км, а между узлами A и E — 40 км. Напишите процедуру ввода графа из такого файла.

Д) Напишите программу для поиска кратчайшего пути с учетом расстояний между городами. Подсказка: измените процедуру обхода в ширину так, чтобы серый кандидат исследовал всех соседей (не только белых), проверяя в них поле расстояния **mDist**. Если путь к соседу через кандидата окажется короче того, что уже отмечен в соседе, то следует изменить как расстояние, так и обратную ссылку в соседе. Вдобавок если сосед не серый, он ставится в очередь.

Е) Предположим, что купцам интересны не расстояния между столицами, а размер пошлин, вносимых при пересечении границ. Эти пошлины зависят от пересекаемой границы (то есть от пары стран). Годится ли для этого случая рассмотренная выше модель с разными расстояниями между городами?

Ж) С некоторых пор купцы учредили свой ежегодный съезд — Континентальный Купеческий Конгресс, где обсуждали свои проблемы. Каждая страна отправляла на съезд по одному делегату, а расходы на пересечение границ (визы) оплачивались из общей кассы. Посчитайте эти расходы (1 пиастр за каждое пересечение), если известна страна проведения конгресса. Учтите, что купцы следовали на съезд кратчайшими маршрутами.

З) Напишите программу для определения страны, где можно провести съезд с наименьшими издержками (см. задачу Ж).

И) Решите задачи Ж и З для случая разной стоимости виз на границах.

Глава 59
Крупные проекты

Вы заметили, насколько разбухли наши программы? Если так пойдет и дальше, то скоро вам понадобятся инструменты, применяемые в крупных проектах.

О модулях и разделении труда

Пока я возился с новой программкой, мой холодильник опустел, а на кухне скопилась немытая посуда. Впрочем, это пустяки в сравнении с заботами наших предков — крестьян, что жили сотни лет назад. Смотрите: вот мужик в окружении десятка голодных ртов, а там ещё князь с дружиной. И всех накорми, одень, согрей. А делать всё самому: пахать и сеять, избу рубить, ткать, шить и лапти плести. Эта морока называлась натуральным хозяйством.

С годами жизнь помаленьку наладилась: явились купцы, оживили торговлю и ремесла. Многие крестьяне бросили плуг, и ушли кто в кузнецы, кто в башмачники, а тот стал ткачом, — так возникло разделение труда. Потом появились фабрики и первые станки. И вот настало наше время — время огромных фабрик и сложнейших станков.

Взять хотя бы производство автомобиля — не самого сложного изделия по нынешним временам. А сколько народу копошится вокруг! Автомобильные заводы грандиозны, хотя здесь лишь собирают машины из частей, что делают на других заводах. Части сложных изделий называют узлами или модулями. Модуль — это часть изделия, которую можно изготовить и проверить отдельно от целого. Сборка изделий из модулей дала и высокое качество товара, и бешеную производительность труда. Так новые технологии породили и роскошные машины, и свободное время, убиваемое нами в автомобильных пробках.

Говорят, что история повторяется, ведь с технологиями программирования случилось то же самое, но гораздо быстрее. Ещё полвека назад одну программу мастерил один человек, — то была эпоха первых компьютеров и «натурального хозяйства» в программировании. По мере развития компьютеров росли и аппетиты пользователей, усложнялись решаемые задачи. Сообразно задачам усложнялись программы, и производство их в одиночку стало немыслимо. Потребовались разделение труда и специализация программистов, — точь-в-точь как в промышленности. И теперь производством сложных программ заняты программистские «фабрики».

Так что же это такое, технологии программирования? Алгоритмы какие-то? Или сверхсильные компьютеры? Нет, технологии — это орудия труда и средства его разумной организации. В чем они состоят применительно к нашему ремеслу?

Модули

Вам срочно потребовалась новая программа? Так призовите больше программистов и пусть они трудятся разом! Легко сказать, но как это сделать? Могут ли три писателя сразу сочинять один роман? Первый строчит начало, второй — середину, а третий — окончание? Или три художника писать одну картину? Но в программировании такое возможно! — выручает модульная организация программ.

В Паскале, как и в других современных языках, большой программный проект можно разбить на ряд частей — модулей (UNIT). Модули могут быть написаны и отлажены разными программистами на разных компьютерах независимо друг от друга, а затем соединены в одну программу. В этом и состоит модульное программирование.

Разделять на модули полезно даже небольшие проекты. Ведь модули — это своего рода склады, хранящие процедуры, функции и другие полезные вещи, — эти запасы пригодятся в других проектах. Модули в Паскале называют ещё библиотеками. Если так, то их содержимое — процедуры и функции — можно уподобить книгам. В поставку IDE включен ряд библиотек, содержащих массу полезных процедур, функций и типов данных. Это богатство служит для связи с операционной системой, устройствами ввода-вывода, и помогает строить красивые оконные интерфейсы. Фирменные библиотеки сделают ваши программы по-настоящему профессиональными, вам надо лишь разобраться в технологии модульного программирования.

Дробление модуля – «смертельный» номер

Рассмотрим заурядный случай. Положим, вы работаете над проектом строк эдак на пятьсот, что составляет десяток страниц печатного текста. Немалая часть вашей программы — процедуры, функции — уже отлажена, проверена и может пригодиться в других работах. Файл программы «разбух», и потому работать с ним неудобно. К чему эти готовые куски маячат перед глазами и мешают рыться в проблемных частях? Зреет здравая мысль: а не вынести ли их куда-нибудь в другой файл? Но так, чтобы связь с ними, не терялась. А потом, при необходимости, воспользоваться этими кусками в других проектах. Эти разумные мысли ведут вас к модульному программированию.

Сейчас мы разобьем на модули одну из наших программ. Нужен «доброволец», пусть им будет программа P_56_1. Мы распилим её на две части, как дамочку в цирке. Не пугайтесь, — программа, как и дамочка, останется живёхонька и здоровёхонька. Напомню, что P_56_1 — это шуточная программа для перестановки строк файла в обратном порядке. Вот её схема, где тела процедур я заменил многоточиями, а часть комментариев удалил. Для экономии места дальше я буду показывать программу схематично.

```
{ P_56_1 - перестановка строк файла }
type  PRec = ^TRec;
      TRec = record
        mStr  : string;
        mNext : PRec;
      end;
var Stack : PRec;  { Голова стека }
procedure Push(const arg : string);
{ . . . }
end;
function Pop(var arg : string): boolean;
{ . . . }
end;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
var F : text;   S : string;
begin       {--- Главная программа ---}
{ . . . }
end.
```

Вероятно, средства для работы со стеком пригодятся нам где-то ещё, и есть смысл сохранить их на складе. Разделим программу на две части. Первую часть, до пунктирной линии — подпрограммы **Push**, **Pop**, объявление типа и переменную **Stack** — скопируем в другой файл и назовем его MyLibr.pas — моя библиотека. Сохраним её в папке с нашими программами. Здесь вам пригодится умение работать с несколькими окнами и держать открытыми оба файла.

Скопированный кусок в исходной программе теперь не нужен, — выбросим его, а то, что осталось, сохраним под именем P_59_1.

```
{ P_59_1 - Первичный файл проекта }
var F : text;   S : string;
begin       {--- Главная программа ---}
{ . . . }
end.
```

Файл с главной программой называют первичным (Primary), стало быть, P_59_1 — это первичный файл нашего проекта. Будет ли он компилироваться? Ясно, что нет. Ведь там вызываются удаленные из файла процедура и функция. Надо подсказать компилятору, что они переехали в файл MyLibr. Для такой переадресации служит список импорта, который возглавляет ключевое слово **USES**. Пользоваться списком импорта легко, — достаточно поместить его в начале программы, перечислив после слова **USES** имена нужных модулей (несколько имен

разделяются запятыми). В нашем случае внешним модулем является файл MyLibr, поэтому дополненный списком импорта первичный файл станет таким.

```
{ P_59_1 - Первичный файл проекта }
uses  MyLibr;     { импортируемый модуль }
var F : text;   S : string;
begin       {--- Главная программа ---}
{ . . . }
end.
```

Как видите, расширение файла в списке **USES** не указывают, а сам список завершает точка с запятой. Теперь всё, что компилятор не найдет в основной программе, он будет искать в файле MyLibr. Говорят, что первичный файл импортирует модуль MyLibr. Компилируется такой проект из двух файлов как обычно — нажатием клавиши *F9*. Впрочем, для успешной компиляции время ещё не пришло.

Теперь обратимся к файлу MyLibr. На первый взгляд он содержит всё нужное для компиляции, но это не так. Ведь это не программа, а лишь часть её — модуль. Файлу надо придать особую форму — форму библиотечного модуля, что достигается вставкой в него четырех обязательных ключевых слов (ниже они подчеркнуты).

```
unit MyLibr;        { имя библиотечного модуля }
interface           { секция интерфейса }
{. . .}
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
implementation      { секция реализации }
{. . .}
end.
```

Обратите внимание на последнее слово **END** с точкой. Так же, как в главной программе, оно обязательно и завершает библиотечный модуль.

Но открывает файл ключевое слово **UNIT** (модуль), — оно назначает модулю имя. Имя указывают за словом **UNIT** через пробел, эта строка завершается точкой с запятой. Имя модуля должно совпадать с именем файла, но без расширения PAS. Впрочем, имя может быть и длиннее — до 63-х символов, но первые восемь из них должны повторять имя файла. Итак, присвойте модулю имя **MyLibr**.

Следующие два ключевых слова: **INTERFACE** (интерфейс) и **IMPLEMENTATION** (реализация) делят модуль на две части. Эти части так и называются: секция интерфейса и секция реализации. Важно правильно расставить эти ключевые слова. Кстати, точка с запятой за ними не ставится!

Начнем с секции реализации, хотя по порядку она идет второй. Эта секция должна вмещать всё (или почти всё), что требуется для работы модуля: объявления типов, констант, переменных, а также процедуры и функции. И потому в неё сейчас свалим то, что перетащили из основной программы. Итак, слово **IMPLEMENTATION** мы поставим перед описанием типов, и в результате библиотечный файл станет таким.

```
unit MyLibr;        { имя библиотечного модуля }
interface           { секция интерфейса }
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
implementation      { секция реализации }
type  PRec = ^TRec;
      TRec = record
        mStr  : string;
        mNext : PRec;
      end;
var Stack : PRec;  { Голова стека }
procedure Push(const arg : string);
{ . . . }
end;
function Pop(var arg : string): boolean;
{ . . . }
end;
end.
```

Теперь обратимся к секции интерфейса. Хотя слово **INTERFACE** уже на месте, но секция пока пуста. Каково её назначение? Здесь пора рассказать о видимости объектов, размещенных в модуле. Точнее, о видимости их за пределами этого модуля. Оказывается, что всё, помещенное нами в секцию реализации, невидимо извне, скрыто от посторонних глаз. Так устроено по причинам, которые мы обсудим позже. Но для компиляции проекта надо приоткрыть часть модуля внешнему миру, иначе из главной программы (которая проживает в другом файле) не будут видны нужные ей идентификаторы библиотечного модуля. Для этого и нужна секция интерфейса.

На обозрение выставим только идентификаторы, нужные внешним модулям. Что именно? Как это узнать? Самый верный способ — запустить компиляцию первичного модуля, и тогда компилятор покажет первый незнакомый идентификатор, — его описание и разместим в секции интерфейса. Повторная компиляция выявит следующий неизвестный идентификатор. Так постепенно мы обнаружим, что первичный модуль нуждается в переменной **Stack**, процедуре **Push** и функции **Pop**. Впрочем, вам это было и так ясно.

Чтобы выставить напоказ константы, переменные и описания типов, надо просто напросто перенести их из секции реализации в секцию интерфейса. Но с процедурами и функциями так не выйдет. В секцию интерфейса нельзя вставлять исполняемый код — только объявления! Но можно разместить заголовки процедур и функций, для чего просто скопировать их ещё раз. Сделав это, мы получим следующий библиотечный модуль.

```
unit MyLibr;        { имя библиотечного модуля }
interface           { секция интерфейса }
type  PRec = ^TRec;
      TRec = record
        mStr  : string;
        mNext : PRec;
      end;
var Stack : PRec;  { Голова стека }
procedure Push(const arg : string);            { заголовок процедуры }
function Pop(var arg : string): boolean;       { заголовок функции }
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
implementation     { секция реализации }
procedure Push(const arg : string);
{ . . . }
end;
function Pop(var arg : string): boolean;
{ . . . }
end;
end.
```

Описание типа и переменная **Stack** объявлены лишь один раз — в секции интерфейса, отсюда они видны как внутри, так и вне библиотечного модуля. Повторное их объявление в секции реализации будет ошибкой. Заголовки процедур и функций в секциях интерфейса и реализации должны совпадать. Но в секции реализации разрешено не повторять списки параметров, и тогда компилятор возьмет их из секции интерфейса.

Всё, что размещено в секции интерфейса, называют списком экспорта библиотечного модуля. Таким образом, первичный модуль программы импортирует то, что экспортирует библиотечный модуль, — так налаживается связь между модулями. На рис. 146 показаны окна с файлами нашего проекта: вверху — первичный файл, внизу — файл библиотечного модуля.

```
Free Pascal IDE
File  Edit  Search  Run  Compile  Debug  Tools  Options  Window  Help
C:\User\Pascal\P_59_1.pas
{ P_59_1 - Первичный файл проекта }

uses MyLibr;

var F : text;   S : string;

begin           {--- Главная программа ---}

  { Открываем входной файл }
  Assign(F, 'P_59_1.pas'); Reset(F);
  { Пока не конец файла, читаем строки и помещаем в стек }
  while not Eof(F) do begin

C:\User\Pascal\MyLibr.pas
unit MyLibr;     { библиотечный модуль }

interface        { --- секция интерфейса --- }

procedure Push(const arg : string);

function Pop(var arg : string): boolean;

implementation   { --- секция реализации --- }

type  PRec = ^TRec;        { Тип указатель на запись }
      TRec = record        { Тип запись для хранения связанных строк }
  1:38
F1 Help  F2 Save  F3 Open  Alt+F9 Compile  F9 Make  Alt+F10 Local menu
```


Рис. 146 – Окна преобразованного проекта P_59_1

Компиляция проекта

Теперь всё готово для компиляции и запуска нашего проекта. Перейдите в окно первичного модуля и нажмите сочетание *Ctrl+F9*, — оба файла будут откомпилированы, и программа запустится как обычно. Я не зря прошу перейти в окно именно первичного модуля. Если при нажатии *Ctrl+F9* активным будет другое окно, компилятор выдаст обидное сообщение: «Cannot run a unit» — нельзя запустить модуль. В самом деле, модуль — это лишь часть программы, он не может быть исполнен. Компилятор же считает, что в активном окне содержится главная программа и пытается её запустить.

Чтобы не спотыкаться здесь, настройте в IDE имя первичного файла, то есть файла с главной программой, — здесь это P_59_1. Для этого обратитесь к пункту меню *Compile → Primary file…* и укажите там нужный файл (рис. 147). Теперь компилятор будет знать, с какого файла начинать компиляцию. Но и библиотечный файл MyLibr он тоже будет обрабатывать всякий раз, когда вы измените в нём что-то, — это очень удобно.

А если начнёте другой проект? Тогда не забудьте сменить имя первичного файла, либо сбросьте это имя через пункт меню *Compile → Clear Primary file.*

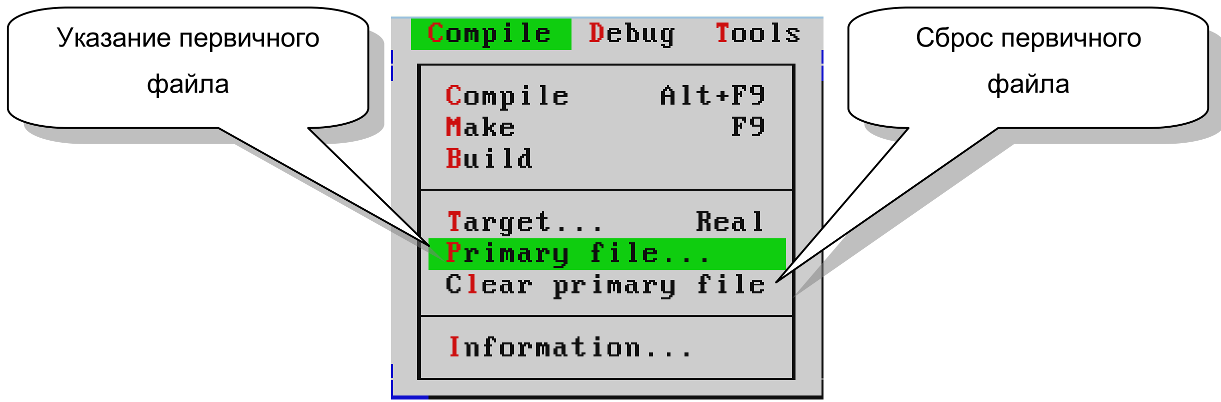


Рис. 147 – Пункты меню для настройки первичного файла

Инициализация модуля

Прежде чем завершить наш многофайловый проект, слегка улучшим его и покажем полные тексты первичного и библиотечного модулей.

Начнем с того, что переменная **Stack** упоминается в главной программе лишь однажды — при инициализации:

```
  Stack:= nil;  { Инициализация стека пустым значением }
```

Когда-нибудь — в будущих проектах — вы забудете об этой важной мелочи, и наживете несколько часов головной боли. Но, если эту инициализацию перенести в модуль **MyLibr**, то можно впредь не вспоминать о ней. Для этого создадим в модуле ещё одну секцию — секцию инициализации. Она располагается в модуле последней и открывается ключевым словом **BEGIN**, вставленным перед завершающим словом **END**. Между этими словами записывают операторы инициализации, — всё это похоже на главную программу в первичном модуле. В нашем случае секция будет такой:

```
unit MyLibr;
{ . . . }
begin       { секция инициализации модуля }
  Stack:= nil;  { Инициализация стека }
end.
```

Когда сработает эта инициализация? Вы знаете, что стрельба начинается с операторов главной программы в первичном модуле P_59_1. Это справедливо, пока не подключены библиотечные модули. С ними порядок исполнения программы слегка изменится. Первыми будут выполнены операторы в секциях инициализации подключенных модулей, причем в том порядке, в каком эти модули перечислены в списке **USES** (если там указано несколько модулей). И лишь затем начнёт выполняться главная программа в первичном модуле. Этот порядок гарантирует компилятор, ваше вмешательство здесь не требуется.

Итак, переместив инициализацию переменной из главной программы в модуль, мы обеспечим её автоматическое выполнение и в будущих проектах. Разумеется, что в главной программе инициализация уже излишня. А раз так, то и переменная **Stack** с описанием её типа в секции интерфейса уже ни к чему, — вернем всё это в секцию реализации. После всех перемещений наш проект обретет окончательный вид.

Внешний библиотечный модуль:

```
unit MyLibr;        { имя библиотечного модуля }

interface           { --- секция интерфейса --- }

procedure Push(const arg : string);      { заголовок процедуры }
function Pop(var arg : string): boolean; { заголовок функции }
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}

implementation      { --- секция реализации --- }

type  PRec = ^TRec;        { Тип указатель на запись }
      TRec = record        { Тип запись для хранения связанных строк }
        mStr  : string;    { хранимая строка }
        mNext : PRec;      { указатель на следующую запись }
      end;
var Stack : PRec;  { Голова стека }

      { Процедура размещения строки в стеке }
procedure Push(const arg : string);
var p : PRec;
begin
  New(p);              { создаем новую переменную-запись }
  p^.mStr:= arg;       { размещаем строку }
  { размещаем в голове стека }
  p^.mNext:= Stack;    { указатель на предыдущую запись }
  Stack:=p;            { текущая запись в голове стека }
end;
```

```
      { Процедура извлечения строки из стека }
function Pop(var arg : string): boolean;
var p : PRec;
begin
  Pop:= Assigned(Stack);  { Если стек не пуст, то TRUE }
  { Если стек не пуст... }
  if Assigned(Stack) then begin
    arg:= Stack^.mStr;      { извлекаем данные из головы стека }
    p:= Stack;              { временно копируем указатель на голову }
    Stack:= Stack^.mNext;   { переключаем голову на следующий элемент }
    Dispose(p);             { удаляем ненужный элемент }
  end
end;
begin       { --- секция инициализации модуля --- }
  Stack:= nil;  { Инициализация стека пустым значением }
end.
```

Теперь в интерфейсной части модуля маячат лишь процедура **Push** и функция **Pop**. Первичный файл проекта с главной программой станет таким:

```
{ P_59_1 - Первичный файл проекта }
uses MyLibr;
var F : text;   S : string;
begin       {--- Главная программа ---}
  { Открываем входной файл }
  Assign(F, 'P_56_1.pas'); Reset(F);
  { Пока не конец файла, читаем строки и помещаем в стек }
  while not Eof(F) do begin
      Readln(F, S);     Push(S);
  end;
  Close(F);
  { Открываем выходной файл }
  Assign(F, 'P_56_1.out'); Rewrite(F);
  { Пока стек не пуст, извлекаем и печатаем строки }
  while Pop(S) do Writeln(F, S);
  Close(F);
end.
```

Откомпилируйте проект, запустите и проверьте, жива ли распиленная «дамочка»?

Структура модуля

Обретя первый опыт модульного программирования, воспарим над частностями и окинем взглядом всю модульную технологию.

Прежде всего, уточним структуру модуля. Из рис. 148 следует, что она схожа со структурой программы. В состав модуля, по мере необходимости, включаются те же самые персонажи: константы, типы данных, переменные, процедуры и функции. Всё это может располагаться в одной из двух секций. То, что требует экспорта, выставляют напоказ в секции интерфейса, а остальное прячут в секции реализации. Что касается процедур и функций, то в секцию интерфейса выносят при необходимости лишь копии их заголовков, а сами подпрограммы поселяют в секции реализации. Константы, типы и переменные, объявленные в секции интерфейса, в секции реализации не повторяют.

О совпадении имен

Настало время ответить на отложенный вопрос: зачем прятать часть модуля, почему бы не выставить всё напоказ? На это есть две причины, и обе веские.

Первая причина такова. Большие программы собирают из десятков библиотечных модулей, их создают разные люди, каждый из которых выбирает названия для своих объектов на свой вкус, не советуясь с другими. Если все эти имена сделать видимыми за пределами модуля, то некоторые из них совпадут и учинят невероятную путаницу.

Компилятор ищет идентификаторы сначала в текущем файле, а если их там нет, то в модулях из списка импорта **`USES`**. Причем перебирает модули в том порядке, в котором они перечислены. Есть вероятность, что вместо переменной, объявленной в одном модуле, компилятор наткнется на процедуру или функцию с тем же именем в другом модуле. Это может быть воспринято как ошибка и программа не скомпилируется.

Но хуже, если программа скомпилируется! В этом состоит вторая причина сокрытия лишних имен. Если компилятор найдет искомое, но не в том модуле, в котором вы предполагали — то-то будет морока с отладкой!

Отсюда вывод: не выставляйте напоказ ничего лишнего, будьте скромнее, — так поступают профессионалы. И всё же полностью исключить нежелательное совпадение имен удается не всегда. Как быть? Выход есть — используйте префикс с именем модуля. Префикс — это приставка перед идентификатором, отделяемая от него точкой. Например, в главной программе нашего проекта вызовы процедур можно записать так.

```
    MyLibr.Push(S);
    while MyLibr.Pop(S) do . . .
```

Этим мы подсказали компилятору, что процедуры **Push** и **Pop** берутся из модуля **MyLibr** и боле ниоткуда.

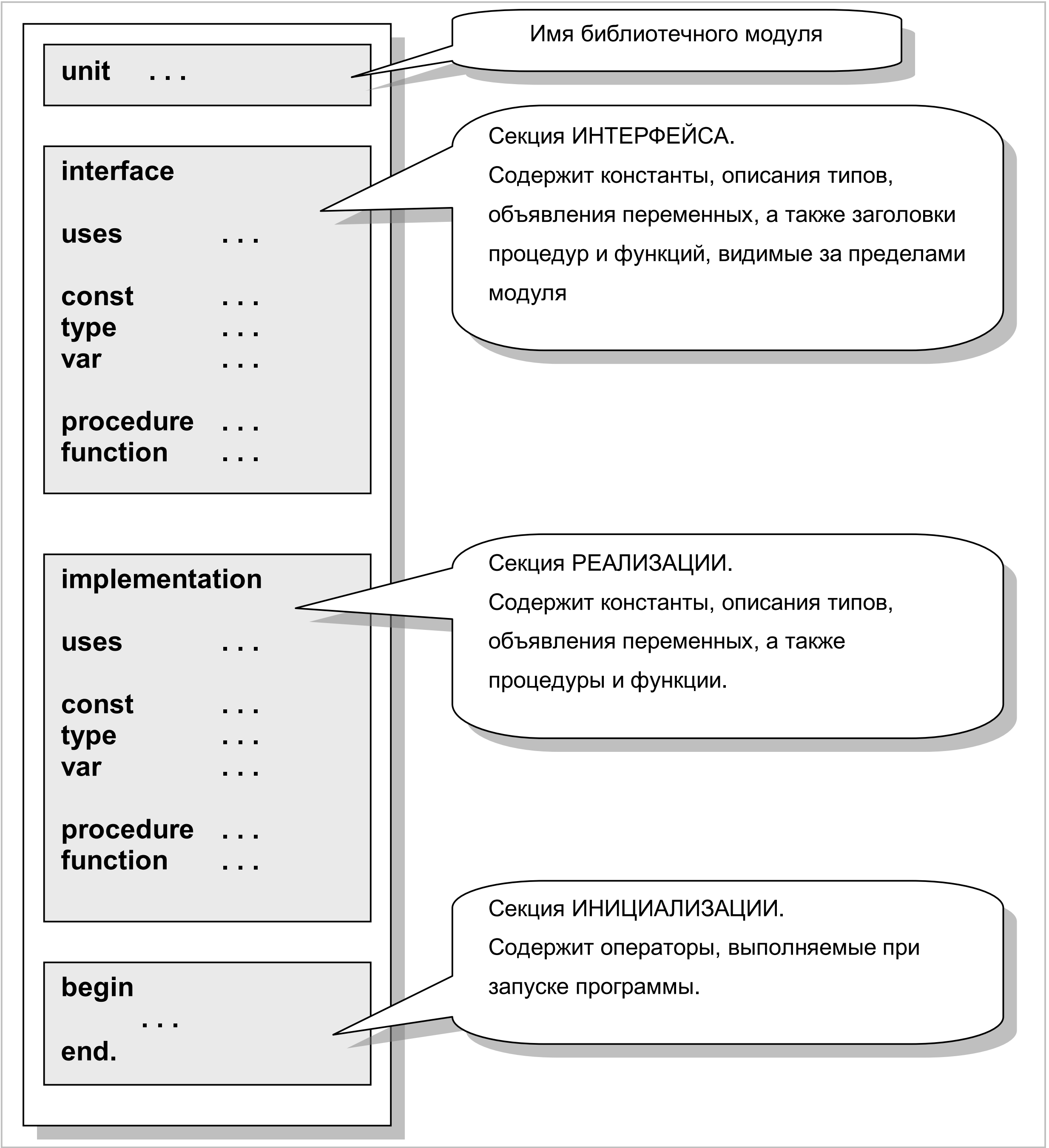


Рис. 148 – Общая структура библиотечного модуля

Модуль, указанный в префиксе, должен упоминаться в списке **USES**. Но есть одно исключение — это модуль по имени **SYSTEM**. В этом библиотечном модуле собраны процедуры и функции, встроенные в Паскаль. Модуль **SYSTEM** в списке **USES** никогда не упоминают.

Иногда программисты называют свои процедуры и функции так же, как встроенные в модуль **SYSTEM**, — это не запрещено. И тогда для уточнения имен пользуются префиксами с именами библиотек, например:

```
System.Writeln(F);      { стандартная процедура }
MyModule.Writeln(S);    { моя «самоделка» }
```

Завершая обзор структуры модуля, обратим внимание на необязательные списки импорта **USES** в секциях интерфейса и реализации (рис. 148). Через эти списки библиотечный модуль импортирует нужные ему элементы из других модулей. Составляя списки, следуйте простому правилу: если внешний модуль требуется только в секции реализации, упоминайте его в списке **USES** секции реализации, а иначе — в секции интерфейса. Упоминать модуль в обоих списках нельзя. А когда импортировать нечего, список импорта не вставляют.

Сборочный цех

Оглядев структуру модуля, войдем теперь в сборочный цех IDE и рассмотрим порядок соединения модулей в единую программу — исполняемый файл.

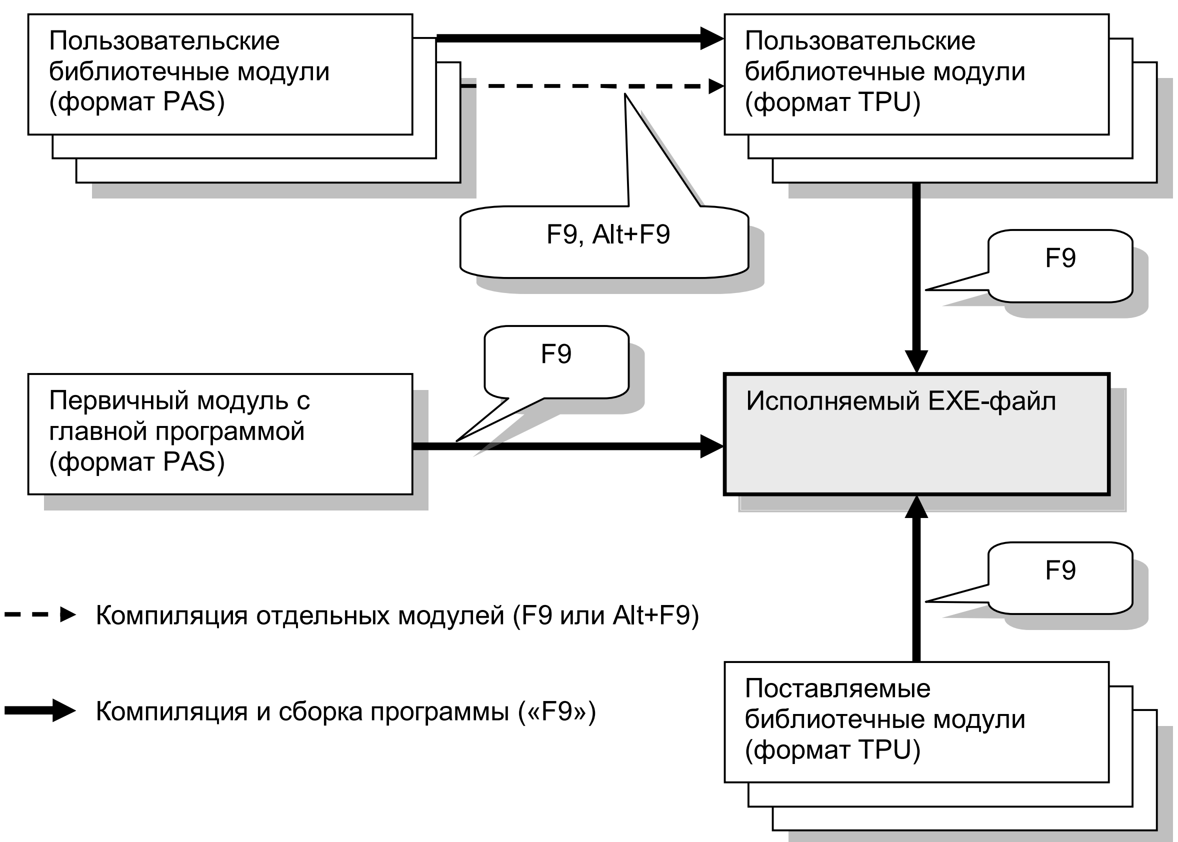


Рис. 149 – «Цех» компиляции и сборки проекта

Перед сборкой проекта все входящие в него библиотечные модули компилируются, в результате получаются файлы, расширения которых зависит от

используемого компилятора. Так, для Borland Pascal файлы получат расширение TPU (это сокращение от «Turbo Pascal Unit»). Для Free Pascal это будут пары файлов с расширениями O и PPU. Модули можно компилировать как по отдельности, так и вместе со всем проектом. Рассмотрим оба случая.

Для компиляции отдельного модуля откройте его в редакторе и нажмите сочетание *Alt+F9*, или выберите пункт меню *Compile → Compile*. Компилятор выполнит свою обычную работу по проверке ошибок и, при отсутствии таковых, сформирует библиотечный файл. На экране появится сообщение об успешной компиляции (рис. 150).

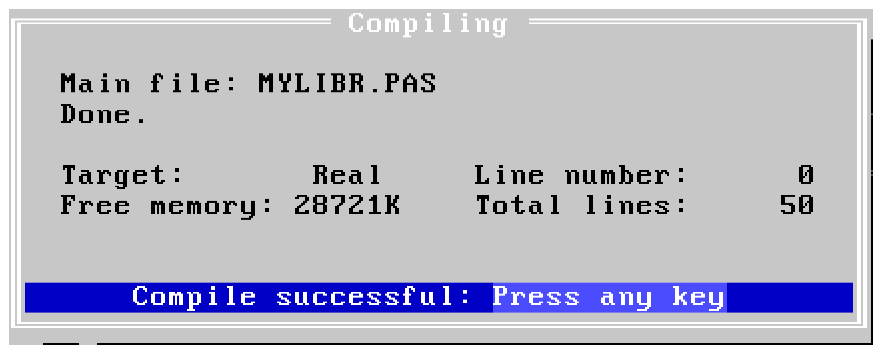


Рис. 150 – Сообщение об успешной компиляции модуля

Если же выявятся синтаксические ошибки, вам придется устранить их.

Модули компилируются и в ходе сборки проекта. При нажатии клавиши *F9* (или при выборе в меню *Compile → Make*) компилятор просматривает списки импорта **USES** как в главной программе, так и в модулях. Обнаружив очередной модуль, компилятор сравнивает время редакции его исходного файла (PAS) со временем создания откомпилированного файла. Если исходный файл оказался свежее откомпилированного или последний пока не существует, то модуль компилируется. Так или иначе, но перед сборкой проекта все его модули будут скомпилированы. Только после удачной компиляции первичного файла и всех связанных с ним модулей создаётся исполняемый EXE-файл.

Фирменные библиотеки

Я уже упоминал о библиотеках, входящих в состав IDE, — они поставляются и в исходном, и в откомпилированном виде. Вы можете применять эти библиотеки наряду со своими, предварительно ознакомившись с ними по документации или по встроенной справке. Ощутить полезность фирменных библиотек можно по этой небольшой программе:

```
uses CRT;   { Из CRT импортируются процедуры Sound, NoSound, Delay, ClrScr }
procedure Beep;   { короткий гудок }
begin
      Sound(300);        { включение динамика на частоте 300 Гц }
      Delay(500);        { задержка на полсекунды }
      NoSound;           { отключение динамика }
end;
begin {--- Главная программа ---}
      ClrScr;            { очистка экрана }
      Writeln('Привет, Мартышка!');
      Beep;              { короткий гудок }
      Readln;
end.
```

Здесь на предварительно очищенный экран выводится приветствие, сопровождаемое коротким гудком. В программе используется ряд процедур из библиотеки **CRT**, — там собраны средства для работы с экраном. Для успешной компиляции надо указать компилятору путь к файлу «CRT.TPU». При установке среды программирования фирменные библиотеки обычно попадают в папку «…\Units» (многоточием обозначена директория установки IDE). Уточнив положение библиотек, подскажите компилятору путь к месту их проживания через пункт меню *Options → Directories…* (рис. 151).

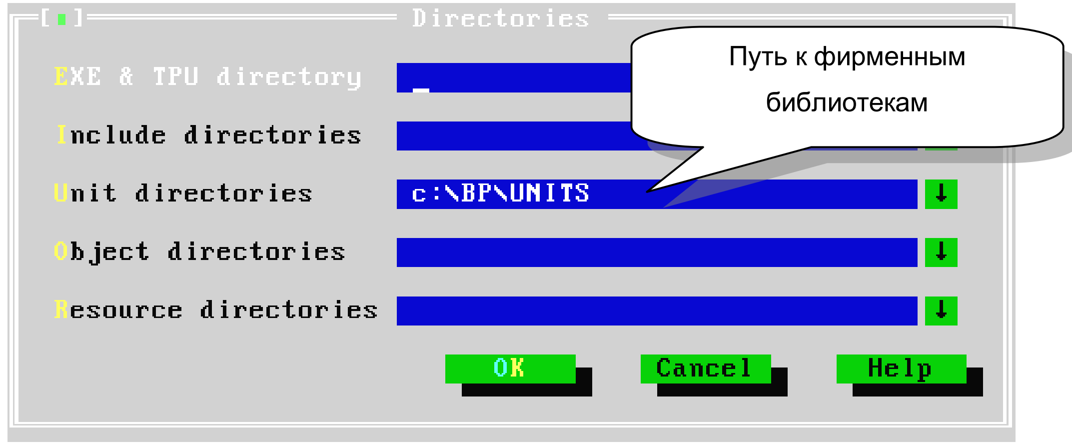


Рис. 151 – Указание пути к фирменным библиотекам

В данном примере предполагаем, что компилятор установлен в директорию «C:\BP», а библиотечные модули размещены в папке «C:\BP\UNITS».

Динамически загружаемые библиотеки (DLL)

Порой несколько разных программ используют общие для них процедуры и функции. Если при их компиляции подключить общую библиотеку, то процедуры из неё войдут в каждую из программ, увеличивая их общий «вес». Кому-то пришла в голову мысль отделить библиотеку от использующих её программ так, чтобы

библиотека загружалась в память лишь единожды в момент запуска первой из применяющих её программ. И тогда, при старте последующих программ, нужные им средства оказываются уже загруженными в память. Это уменьшает общий объём оперативной памяти, потребляемой всеми работающими программами. Динамически загружаемые библиотеки (DLL) могут разрабатываться не только на Паскале, но и на других языках (например, на Си или Ассемблере).

Итоги

- В программировании принято разделение труда посредством модульной технологии: современные программы собирают из модулей, разработанных разными программистами.
- Модуль содержит все необходимое для выполнения логически связанных действий: константы, типы, переменные, процедуры и функции.
- Каждый модуль обладает именем и содержит две обязательные секции: секцию интерфейса и секцию реализации.
- Имя модуля должно совпадать с именем файла без расширения.
- Секция интерфейса содержит объявления, видимые за пределами модуля.
- Секция реализации содержит невидимые за пределами модуля объявления, а также тела процедур и функций.
- Установку начальных значений глобальных переменных модуля выполняют в секции инициализации.

А слабо?

А) Разбейте на два модуля проект P_58_1 — обход графа в ширину. Что должно быть видимо за пределами модуля? Что поместить в секцию инициализации?

Задачи на темы предыдущих глав

Б) Императорские заботы. После постройки империи (см. главы 57 и 58) бывшие независимые государства стали провинциями и породили новые проблемы. Для доставки туда правительственных бумаг император нанял гонцов, которые для ускорения доставки следовали из столицы кратчайшими путями и лишь в одном направлении — от центра к окраинам империи. Сколько гонцов для этого нужно? — вот первый вопрос. Сколько времени потребуется для достижения самых дальних окраин, если переход из провинции в провинцию отнимает сутки? — это второй вопрос. В конечных пунктах (на окраинах) перед возвращением гонцам нужен отдых, на каких окраинах построить гостиницы? — это третий вопрос. Подсказка: возьмите за основу программу P_58_1 — обход графа в ширину — и внесите необходимые дополнения в процедуру **Expand**.

Глава 60
Мелкие хитрости

Нелегко совладать с крупным проектом, и тут не грех прибегнуть ко всяким уловкам и хитростям!

Включаемые файлы

Рассмотрим ещё одно средство дробления программного проекта — включаемые файлы, которые называют ещё INCLUDE-файлами. В сравнении с библиотечными модулями возможности этих файлов скромны, но каждая вещь хороша на своем месте.

Механизм включаемых файлов до безобразия прост: содержимое такого файла как бы вставляется в другой. Место вставки определяется директивой **$I**, за которой следует имя вставляемого файла. Вы скажете, что вставку можно сделать иначе — редактором текста. Но ценность директивы **$I** в том, что вставки как таковой не происходит, — оба файла не изменяются. Но в момент компиляции проекта включаемый файл как бы составит часть того файла, в который он «вставлен».

Вот пример. Создадим и сохраним в рабочей папке два файла, первый из которых назовем «HELLO.INC», и в нём будет лишь одна строка.

```
    Writeln('Привет!');
```

Второй файл — «HELLO.PAS» — будет таким.

```
begin {--- Главная программа HELLO.PAS ---}
    {$I Hello}
end.
```

Компиляция файла «HELLO.PAS» породит приветливую программу. Здесь в директиве **$I** указано имя вставляемого файла без расширения, поскольку расширение INC берется по умолчанию. Разумеется, что INCLUDE-файл не вставишь куда попало, — его содержимое должно сочетаться с тем окружением, в которое его погружают.

Незамысловатый механизм включаемых файлов даёт ощутимую пользу, вот пример. Предположим, вы работаете над крупным проектом, состоящим из нескольких модулей. Время от времени вам надо компилировать эти модули вместе с первичным файлом так, чтобы опции компилятора для всех файлов совпадали. Такого совпадения можно добиться следующим образом.

Сначала настройте нужные опции компилятора через пункт меню *Options → Compiler…*. Затем создайте новый файл и вставьте в него директивы

компиляции нажатием комбинации *Ctrl+O+O*; в результате в файле могут оказаться такие, например, строки.

```
{$A+,B-,D+,E-,F-,G+,I-,L+,N+,O-,P-,Q-,R-,S-,T+,V-,X+,Y+}
{$M 16384,0,655360}
```

Сохраните этот файл, пусть он называется «Options.inc» (не забудьте указать расширение). Затем в первой строке каждого модуля, где вы намерены применить эти опции, вставьте директиву **{$I Options}**.

```
{$I Options}
. . .
```

Теперь компиляция всех файлов с одинаковыми настройками гарантирована, поскольку опции, заданные в директивах, преобладают над «менюшными». Потребовалось изменить настройки? — тогда исправьте только файл «Options.inc» и повторно откомпилируйте проект.

Условная компиляция

Моряка украшает загрубелое лицо, землекопа выдают мозолистые руки, а программиста — шишки, набитые при отладке программ. И это не шутка! Отладка — пожалуй, самая сложная часть работы, и здесь уместны разные хитрости.

Одно из таких ухищрений — печать промежуточных результатов или так называемая трассировка, — вы знакомы с этим приемом. По завершении отладки, расставленные там и сям, операторы трассировки только мешают, — их приходится удалять. Умные программисты не убирают их навсегда, а комментируют, то есть заключают в фигурные скобки, — а вдруг ещё пригодятся? А хитрые поступают иначе. «Зачем мне копаться в файлах проекта, выискивая лишние операторы? — рассуждают они, — пусть компилятор сделает это сам». Здесь они уповают на условную компиляцию.

Условная компиляция — это механизм, встроенный в компиляторы многих современных языков. Через него можно указать части программы, которые, в определенных случаях, компилировать не следует. Пропущенные таким образом куски программы равносильны комментариям. Условная компиляция организуется двумя директивами, с которыми мы сейчас ознакомимся.

Первая из них — **$DEFINE** — определяет некоторое имя. Это имя никак не связано с именами переменных, процедур и прочих объектов программы и может даже совпадать с ними — это не опасно. Определенное директивой имя используется лишь для условной компиляции так, как это будет показано далее. Вот парочка примеров определения таких имен.

```
{ $define Test }
{ $define Print }
```

Вторая директива — это собственно директива условной компиляции. Она похожа на условный оператор Паскаля, — не путайте их! Условный оператор срабатывает при исполнении программы, а директива — при компиляции проекта. Повторяю: директива условной компиляции — это всего лишь подсказка компилятору со стороны программиста!

Подобно условному оператору, такая директива выражается тремя словами, образующими две ветви компиляции.

```
{ $ifdef ABC - начало директивы }
      { эта часть скомпилируется, если имя ABC определено }
{ $else }
      { эта часть скомпилируется, если имя ABC НЕ определено }
{ $endif }
```

Посредством **$IFDEF** компилятор проверяет, определено ли где-то ранее директивой **$DEFINE** некоторое имя. Если да, то следующие за директивой операторы, вплоть до **$ELSE** будут откомпилированы, а иначе — пропущены. Операторы, следующие за **$ELSE**, ждет обратная участь. Возможен и сокращенный вариант директивы, содержащий лишь одну ветвь компиляции.

```
{ $ifdef ABC - начало директивы }
      { эта часть скомпилируется, если имя ABC определено }
{ $endif }
```

Таким образом, пара директив **$DEFINE** и **$IFDEF-$ELSE-$ENDIF** совместно образуют механизм условной компиляции.

Рассмотрим ещё пример. Предположим, вы пишите программу, выводящую результат в дисковый файл, однако при отладке вам хочется наблюдать результат на экране. Ну что ж, перенаправить вывод на экран совсем несложно, достаточно указать для файла пустое имя. Но при этом надо ещё организовать остановку программы после вывода на экран, чтобы успеть рассмотреть что-то. По окончании отладки надо восстановить имя дискового файла и удалить лишний оператор. Все эти хитрости легко устроить директивами условной компиляции.

```
{$define Debug - Определяем признак отладочной компиляции }
const
      {$ifdef Debug}
            FName = '';             { Предстоит вывод на экран }
      {$else}
            FName = 'Hello.out';    { Предстоит вывод в файл на диске }
      {$endif}
var F : text;
begin
      Assign (F, FName);  Rewrite(F);
      Writeln(F, 'Привет, мартышка!');
      {$ifdef Debug}    Readln;     {$endif}
      Close(F);
end.
```

Здесь определено некое условное имя **Debug** (отладка), а программа содержит два варианта вывода: на экран и в файл. Разумеется, что после компиляции такой программы вывод пойдет на экран. Но переделка программы в боевую версию будет элементарна: достаточно удалить знак доллара в первой строке, и тогда директива **$DEFINE** превратится в безобидный комментарий.

```
{ define Debug - это всего лишь комментарий, а не директива! }
```

Теперь, когда условное имя **Debug** не определено, при повторной компиляции константа **FName** примет значение «**Hello.out**», оператор **Readln** не откомпилируется, и мы получим вывод в дисковый файл.

А что, если надо компилировать многофайловый проект в разных вариантах: отладочном и боевом? Здесь разумно сосредоточить определения условных имен в одном месте, разместив их во включаемом файле (как мы сделали это с опциями компилятора). Возьмем, например, созданный ранее файл опций «Options.inc». Добавив в него определения условных имен, мы сможем воздействовать на компиляцию сразу всех файлов проекта.

```
{$A+,B-,D+,E-,F-,G+,I-,L+,N+,O-,P-,Q-,R-,S-,T+,V-,X+,Y+}
{$M 16384,0,655360}
{$define Test - действует на условия типа $IFDEF Test }
{$define Print - действует на условия типа $IFDEF Print }
```

Разумеется, что в модулях проекта должны быть расставлены надлежащие директивы условной компиляции.

Итоги

- Включаемые (INCLUDE) файлы – это части программы, которые в момент компиляции автоматически вставляются в указанные директивами **$I** места. Содержимое включаемого файла должно сочетаться с окружением, в которое оно вставляется.
- Условная компиляция – это удобное средство создания различных версий одной программы. Собрав условные имена и опции во включаемом файле, можно централизовано управлять компиляцией крупного проекта.

А слабо?

А) Создайте отладочный вариант программы обработки классного журнала (глава 31), в котором вывод результатов будет выполняться на экран. Примените условную компиляцию.

Задачи на темы предыдущих глав

Б) Контрразведка перехватила несколько зашифрованных файлов, и подозревала, что это тексты написанных на Паскале вирусов. Позвали Шерлока Ивановича Холмского в надежде, что тот расшифрует их. Шерлок Иванович предположил, что шифровали методом Юлия Цезаря (вспомните главу 24). Нужен ключ! После недолгих раздумий Шерлок Иванович создал программу для подбора ключей к таким текстам. Повторите ещё один «подвиг контрразведчика», или слабо? Подсказка: в таких файлах после расшифровки обязательно встречаются ключевые слова **BEGIN** и **END** — воспользуйтесь этим.

В) Рейтинговое голосование. Избирательный закон Иксляндии даёт каждому избирателю право голосовать за всех кандидатов, расставляя их в порядке своего предпочтения. Побеждает кандидат, набравший наименьшую сумму мест (если таковых несколько, то проводят второй тур). Предположим, баллотируются четыре кандидата с номерами 1-4, а бюллетени содержат следующие предпочтения избирателей:

```
3 4 2 1
2 4 3 1
4 1 3 2
```

Здесь первый кандидат набирает сумму 10, второй — 8, третий — 7, четвертый — 5. Таким образом, побеждает четвертый кандидат в списке.

Количество кандидатов известно и равно пяти. Ваша программа принимает файл, каждая строка которого содержит 5 чисел — данные одного бюллетеня. Надо выдать список победителей голосования (одного или нескольких).

Глава 61
«Кубики» программиста (ООП)

А на десерт отведайте волшебного объектно-ориентированного программирования (ООП). Эта технология преобразила наше ремесло, — так пройдем ли мимо?

Фокус-покус

Вначале дадим слово Паскалю. Если вы работаете в IDE Borland Pascal, введите и запустите следующую программку. Только убедитесь в том, что библиотечный файл «APP.TPU» доступен компилятору (обычно он находится в папке «UNITS», где собраны фирменные библиотечные модули).

```
      {P_61_1 - демонстрация работы Turbo Vision }
uses App;
var A : TApplication;
begin
      A.Init;
      A.Run;
      A.Done;
end.
```

Как это работает? — не спрашивайте, ведь программка простейшая, что ещё надо? После ее запуска вам откроется следующая картина (рис. 152).

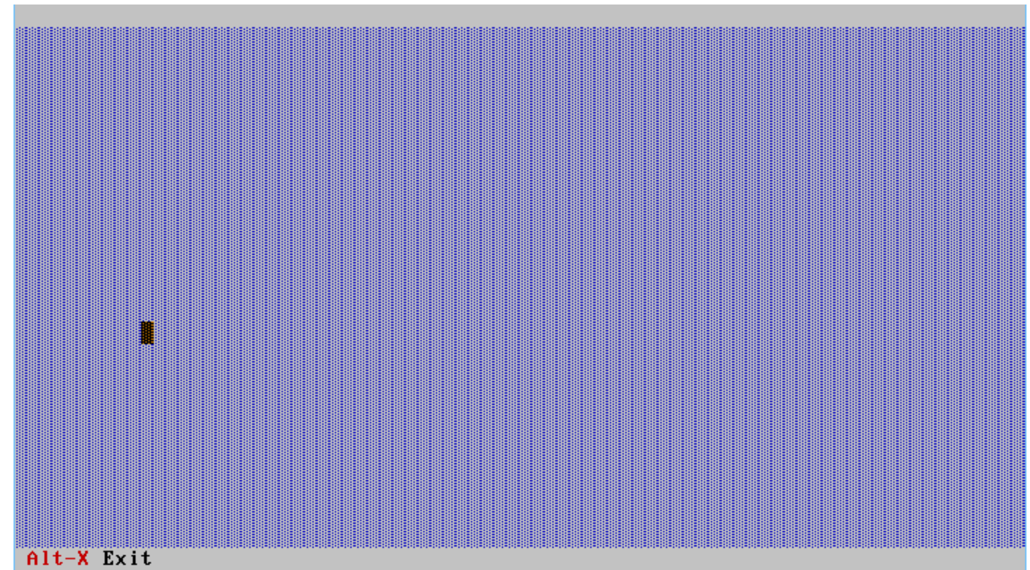


Рис. 152 – Окно программы MyApp

Неужто сломалась IDE? В самом деле, где меню и окно с текстом программы? Да и статусная строка совсем не та. Но мышка по-прежнему бегает, и клавиатура

жива. Можно щелкнуть по слову «Exit» и выйти из программы, или сделать то же самое нажатием *Alt+X*. Да, друзья, вы наблюдаете действие той самой малюсенькой программки! Ее поведение чудесно, но возможности этим не исчерпаны. Своей мощью она обязана объектам из библиотеки Turbo Vision, на которой построена вся IDE Borland Pascal.

Вместо паяльника

Все восторгаются объектным программированием. А откуда оно взялось?

Представьте, что при покупке телевизора вместо работающего изделия вам вручают его схему, коробку с деталями и предлагают собрать телевизор самому! Примерно в таком же положении находились когда-то и программисты. Используемые ими библиотеки хранили массу полезных процедур и функций, — своего рода схемы для сборки программ. А деталями были обрабатываемые данные. Программист распределял эти данные в программе, придавая им нужную структуру, применял к данным в надлежащем порядке процедуры и функции, отслеживая при этом влияние одних данных на другие. Работа эта сродни сборке телевизора! Плодовитость программистов и качество их изделий оставляли желать лучшего, ненадежные программы было трудно править, да и пользоваться ими было неудобно.

Так вернемся к телевизору; что нужно знать мне, его владельцу? Всего лишь несколько кнопок: включить, отключить, выбрать канал, настроить громкость. И всё! Остальное пусть будет спрятано. Вот бы и в программировании добиться такого удобства! Изобретатели ООП стремились именно к этой цели — упростить работу со сложной совокупностью данных. Они догадались объединить в одно целое данные и процедуры, их обрабатывающие. Совокупность данных и процедур назвали объектом.

На первый взгляд объект похож на запись. Но, в отличие от записи, большинство данных и процедур объекта спрятано внутри и не видно за его пределами. Снаружи доступно лишь то, что интересует пользователя объекта, — в этом объект похож на модуль (или на собранный телевизор).

На трех китах

Основу ООП составляют три идеи, три «кита», а именно:

- инкапсуляция;
- наследование;
- полиморфизм.

Рассмотрим их в этом порядке.

Примечание. Для примеров этой главы настройте компилятор в режим, совместимый с Borland Pascal.

Инкапсуляция

Инкапсуляция — это объединение данных и обрабатывающих их процедур. Рассмотрим простой пример: построим объект для хранения и обработки информации о человеке. Человеку свойственны такие атрибуты как год рождения, имя и фамилия. Поставим цель упростить работу с этими атрибутами, — создадим объект, способный хранить и распечатывать эту информацию.

Объявление объекта

Объявление объекта похоже на объявление записи, с той разницей, что ключевое слово **RECORD** заменяют словом **OBJECT**. В Delphi и совместимом с ним режиме Free Pascal применяют ключевое слово **CLASS**. Итак, застолбим место хранения информации о человеке тремя полями, как это показано ниже.

```
type  TPerson = object
            mBearing : integer;     { год рождения }
            mName    : string;      { имя }
            mFam     : string;      { фамилия }
      end;
```

Здесь объявлен тип данных **TPerson** (персона), содержащий три поля с данными о человеке. Для распечатки данных учредим процедуру по имени **Report**. Но процедура эта особая! Её заголовок помещен внутрь описания объекта следующим образом:

```
type  TPerson = object
            mBearing : integer;     { год рождения }
            mName    : string;      { имя }
            mFam     : string;      { фамилия }
            procedure Report;       { процедура распечатки объекта }
      end;
```

Методы

Процедуры и функции, объявленные внутри объекта, называют методами объекта. Методы, как и поля, — неотъемлемая часть объекта. Но объявить метод недостаточно, надо создать и его тело или, как принято говорить, реализацию метода. Реализация — это подпрограмма (процедура или функция), которая от обычной подпрограммы отличается заголовком: там к имени метода добавляется приставка, указывающая тип объекта, которому принадлежит метод. Реализация метода **Report** будет очень простой:

```
procedure TPerson.Report;
begin
      Writeln(mBearing:6, 'Фамилия: '+mFam:20, '  Имя: '+mName);
end;
```

Процедура распечатывает атрибуты человека. Но откуда она берет их? — эти данные не передаются через параметры, и не хранятся в глобальных переменных. Они объявлены как поля объекта, и этого достаточно, чтобы метод объекта получил доступ к ним.

Инициализация, конструктор

Поля объекта, как любые переменные, нуждаются в инициализации. Как осуществить её? Можно присвоить значения полям так, как это делается для записей.

```
var P : TPerson;        { переменная-объект }
begin
      P.mFam:='Сидоров';
      P.mName:= 'Тимофей';
end.
```

Но, когда полей много, вы забудете что-то, и в этом слабость идеи. Куда надежней учредить особый метод для инициализации полей. Такой метод и назван особо — конструктор. Вместо слова **PROCEDURE** перед именем конструктора так и пишут: **CONSTRUCTOR**. Назвать конструктор можно как угодно, но по традиции ему дают имена **Init** (инициализировать) или **Create** (создать). Например, для нашего объекта объявить конструктор и реализовать его тело можно так:

```
type  TPerson = object
            { . . . }
            { заголовок конструктора внутри объекта }
            constructor Init(aBearing: integer; const aName, aFam : string);
      end;
            { реализация конструктора }
constructor TPerson.Init(aBearing: integer; const aName, aFam : string);
begin
      mBearing:= aBearing;    mName:= aName;    mFam:= aFam;
end;
```

В этом примере конструктор **Init** копирует три своих параметра в поля объекта. Теперь переменную-объект **P** можно инициализировать вызовом конструктора.

```
var P : TPerson;        { переменная-объект }
begin
      P.Init(1995, 'Мария', 'Рыбкина');
```

Так ни одно поле объекта не будет пропущено, — за этим присмотрит компилятор!

Вот пока всё, что следует сказать об инкапсуляции. Приведенный ниже пример P_61_2 демонстрирует объект типа **TPerson**: здесь описана его структура и реализация методов, а затем объявлены две переменные, выполнена их инициализация и распечатка полей.

```
      { P_61_2 Программа с применением объекта типа «человек» (персона) }
type  TPerson = object
            mBearing : integer;     { год рождения }
            mName    : string;      { имя }
            mFam     : string;      { фамилия }
            constructor Init(aBearing: integer; const aName, aFam : string);
            procedure Report;       { процедура распечатки объекта }
      end;
      {--- Реализация двух методов объекта ---}
constructor TPerson.Init(aBearing: integer; const aName, aFam : string);
begin
      mBearing := aBearing;   mName    := aName;      mFam     := aFam;
end;
procedure TPerson.Report;
begin
      Writeln(mBearing:6, 'Фамилия: '+mFam:20, '  Имя: '+mName);
end;
var P1, P2 : TPerson;   { две переменных объектного типа }
begin       {--- Главная программа ---}
      P1.Init(1985, 'Иван', 'Грозный');
      P2.Init(1995, 'Мария', 'Рыбкина');
      P1.Report;
      P2.Report;
      Readln;
end.
```

Наследование

Кажется, что инкапсуляция не упростила программирование. Да, верно, если рассматривать её в отрыве от других механизмов ООП: наследования и полиморфизма. Выигрыш увидим, когда в ход пойдут все рычаги.

Наследование даёт возможность создавать новые типы объектов на основе существующих. Вновь создаваемые типы объектов — потомки — приобретают в наследство поля и методы своих предков. И вдобавок могут содержать новые поля и методы, а также изменять унаследованные.

Например, взяв почти готовый объект — окно в библиотеке — программист добавляет к нему свои поля, методы и получает другой тип окна, работающий схожим образом, но с учетом потребностей программиста. При этом не придётся вникать в тонкости объекта-предка, достаточно ознакомиться лишь с несколькими основными методами (подобно тому, как телезрителю хватает лишь нескольких кнопок). Не нужен даже исходный текст модуля с описанием объекта-предка!

И это не всё. Постройка одних объектов на основе других формирует иерархию родственных объектов. С разными объектами этой иерархии можно обращаться сходным образом — это и есть полиморфизм. Буквальный перевод этого слова — «многоструктурность» — почти ничего не объясняет. Принципы наследования и полиморфизма легче понять на примере.

Приборостроение

Я знаю, чем напичкано ваше жилище — электрическими приборами. И простыми, такими, как лампочка или утюг. И сложными: телевизор, стиральная машина, компьютер, наконец. Взглянем на них глазами программиста: любой такой прибор, выражаясь языком ООП, обладает, по крайней мере, двумя общими методами: **включить** и **отключить**. В разных приборах эти операции выполняются по-разному, но в целом они сходны. Можно сказать, что эти методы — общие свойства всех электроприборов.

Если бы приборы создавал программист, то построил бы их на базе общего предка — абстрактного (воображаемого) электрического прибора. Этот прибор обладал бы двумя методами: **включить** и **отключить**. На Паскале этот абстрактный электроприбор можно объявить так:

```
type  Электроприбор = object
            procedure   Включить; virtual;
            procedure   Отключить; virtual;
      end;
```

Здесь встречаем новое волшебное словечко — **VIRTUAL**, что значит «воображаемый». Это ключевое слово Паскаля следует за объявлением тех методов объекта, которые разрешено изменять в его наследниках. Изменение метода в наследниках называют переопределением метода. Итак, слово **VIRTUAL** указывает компилятору, что в наследниках методы включения и отключения прибора могут быть изменены в соответствии с особенностями этих наследников. К примеру, лампочка и телевизор включаются по-разному.

Учредив абстрактный электроприбор, построим на нём прибор «чисто конкретный», например, телевизор.

```
type  Телевизор = object (Электроприбор)
          procedure   Включить; virtual;
          procedure   Отключить; virtual;
          procedure   Выбрать_канал;
          procedure   Настроить_громкость;
          procedure   Настроить_яркость;
      end;
```

Поскольку телевизор порожден от электроприбора, название его предка — «электроприбор» — указано в скобках за ключевым словом **OBJECT**. Наследник обязан помнить о предке, ссылаться на него, иначе не получит своего наследства — полей и методов. Виртуальные методы **включить** и **отключить** объявлены в наследнике точно так же, но будут реализованы иначе. К ним добавлены ещё три метода, характерные именно для телевизора. Схожим образом строятся и другие «конкретные» электроприборы. В результате сформируется иерархия родственных объектов, показанная на рис. 153.

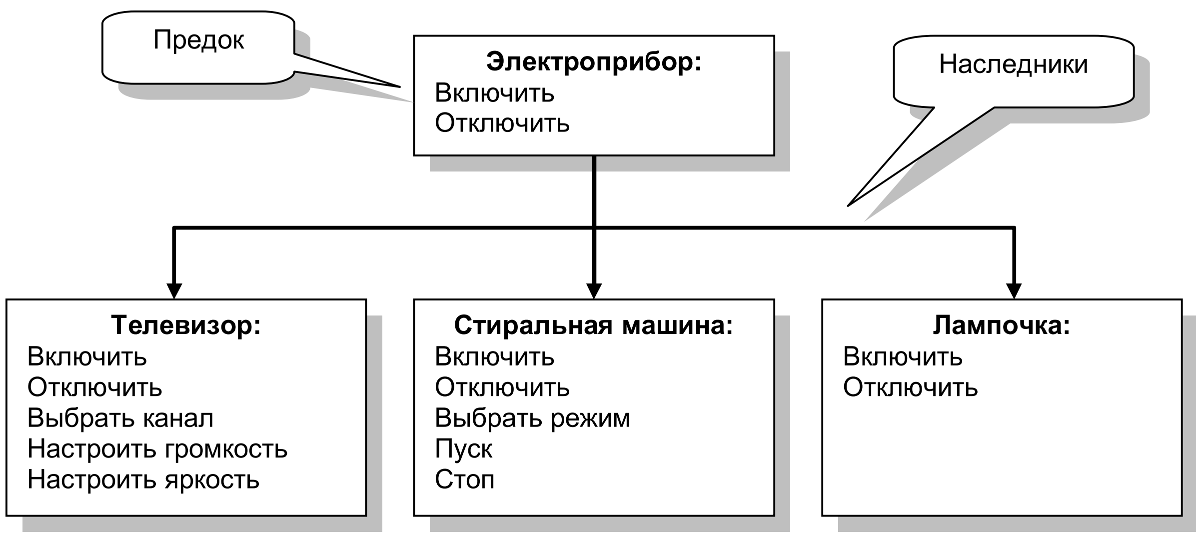


Рис. 153 – Иерархия электрических приборов

Гражданское строительство

Но всё это лишь присказка, теперь испытаем наследование и полиморфизм в деле. Создадим на базе спроектированного ранее объекта **TPerson** (человек) два новых типа данных: военнослужащий (**TMilitary**) и гражданский чиновник (**TCivil**), иерархия этих типов изображена на рис. 154. Эти новые типы «людей» будут содержать дополнительные поля с характерной для наследников информацией. Вдобавок изменим конструктор **Init** и метод **Report** с тем, чтобы

учесть наличие новых полей. Конструктор будет содержать дополнительный параметр, а процедура распечатки — выводить на экран ещё одно поле объекта.

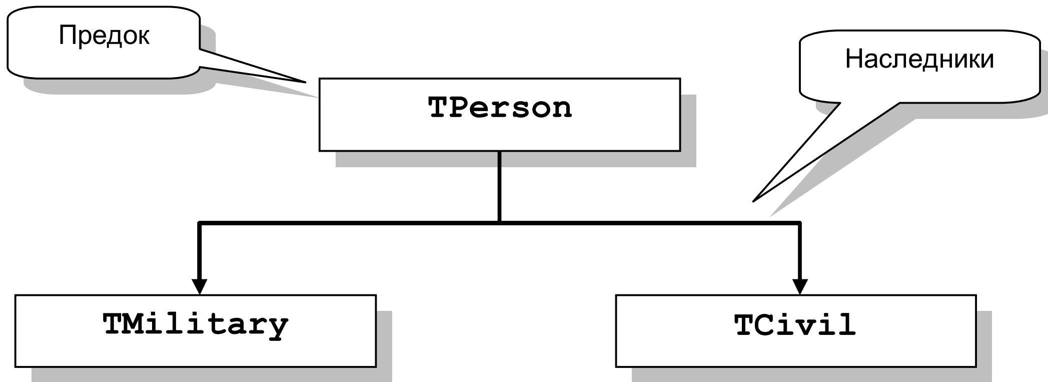


Рис. 154 - Иерархия «человеческих» типов

Объявление

Начнем с военнослужащего, чем разнится он от простых смертных? Гордым воинским званием — от рядового до маршала. Для хранения воинского звания в объекте **TMilitary** добавим строковое поле **mRank** (**Rank —** звание). Ясно, что при создании объекта конструктором надо указать этот элемент. Добавим ещё один параметр конструктору объекта **Init** — параметр **aRank**, и тогда заголовок конструктора в объекте **TMilitary** станет таким.

```
constructor Init(aBearing: integer; const aName, aFam, aRank : string);
```

В новом конструкторе больше параметров, и работать он будет, в сравнении с предком, чуть иначе. Другими словами, в наследнике он переопределен. А если так, то где же волшебное слово **VIRTUAL**? Его здесь нет и не должно быть, поскольку конструктор виртуален по определению.

Теперь обратимся к процедуре распечатки **Report**. В наследнике она, кроме прочего, должна распечатать поле воинского звания, а значит, будет переопределена. Поэтому и объявлена виртуальной, причем и в наследнике **TMilitary**, и в его предке **TPerson**. Так сделано потому, что лишь виртуальный метод предка может быть виртуальным у наследника: виртуальность передается по наследству. С учетом всего сказанного, объявления типов **TPerson** и **TMilitary** теперь будут такими.

```
    TPerson = object
          mBearing : integer;     { год рождения }
          mName    : string;      { имя }
          mFam     : string;      { фамилия }
          constructor Init(aBearing: integer; const aName, aFam : string);
          procedure Report; virtual;
    end;

    TMilitary = object (TPerson)
          mRank     : string;     { воинское звание }
          constructor Init(aBearing: integer; const aName, aFam,
                                              aRank : string);
          procedure Report; virtual;
    end;
```

Подытожим все изменения. В предке **TPerson** процедура **Report** стала виртуальной. В наследнике **TMilitary** добавлено поле **mRank**, а также изменены два метода: конструктор и процедура **Report**.

Отселение в отдельный модуль

Настало время реализовать методы наследника. Но прежде, чем взяться за это, совершим одно полезное дельце — переместим объект-предок **TPerson** в отдельный модуль. Именно так поступают профессионалы, создавая библиотеки объектов. Порядок создания программного модуля подробно изложен в главе 59, вкратце я напомню основные шаги.

Итак, создайте новый файл, перенесите туда через буфер обмена объявление типа **TPerson** и реализацию его методов. Объявление объекта разместите в секции **INTERFACE** модуля, а реализацию — в секции **IMPLEMENTATION**. И не забудьте объявить виртуальной процедуру **Report**. Дайте модулю имя **PERSON** и сохраните под именем «PERSON.PAS». У вас получится файл, показанный ниже. В нём объявлен ещё один тип данных — указатель на объект **PPerson**, но к нему обратимся позже.

```
unit  Person; { Модуль, содержащий описание и реализацию объекта «ЧЕЛОВЕК» }

interface

type  PPerson = ^TPerson;     { указатель на объект «ЧЕЛОВЕК» }
      TPerson = object
            mBearing : integer;     { год рождения }
            mName    : string;      { имя }
            mFam     : string;      { фамилия }
            constructor Init(aBearing: integer; const aName, aFam : string);
            procedure Report; virtual;
      end;

implementation

      {--- Реализация объекта «ЧЕЛОВЕК» ---}
constructor TPerson.Init(aBearing: integer; const aName, aFam : string);
begin
  mBearing := aBearing;
  mName    := aName;
  mFam     := aFam;
end;

procedure TPerson.Report;
begin
  Writeln(mBearing:6, 'Фамилия: '+mFam:20, '  Имя: '+mName);
end;
end.
```

Теперь то, что переехало в модуль **Person**, из первичного файла проекта удалим, и добавим в начале программы ссылку для импорта модуля.

```
USES Person;
```

Сохраните новую версию файла под именем P_61_3 и убедитесь, что она компилируется без ошибок. Затем вставьте в первичный файл приведенное ранее объявление типа для наследника **TMilitary**. В итоге заготовка будущей программы станет такой.

```
{ P_61_3 - Демонстрация принципов наследования и полиморфизма }

uses Person;            { Объект TPerson импортируется из модуля Person }
type  { объект «ВОЕННОСЛУЖАЩИЙ» }
  TMilitary = object (TPerson)
    mRank : string;     { воинское звание }
    constructor Init(aBearing: integer; const aName, aFam, aRank : string);
    procedure Report; virtual;
  end;

begin
end.
```

Реализация методов

Отсюда приступим к реализации переопределенных методов нового объекта. Начнем с конструктора. Конечно, он мог бы повторить действия объекта-предка, но это неразумно. Ведь цель объектной технологии — упростить программирование, избежать повторов, не так ли? Избежать повтора здесь очень просто: внутри конструктора наследника вызовем конструктор предка, передав ему нужные параметры.

```
      TPerson.Init(aBearing, aName, aFam);
```

Вызов конструктора предка содержит имя этого предка — префикс **TPerson**. Обращение потомка к методам предка — обычная практика. По этой причине в Паскале учреждено ключевое слово **INHERITED** — «унаследованный». Если предварить им вызов унаследованного метода, то префикс с именем предка станет излишним.

```
      inherited Init(aBearing, aName, aFam);
```

В таком вызове унаследованного метода трудней ошибиться. Ведь иерархия предков может быть глубокой, а представленный здесь способ вызывает метод непосредственного (ближайшего) предка, что обычно и требуется.

Итак, поля, унаследованные от предка, инициализированы конструктором, унаследованным от него же. Оставшееся поле **mRank** заполним как обычно, в результате конструктор наследника примет такой вид.

```
constructor TMilitary.Init(aBearing: integer; const aName, aFam,
                                               aRank : string);
begin
      inherited Init(aBearing, aName, aFam);   { вызов метода предка }
      mRank:= aRank;
end;
```

Переходим к методу **Report** наследника. Здесь, вдобавок к прочим данным, надо распечатать ещё и воинское звание. Прочие данные распечатаем унаследованным методом **Report**, а воинское звание — дополнительным оператором печати. Вы уже догадались, что реализация метода будет такова.

```
procedure TMilitary.Report;
begin
      inherited Report;      { вызов метода предка }
      Writeln('Воинское звание: '+mRank);
end;
```

Породив «военного человека», перейдём к мирному строительству: создадим объект, играющий роль гражданского служащего. Назовем его **TCivil**, а род его пойдет от того же предка **TPerson**. У гражданских своя гордость и своя служебная лестница, ступеньки которой — категории — нумеруются числами. Хранить информацию о карьерном росте будем в числовом поле, назовем его **mLevel** — «уровень». Так же, как и для военного, нам придется дополнить конструктор объекта и метод распечатки **Report**. Ход рассуждений будет прежним, а потому не буду повторять его, сделайте эту работу сами.

Сотворив наследников «человека» — объекты **TMilitary** и **TCivil**, мы почти разобрались в механизме наследования. А где же полиморфизм? В чем он проявляется? Для ответа обратимся к динамическим объектам.

Динамические объекты

Динамические переменные знакомы нам ещё с 52-й главы. Указатели на объекты ничем не отличаются от таковых для других типов данных. Например, указатель на тип **TPerson** объявляется так:

```
type  PPerson = ^TPerson;
```

Теперь можно объявить переменную этого типа, взять для неё память в куче, а затем инициализировать поля конструктором.

```
var   P : PPerson;            { указатель на объект }
begin
      New(P);                            { выделение памяти в куче }
      P^.Init(1985, 'Иван', 'Грозный');  { инициализация объекта }
```

В серьезных программах объекты обычно используют динамически, а выделение памяти и инициализацию выполняют на каждом шагу. Потому в Паскаль введена функция **New**, совмещающая эти действия. Функция **New** подобна процедуре **New**, но вдобавок вызывает ещё и конструктор объекта. Функция принимает два странных параметра: тип-указатель на объект и конструктор этого объекта, а возвращает указатель на созданный объект. Так, динамический объект типа **TPerson** может быть порожден и инициализирован одним оператором.

```
      P:= New(PPerson, Init(1985, 'Иван', 'Грозный'));
```

Обратите внимание, что первый параметр функции — это тип-указатель **PPerson**, а не тип объекта **TPerson**!

Примечание. В языке Delphi и совместимом с ним режиме Free Pascal применяют иной синтаксис вызова конструктора, например:

```
var   P : TPerson;      { это указатель на объект! }
. . .
P:= TPerson.Init(1985, 'Иван', 'Грозный'); { создаём динамический объект }
```

Дело в том, что все объекты в Delphi — это динамические переменные, и переменная типа **TPerson** является указателем на объект. Для создания таких объектов применяют не функцию **New**, а вызов конструктора с префиксом, указывающим тип объекта.

Полиморфизм

Теперь, после знакомства с динамическими объектами, вернемся к полиморфизму. Предположим, что в программе объявлены указатели трех типов.

```
var   P1 : PPerson;     { указатель на предка }
      P2 : PMilitary;   { указатель на потомка }
      P3 : PCivil;      { указатель на потомка }
```

Здесь **P1** является указателем на предка, а **P2** и **P3** — на разных его потомков. Отчасти полиморфизм состоит в том, что указателю на предка разрешено присваивать указатели на любого его потомка, стало быть, следующие операторы не вызовут протеста компилятора.

```
      P1:= P2;
      P1:= P3;
```

Думаете, пустая формальность? Зря вы так! Воистину здесь таится глубокий смысл, поскольку через указатель на предка можно вызывать методы его потомков. Но при условии, что эти методы унаследованы от предка как виртуальные. Так, в следующем примере указателю **P1** трижды присваиваются указатели на объекты разных типов: сначала на предка **TPerson**, а затем на двух его потомков, после чего всякий раз вызывается виртуальный метод **Report**. Но в реальности происходит вызов трех разных методов **Report** — соответственно типу объекта, на который в текущий момент ссылается указатель **P1**. Так срабатывает механика полиморфизма!

```
      P1:= New(PPerson, Init(1985, 'Иван', 'Грозный'));
      P1^.Report;       { вызывается TPerson.Report }
      P1:= New(PCivil, Init(1995, 'Мария', 'Рыбкина', 12));
      P1^.Report;       { вызывается TCivil.Report }
      P1:= New(PMilitary, Init(1985, 'Андрей', 'Быков', 'Майор'));
      P1^.Report;       { вызывается TMilitary.Report }
```

Кажется, что полиморфизм одушевляет объект и делает его умнее: объект сам «понимает», как ему исполнить то, или иное желание программиста. Тот лишь вызывает нужный метод, не вникая в детали. Это похоже на управление телевизором или другим прибором. Подайте им напряжение, и все они включатся: хоть и по-разному, но каждый по-своему правильно.

Но мощная механика полиморфизма срабатывает лишь для родственных объектов, состоящих в отношении предок-потомок. Именно в таких отношениях находятся созданные нами объекты. А вот пример иного рода.

```
type  TA = object
            constructor Init;
            procedure Report; virtual;
      end;

      TB = object
            constructor Init;
            procedure Report; virtual;
      end;
```

Здесь объявлены два типа объектов с одноименными виртуальными методами. Но полиморфизмом тут и не пахнет, поскольку объекты не родственны меж собой!

В завершение темы изучите программу P_61_3, где собрано всё, что было сказано о «человечьих» объектах.

```
{ P_61_3 - Демонстрация принципов наследования и полиморфизма }

uses Person;            { Объект TPerson импортируется из модуля Person }

type  PMilitary = ^TMilitary;      { указатель на объект «ВОЕННОСЛУЖАЩИЙ» }
      TMilitary = object (TPerson)
        mRank     : string;  { воинское звание }
        constructor Init(aBearing: integer; const aName, aFam,
                                          aRank : string);
        procedure Report; virtual;
      end;
      PCivil = ^TCivil;      { указатель на объект «ГРАЖДАНСКИЙ СЛУЖАЩИЙ» }
      TCivil = object (TPerson)
        mLevel     : integer;      { должностная категория }
        constructor Init(aBearing: integer; const aName, aFam : string;
                                          aLevel: integer);
        procedure Report; virtual;
      end;

      {--- Реализация объекта «ВОЕННОСЛУЖАЩИЙ» ---}

constructor TMilitary.Init(aBearing: integer; const aName, aFam,
                                          aRank : string);
begin
  inherited Init(aBearing, aName, aFam);
  mRank:= aRank;
end;

procedure TMilitary.Report;
begin
  inherited Report;
  Writeln('Звание: '+mRank);
end;
```

```
      {--- Реализация объекта «ГРАЖДАНСКИЙ СЛУЖАЩИЙ» ---}
constructor TCivil.Init(aBearing: integer; const aName, aFam : string;
                                               aLevel: integer);
begin
  inherited Init(aBearing, aName, aFam);
  mLevel:= aLevel;
end;

procedure TCivil.Report;
begin
  inherited Report;
  Writeln('Категория: ', mLevel);
end;

var Persons : array[1..3] of PPerson;    { массив указателей на ПРЕДКА }
    i : integer;

begin      {--- Главная программа ---}
  { Массив заполняется объектами РАЗНЫХ, но родственных типов }
  Persons[1]:= New(PPerson, Init(1985, 'Иван', 'Семенов'));
  Persons[2]:= New(PCivil, Init(1995, 'Мария', 'Рыбкина', 12));
  Persons[3]:= New(PMilitary, Init(1985, 'Андрей', 'Быков', 'Майор'));
  { В ходе распечатки вызывается метод ФАКТИЧЕСКОГО объекта }
  for i:=1 to 3 do Persons[i]^.Report;
  Readln;
end.
```

Сокрытие полей и методов

Вы догадываетесь, из чего строят современные программы? Из сотен «умных» объектов, образующих ветвистую иерархию родственных связей, которая открывает простор полиморфизму.

Многие объекты фирменных библиотек — это полуфабрикаты, требующие лишь небольшой настройки под конкретное применение. В ходе такой настройки программист добавляет к базовому объекту свои поля и методы. И здесь порой случается то же, что при использовании библиотечных модулей: имя, назначенное программистом, может совпасть с уже объявленным именем в предке. И тогда имена могут конфликтовать. В библиотечных модулях эта проблема решается скрытием большей части переменных, процедур и функций в невидимой извне секции реализации **IMPLEMENTATION**.

Схожий прием используют и в объектном программировании. Поля и методы, доступ к которым наследникам не нужен, прячут в объекте-предке так, что они

становятся невидимыми за пределами предка. И тогда спрятанные имена можно использовать в наследниках повторно по иному назначению. Не будет ли здесь путаницы? Нет, поскольку методы предка не знают о новых именах и обращаются к старым. А методы наследника не видят старых имен и обращаются к новым. Разумеется, что разработчик объекта-предка тщательно отбирает те поля и методы, которые потребуются создателям потомков.

Сокрытие имен объекта организовано очень просто: в объявление объекта вставляют ключевые слова **PRIVATE** (личный) и **PUBLIC** (общедоступный). Эти слова разбивают объявление объекта на две части — приватную и общедоступную, например:

```
type  TParent = object        { объект-предок }
      private
            A, B : integer;
            function Calc(arg: integer): integer;
      public
            Constructor Init(a, b : integer)
            function GetSum: integer; virtual;
      end;
```

Здесь поля **A** и **B**, а также функция **Calc**, скрыты от взоров потомков. Поэтому программист, импортировавший объект типа **TParent**, может спокойно добавить в него свои поля или методы с теми же самыми именами, например, так:

```
type  TChild = object (TParent)     { объект-наследник }
            A, B : string;
            procedure Calc;
            . . .
      end;
```

Здесь в потомке поля **A** и **B** имеют другой тип, а имя **Calc** принадлежит не функции, а процедуре. Но поля и методы предка с этими же именами всё ещё существуют! Но доступны только предку, вот и всё.

А если в объявлении объекта не указаны ключевые слова **PRIVATE** и **PUBLIC**? Тогда все его поля и методы по умолчанию будут общедоступными.

Итак, мы рассмотрели идеи и механизмы, лежащие в основе объектно-ориентированного программирования. К сожалению, одной главы маловато для освоения всех тонкостей этой технологии — на то есть другие книги. Объектные технологии — это настоящее и будущее программирования, не жалейте времени на их освоение. Здесь, как и во всем, важна практика. В начале главы я дал пример использования библиотеки Turbo Vision. Изучение этой и ей подобных библиотек

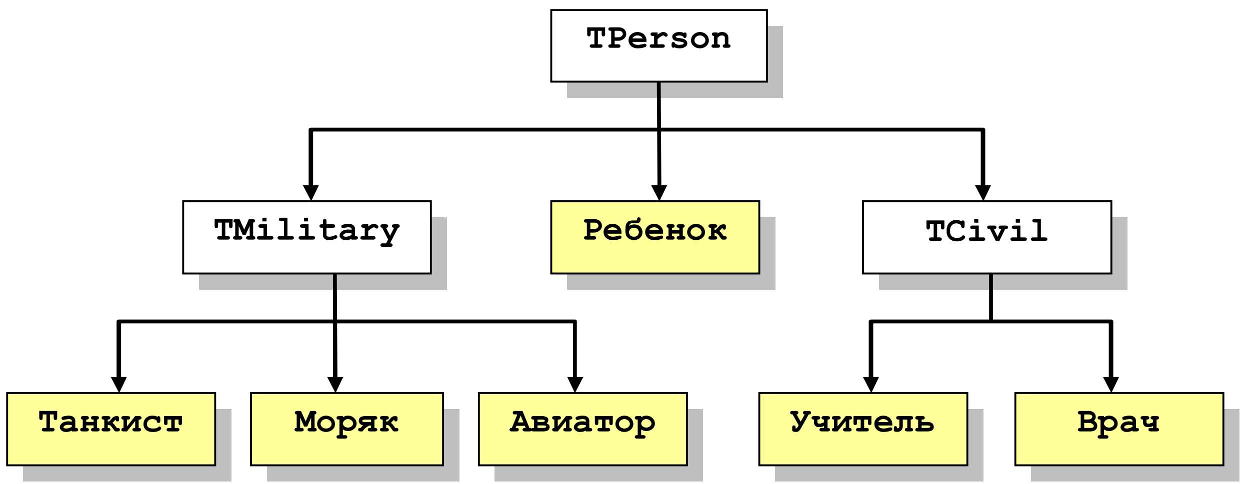
— прекрасный способ освоения объектной технологии, подробное описание библиотеки можно найти в Сети и в литературе.

Итоги

- Объектно-ориентированное программирование – это современная технология быстрой разработки крупных и надежных программ.
- Объект – это сложный тип данных, совмещающий в себе собственно данные и процедуры, обрабатывающие их.
- Объектно-ориентированное программирование основано на трех дополняющих друг друга механизмах: инкапсуляции, наследовании и полиморфизме.
- Инкапсуляция – это объединение данных и процедур, их обрабатывающих, в едином объекте.
- Наследование позволяет создавать новые типы объектов на базе существующих. Так создается иерархия родственных объектов.
- Полиморфизм состоит в схожем поведении объектов родственных типов.

А слабо?

А) Разработайте иерархию «человечьих» объектов в соответствии со следующим рисунком (новые типы объектов выделены цветом).



Дайте новым типам объектов подходящие имена, дополните их надлежащими полями, переопределите конструкторы и метод **Report**. Затем исследуйте механизм полиморфизма на предке и всех его потомках.

Б) Мощность множеств в Паскале не превышает **256**, и часто этого бывает недостаточно. Сконструируем свой тип множеств, назовем его **TBigSet**, мощность которого составит **65536** (соответствует диапазону для типа **Word**). Оформим это множество как объект.

```
type
  TSetArray = array [0..4096] of word;  { хранит 65536 бит (4096*16) }
  PSetArray = ^ TSetArray;               { тип-указатель на массив }

  TBigSet = object
      mArray : PSetArray;  { указатель на динамический массив }
      Constructor Init; { создает динамический массив mArray )
      Destructor Done;  { освобождает память, занятую массивом }
      procedure ClearAll; { опустошает множество }
      procedure SetAll; { делает множество полным }
      procedure Insert(N); { вставляет элемент N в множество }
      procedure Delete(N); { удаляет элемент N из множества }
      function Member(N):Boolean; { проверяет принадлежность N к множеству }
      function IsEmpty:Boolean; { проверяет пустоту множества }
      procedure Union(BS: TBigSet); { добавляет другое множество }
      procedure InterSect(BS: TBigSet); { формирует пересечение множеств }
      procedure Load(var F: text); { вводит множество из файла }
      procedure Save(var F: text); { выводит множество в текстовый файл }
  end;
```

Примените здесь сведения из главы 48, а также идеи из задачи 49-В (глава 49). Так, включение в множество и исключение из него элемента **N** может быть выполнено установкой и сбросом бита в массиве **mArray^**.

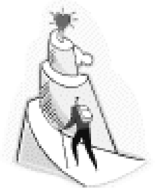
```
    mArray^[N div 16]:= mArray^[N div 16] or (1 shl (N mod 16))
    mArray^[N div 16]:= mArray^[N div 16] and not (1 shl (N mod 16))
```

Объединение с другим множеством **Union(BS:TBigSet)** можно сделать логическим суммированием массивов:

```
 for I:=0 to 4095 do mArray^[ I ]:= mArray1^[ I ] or BS.mArray^[ I ]
```

И так далее. Напишите реализацию всех методов объекта и примените его к решету Эратосфена и прочим задачам из главы 38.

Глава 62
Всё только начинается!



Мы у финишной черты, где принято подводить итоги. Нет, друзья, повременим с итогами, ведь для вас всё только начинается, — лучше обсудим ваши планы на будущее.

Крупицы мастерства

Чем зарабатывает программист? — создает программы. У хорошего мастера дело спорится, и товар его добротен. Как скорее достичь мастерства? За что браться, с чего начать? Вот несколько советов.

Постигайте языки программирования

Первым делом хорошенько оседлайте Паскаль — один из лучших языков программирования. Даже школьник, владеющий Паскалем, — это наполовину инженер, ведь мощная система программирования Delphi построена на этом языке. Часть пути к вершинам Паскаля мы преодолели вместе, но освоили далеко не все его возможности. Дальше ступайте сами: преодолев робость и сомнения, откройте «взрослый» учебник по Паскалю, — некоторые из таких учебников найдете в списке рекомендуемой литературы. Там вас ждет немало открытий!

А что другие языки? Среди них отметим Си — один из самых используемых. Но почему не Паскаль?

Языки Паскаль и Си — ровесники, они родились в начале 70-х годов прошлого века. Паскаль был задуман как строгий язык для надежного программирования, но на первых порах применялся лишь в образовании. Создатели языка Си преследовали иную цель, — им срочно понадобился незатейливый язык для появившихся в ту пору мини-ЭВМ. Надо заметить, что программы, написанные на Паскале, эти слабенькие ЭВМ переваривали с трудом, а строгие ограничения надежного Паскаля по рукам и ногам вязали ретивых системщиков. Потому сработанный на скорую руку простенький и ненадежный Си вдруг вынырнул вперед и захватил лидерство. А что было дальше? — об этом нельзя промолчать.

По мере того, как компьютеры становились мощнее, а программы сложнее, требования к их надежности возросли — ошибки программистов крайне дороги! Сторонники Си осознали необходимость типизации данных и прочих мер повышения надежности и позаимствовали эти идеи из Паскаля. С другой стороны слишком суровый контроль типов в Паскале был слегка ослаблен — в разумных пределах. Это и другие новшества, добавили языку гибкости, и Паскаль пробился в области, где безраздельно хозяйничал Си. Нынешние потомки Паскаля — языки Ada и Modula — применяют для создания надежных, ответственных программ (авиация, космос, вооружения). Но это секрет, о котором «настольные»

программисты не знают, поэтому для них C/C++ — «языки профессионалов». Впрочем, в «настольном» программировании эти языки ещё востребованы, хотя и вытесняются где-то более современными Java и C#.

Ещё одна популярная сфера — разработка сайтов. Тем, кто углубится в WEB-программирование, пригодятся PHP, Perl, JavaScript, Pyton и другие WEB-языки (они всё плодятся и плодятся!).

Особняком от «толпы» языков программирования держится Ассемблер (Assembler переводится как «сборщик»). Его относят к языкам низкого уровня, — в отличие от Паскаля, Си и многих других, причисляемых к языкам высокого уровня. За что же так «унижен» Ассемблер? За то, что оперирует с потрохами процессора: адресами памяти, регистрами, флагами. Программирующий на Ассемблере должен хорошо представлять устройство компьютера и процессора, — на низком уровне нужна высокая квалификация! Ассемблер — удел профессионалов, и к нему прибегают там, где другие языки не годятся.

Технологии – не упускайте их из виду!

Изучая современные языки, вы постигните и технологии разработки крупных проектов: модульное и объектное программирование. И это не всё. Технологии не стоят на месте, и новинки стремительно сменяют одна другую. Держите «руку на пульсе», следите за развитием технологий, — кто зазевался, тот отстал!

Изучайте типовые алгоритмы и структуры данных

Вам надо хорошо овладеть языками, но это не всё. Любой проект начинают с разработки алгоритмов. Многие типовые задачи давным-давно решены, и для них созданы эффективные алгоритмы. Не изобретайте велосипед, изучайте типовые алгоритмы и структуры данных. Некоторые из них вам уже знакомы: это сортировка, двоичный поиск, списки, графы, стеки и очереди. И, хотя нерешенным задачам счету нет, в основе их решений лежат типовые алгоритмы и структуры данных.

Расширяйте кругозор

Программист, не «въезжающий» в решаемую задачу, — самое жалкое существо на свете! Увы, такие бедолаги — не редкость, и на то есть причины, — ведь компьютеры распространились повсеместно, где их только нет!

Принимаясь за очередной проект, поневоле вникаешь в премудрости той сферы, для которой он предназначен. Это может быть электроника, механика, химия или экономика — всего не перечислить. И всякий раз ищешь общий язык с экспертами в данной области. Ведь постановка задачи — это результат встречного движения программиста и заказчика. Значит, программист должен знать всё? Но это невозможно! Да, всезнайкой вам не быть, но понемногу обо всем знать необходимо. Школяр! не пренебрегай науками, тебе сгодится всё!

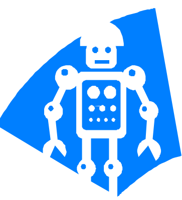
Угождайте пользователю

Программы для персональных компьютеров, как правило, общаются с людьми напрямую. А утомленному человеку свойственно ошибаться. Радейте о пользователе, — удобный и понятный интерфейс вашей программы должен радовать глаз, ограждая человека от возможных ошибок.

Проверяйте входные данные

Удобный и надежный интерфейс — это прекрасно, но даже он не гарантирует верного ввода исходных данных. Ошибки более вероятны, когда данные читаются из подготовленного вручную файла. Неверные данные могут «уронить» программу (например, при попытке деления на ноль). Громкое «падение» программы, сопровождаемое английскими «ругательствами», устрашит пользователя и породит в его душе сомнение в вашем мастерстве. Проверяйте вводимые данные на допустимость и непротиворечивость. Обнаружив ошибку, ваша программа должна вразумительно сообщить об этом пользователю и подсказать пути решения проблемы.

Тщательно тестируйте свои творения



Ошибки проявляются иногда так редко, что автор программы за это время несколько раз сменяет работу. Чем раньше вы обнаружите слабости своей поделки, тем лучше. Испытайте ее на самых разных наборах исходных данных — и правильных, и неправильных.

Комментируйте свои действия

В борьбе с ошибками не гнушайтесь ничем. Даже комментарии помогают. Поясняйте каждый крупный фрагмент: процедуру, блок операторов, а то и отдельный оператор. Если комментарий говорит о вашем намерении, то операторы показывают, что вы действительно сделали. Ошибка порождает противоречие между комментарием и оператором. Будучи незамеченным в первый момент, это несоответствие будет обнаружено вами или другими при повторном чтении программы.

Мы нередко возвращаемся к готовой программе спустя месяцы и годы после ее внедрения. За это время забывается почти всё, что придумали. А если вашу программу будет править кто-то другой... — я не завидую этому парню! И здесь вновь выручат комментарии.

Роль комментариев выполняют и выразительные имена, облегчающие понимание программы. Обычно программисты придумывают их на основе английских слов — ведь это международный язык. Следуйте некоторой системе в назначении имен, например, той, которая предложена в этой книге. Напомню о некоторых наших договоренностях. Имена для типов данных, констант и переменных начинаем с определенных букв, а именно:

- **C** – для констант;
- **T** – для типов данных;
- **a** – для аргументов процедур и функций;
- **m** – для полей записей и объектов.

Переменным лучше давать имена существительных, а для процедур и функций подходят глаголы. Глобальным переменным предпочтительно давать длинные развернутые имена. Наоборот, локальные переменные, назначение которых очевидно (счетчики циклов, временные значения и тому подобное), лучше называть коротко — одной-двумя буквами.

«Вылизывайте» структуру программы

Мастерское владение языком программирования и ясное понимание поставленной задачи — вот основа вашего успеха. Но что отличает профессионала? Умение распределять сложность в пространстве программы. Ваша процедура или функция вышла громоздкой и запутанной? Так разбейте её на ряд простых. Не увлекайтесь глобальными переменными, — для их объявления нужны веские основания. Лучше, когда процедуры и функции принимают данные через параметры.

Программисты, на старт!

По многим школьным предметам проводят олимпиады, в том числе по информатике. Цель этих олимпиад — выявить самых способных. Профильные факультеты ВУЗов с удовольствием принимают победителей олимпиад на льготных условиях. Ради этого можно постараться, не так ли? Если вам это интересно, я открою некоторые тайны олимпиад по информатике.

Этапы и участники олимпиад

Олимпиады по информатике (программированию) проводят в несколько этапов, — на школьном, районном, областном, зональном, всероссийском и международном уровнях. Разумеется, что в следующий этап выходят победители предыдущего.

Участники соревнуются в двух возрастных группах: старшей (11-й класс), и младшей (10-й класс и моложе). В отличие от олимпиад по прочим предметам, программисты обеих возрастных групп решают одни и те же задачи. На ранних этапах каждая возрастная группа оценивается отдельно. А вот на всероссийском и международном этапах все возрасты оценивают вместе, и чемпионом может стать любой. Не смущайтесь своих юных лет. Чуете в себе силы? Так смело врубайтесь в скопище бойцов!

Определение победителей и призеров

Обычно «олимпийцам» предлагают решить несколько задач, и для каждой из них следует написать программу. Разумеется, что время на решение ограничено (обычно это от 3 до 5 часов на все задачи). По истечении этого времени судьи приступают к тестированию программ. Тесты приготовлены заранее, но неизвестны участникам. Каждую программу проверяют на нескольких тестах, а баллы начисляют за каждый успешно выполненный тест. Щедрость теста зависит от его сложности и сложности решаемой задачи. В конце концов, побеждает тот, кто наберет больше баллов.

Олимпиады программистов отличает ещё одна особенность: для проверки решений здесь применяют тестирующие программы. Они автоматически компилируют исходный текст, а затем несколько раз запускают исполняемый файл, передавая ему всякий раз условия очередного теста. Это ускоряет проверку работ и придает ей объективность. Тестирующие программы применяют на областном и последующих этапах.

Организация туров

Тур — это один соревновательный день. Школьный и районный этапы, где количество решаемых задач невелико, проводят в один тур, а последующее — в два тура. Обычно в одном туре предлагается от двух до четырех задач, после чего подводятся и оглашаются его итоги.

Перед началом соревнований организаторы знакомят участников с правилами проведения этапа. Каждому участнику дают компьютер, а для приема решений выделяют сетевую папку. На всероссийском и международном этапах проводят предварительный пробный тур, где участники, решая простенькую задачу, знакомятся с рабочими местами и тестирующей системой. Пробный тур не оценивают.

И вот соревновательный тур настал: билеты с условиями задач розданы, часы включены. В течение тура участник может задавать жюри вопросы для устранения неясностей в условиях задач. Вопросы задают только в письменной форме (другие участники их слышать не должны!) и формулируют так, чтобы ответом было «да» или «нет». Нельзя, например, спросить, в каких пределах находится число **N**, но можно выяснить, является ли это число положительным («да» или «нет»?).

По окончании тура, участники покидают рабочие места, и начинается официальное тестирование их решений. Решение задачи — это исходный файл на выбранном языке (обычно на Паскале или Си). Тесты запускают в присутствии участника с использованием упомянутой тестирующей системы. Успешное решение должно компилироваться и выдавать правильные решения на предлагаемые тесты. Если тест не проходит, соответствующие ему баллы не начисляются. Жюри не рассматривает исходный текст как таковой, его интересуют лишь правильность прохождения тестов. Участник вправе опротестовать результат

тестирования, если, по его мнению, на это есть основания (например, ошибка в тестовых данных). Протест подается в жюри в письменной форме. Жюри рассматривает все протесты и принимает своё решение. По окончании тестирования и рассмотрения протестов, жюри подводит официальные итоги, объявляя победителей и призеров олимпиады.

Олимпиадные задачи

Как выглядит олимпиадная задача? — пример найдете в приложении M. В чем особенности таких задач?

Прежде всего, отметим способ ввода и вывода данных. То и другое выполняется только через текстовые файлы, которым присваивают заранее оговоренные имена. Так, если для файла с решением оговорено имя «ABC.PAS», то входной файл будет называться «ABC.IN» а выходной — «ABC.OUT». На эти расширения имен рассчитаны упомянутые тестирующие системы.

Входные данные олимпиадных задач считаются корректными. Это избавляет участника от их проверки и позволяет сосредоточиться на решаемой проблеме.

Упомяну о некоторых ограничениях. В решениях задач нельзя использовать внешние библиотечные модули (фирменные библиотеки), каждое решение представляется одним исходным файлом. Существуют ограничения и по времени исполнения программы (например, 5 секунд). Это побуждает участника искать быстрые, эффективные алгоритмы.

Таковы особенности олимпиад по информатике, теперь обсудим подготовку к ним.

Подготовка к олимпиаде

Предметные олимпиады — это ступень на лестнице профессионального роста. Советы программисту-ремесленнику отчасти годятся и «спортсмену». И всё же в олимпиадном программировании есть свои особенности.

Олимпийцы — всего лишь школьники, им хватит базовых средств выбранного языка. Применительно к Паскалю, участнику вполне достаточно того, что изложено в этой книге. Но владеть этим надо твердо. Вас не должен смущать ввод-вывод через текстовые файлы. Орудовать сложными типами данных надо так же свободно, как вилкой и ложкой. То же самое относится к процедурам и функциям. И в основных алгоритмах надо разбираться. Но, поскольку библиотечные модули правилами олимпиад запрещены, модульные технологии вам здесь не пригодится.

Итак, владение основами языка — безусловное требование к соискателям олимпийских лавров. Но этих знаний хватит лишь на первых порах — на школьном и районном этапах. Плюс природная смекалка. На последующих

уровнях смекалки уже мало, — нужна целенаправленная подготовка. В чем она состоит? Изучайте основные алгоритмы (в этом помогут книги, предложенные в библиографическом списке). Потренируйтесь в решении олимпиадных задач, — их полно в Сети, ссылки на некоторые ресурсы вы найдете в табл. 13.

Табл. 13 – Некоторые ресурсы Сети по олимпиадному программированию

Ресурс	Ссылка
Алгоритмы	http://algolist.manual.ru
Алгоритмы	http://bestalgorithm.ru
Олимпиады по программированию	http://www.olympiads.ru
Олимпиады по информатике (редактор – Андрей Станкевич)	http://neerc.ifmo.ru/school
Разбор олимпиадных задач (редактор – Михаил Густокашин)	http://g6prog.narod.ru
Олимпиады по информатике (сайт мытищинской школы программистов)	http://www.informatics.ru
Уральские олимпиады по программированию	http://contest.ur.ru
Спортивное программирование (соревнования «онлайн» с дистанционным тестированием)	http://acm.timus.ru
Командные олимпиады	http://de.ifmo.ru/cyber-net

Олимпиадные хитрости

Теперь о том, как биться на олимпиаде. На бой выходите со свежей головой, что называется в здравом уме и крепкой памяти. Ударные занятия перед олимпиадой только навредят, — лучше на пару дней забудьте о Паскале, погуляйте на воздухе и хорошенько отоспитесь.

Явитесь на олимпиаду без опоздания. Не тревожьтесь, ведь коленки дрожат не только у вас. Волнение — это нормальная реакция на опасность, оно мобилизует организм, но всё хорошо в меру.

Внимательно выслушайте предполетный инструктаж, в котором организаторы объяснят правила олимпиады. Получив условия задач, обязательно прочитайте их

все и оцените сложность. Не хватайтесь за первую попавшуюся задачу, начинайте с самой легкой на ваш взгляд. Действуйте по принципу «лучше синица в руке, чем журавль в небе». Одолев легкую задачу, беритесь за самую простую из оставшихся. Так вы наберете часть баллов, и настроитесь на решение трудных задач.

А если всё решить невмоготу? Тогда примените разрешенную правилами хитрость: выдайте хотя бы одно или два частных решения. Что это такое? Это решение, удовлетворяющее одному из тестов. Предположим, что по условию задачи вы должны вычислить факториал числа. Но вы напрочь забыли о факториале всё, кроме того, что факториалы для нуля и единицы равны единице. Тогда выдавайте единицу как решение для всех входных данных. Составители тестов наверняка проверят эти точки, и тогда ваша программа пройдет, по крайней мере, пару тестов.

Олимпиада быстротечна, и время надо беречь. А потому не пишите комментариев, не измышляйте слишком длинных имен. Но основными правилами оформления программ не пренебрегайте (логическими отступами, например), — они избавят вас от путаницы в собственном творении.

Ваше время сэкономит шаблон будущих программ. Вот пример такого шаблона, его можно напечатать в начале тура и сохранить в отдельном файле. И тогда не придется повторно набивать одни и те же операторы в разных задачах.

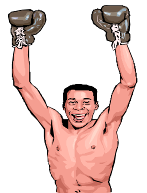
```
const CTask = 'Name';
      CIn = CTask+'.in';
      COut = CTask+'.out';
procedure ReadData;
var F: text;
begin
  Assign(F, CIn); Reset(F);
  Close(F);
end;
procedure WriteData;
var F: text;
begin
  Assign(F, COut); Rewrite(F);
  Close(F);
end;
```

```
begin
  ReadData;
  WriteData;
end.
```

Взяв за основу шаблон, и определив константу **CTask** для имен файлов, вы получите отчасти готовую программу.

Организаторы гарантируют правильность входных данных в своих тестах. Если в условии сказано, что число лежит в пределах от 1 до 10, то так оно и будет. Но это не значит, что вы введете его таким, — ошибка может затаиться в процедуре ввода. Она сведет на нет последующие правильные действия, и вы не получите нужный ответ. Так проверяйте ввод данных (отладчиком или выводом на экран). Разумеется, что для проверки решения вам придется выдумать свои тесты, поскольку тесты жюри участникам неизвестны.

Когда пыль уляжется…



И вот тестирование завершено, и результаты оглашены. Где мы с вами? Впереди на белом коне? Поздравляю! Искупайтесь в лучах славы, и, обсохнув на лаврах, начинайте готовиться к следующему этапу.

А если неудача? Каждый переживает ее по-своему. Кто-то даст волю чувствам и размажет слезы по щекам. Но вы не из тех, — я знаю. Поражение закаляет упорных. Внимательно прослушайте разбор задач, — его устраивают по окончании туров. Разберитесь в своих ошибках, решите задачи дома. А может, вам просто не хватило времени? — не беда, ведь и среди великих встречались тугодумы. Не опускайте рук, работайте, и обязательно добьетесь успехов. В конце концов, олимпиада — это всего лишь ступенька, которую можно... перепрыгнуть. Всё ещё впереди!

Приложение А
Установка и настройка IDE Borland Pascal

Рассмотрим установку и настройку интегрированной среды разработки Borland Pascal. Она создавалась для операционной системы MS-DOS, а такие приложения, исполняемые под Windows, своенравны, и требуют особого подхода.

Далее мы рассмотрим:

- историю создания IDE Borland Pascal и её состав;
- установку IDE из официального дистрибутива;
- устранение ошибки в модуле CRT;
- русификацию справочной системы;
- организацию рабочей папки;
- создание и настройку ярлыка для запуска IDE;
- предварительную настройку IDE;
- русификацию консольного окна;
- школьный пакет Turbo Pascal School Pak.

Полагаю, что ваш компьютер оснащен одной из операционных систем семейства Windows. Разным системам свойственны небольшие отличия в настройке IDE, они будут отмечены мною по ходу изложения.

История IDE Borland Pascal, состав дистрибутива

Седьмая версия IDE Borland Pascal — самая совершенная версия этого продукта. Её появление в начале 90-х годов совпало с расцветом операционной системы Windows 3.1 — первой коммерческой системы этого рода. Фирма Borland, шагая в ногу со временем, предусмотрела в своем товаре компилятор для разработки программ под Windows, ставший прообразом Delphi. В этом же пакете Borland предложила два компилятора для MS-DOS: один, более мощный, — для современных компьютеров, а другой — для компьютеров на слабеньких процессорах Intel 80x86. Прошли годы, и — странное дело! — компилятор под Windows 3.1 давно забыт, а для MS-DOS — всё ещё в ходу!

Несколько позже, после седьмой версии IDE, было выпущено её обновление — версия 7.1. Предпочтительней пользоваться обновленной версией. Различить их можно по времени создания файлов, которое составляет соответственно «07:00» и «07:01». После установки версии 7.0, обновление IDE до версии 7.1 выполняется заменой части файлов.

Нельзя умолчать о проблеме, проявившейся на мощных процессорах Pentium. Высокая скорость этих чипов выявила изъян в библиотечном модуле CRT, что

порождало беспричинные аварийные сообщения при запуске программ: «Runtime Error 200» — ошибка деления на ноль. Фирма Borland эту проблему не устранила, но её решили почитатели Паскаля. В Интернете можно найти бесплатные заплатки, исправляющие модуль CRT.

И напоследок о приятной мелочи. В том же Интернете есть перевод на русский язык встроенной в IDE справочной системы, рекомендую скачать русскую справку и заменить ею фирменный англоязычный файл.

Итак, для полноценной установки IDE Borland Pascal вам потребуются:

- дистрибутив Borland Pascal 7.0;
- обновление дистрибутива до версии Borland Pascal 7.1 (содержит в числе прочих файлы «BP7BIN.ZIP» и «BP7ETC.ZIP»):
- заплатка «T7TPLFIX.EXE», исправляющая библиотечный модуль CRT;
- файл справки на русском языке.

Установка IDE Borland Pascal

В ходе установки с официального дистрибутива предстоит сделать пять шагов, а именно:

1. копирование файлов дистрибутива;
2. установку IDE версии 7.0;
3. обновление IDE до версии 7.1;
4. исправление библиотечных файлов;
5. русификацию справочной системы.

Первый шаг – копирование файлов

Скопируйте все файлы дистрибутива в одну папку на жестком диске. Это предохранит оригинал от случайного повреждения и ускорит процесс установки. Предположим, что вы поместили все файлы дистрибутива в папку «D:\Distrib\Langs\Bpascal.700», — в дальнейшем я буду ссылаться именно на нее.

Второй шаг – установка версии 7.0

Запустите программу установки «INSTALL.EXE», она находится в папке с дистрибутивом. Явится окно с предварительной информацией о предстоящей установке. Для продолжения нажмите клавишу *Enter*, и перед вами появится следующее окно (рис. 155).

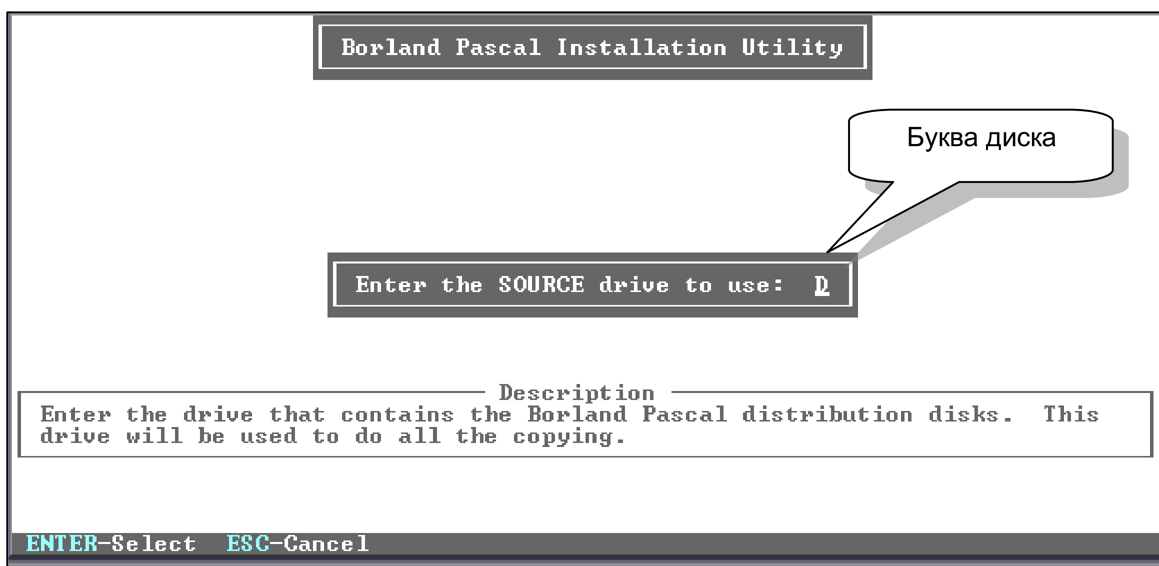


Рис. 155 – Указание буквы диска с дистрибутивом

Установщик просит ввести букву диска, на котором размещен дистрибутив (диск D). Разумеется, что нас эта буква устраивает, жмем *Enter* и переходим к следующему окну (рис. 156).

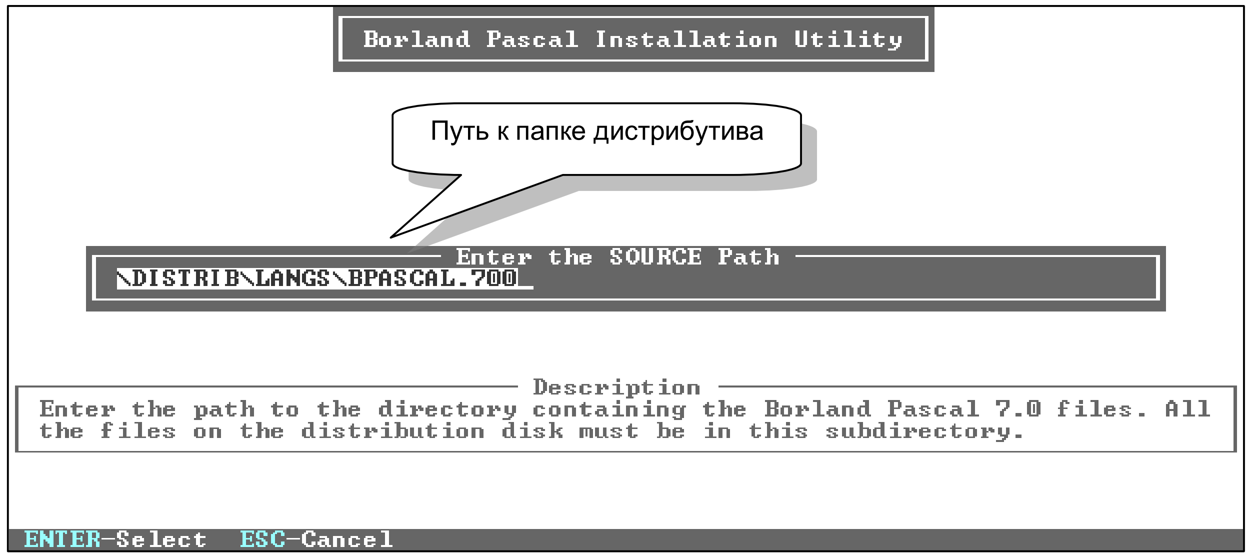


Рис. 156 – Указание папки, содержащей дистрибутив

Теперь надо указать папку с дистрибутивом. Программа предлагает ту папку, из которой была запущена. Нас это тоже устраивает, опять жмем *Enter* для перехода в следующее окно (рис. 157).

```
Borland Pascal Installation Utility

Borland Pascal directory   : D:\BP
Wind
Prog  c:\Lang\BP
Host                                  ]
Target Platforms           : [ DOS DPMI Windows ]

Product Component Options...
Example Options...
Directory Options...

Start Installation

Description
This drive/directory combination acts as the base for all other paths used
by the install program (with the exception of the Windows and BWCC
directories).

ESC-Cancel
```

Рис. 157 – Указание целевой папки (в неё будет установлен Borland Pascal)

Здесь предложено восемь пунктов для настройки опций предстоящей установки, девятый же пункт — «Start Installation» — запускает установку. Для перемещения между пунктами жмите клавиши со стрелками, для входа в пункт меню — клавишу *Enter*, а для выхода — *Esc*. Нам интересны первые пять пунктов, рассмотрим их подробней.

В пункте «Borland Pascal Directory» задается целевая папка, в которую будет установлена IDE. По умолчанию предложена папка «D:\BP», но я захотел установить IDE в папку «C:\Lang\BP». А потому, выбрав этот пункт и нажав *Enter*, впечатал нужный мне путь и снова нажал *Enter* (рис. 157). В ходе установки указанные мною папки «Lang» и «BP» будут созданы автоматически.

Затем, перейдя к пункту «Program Manager Group» и нажав *Enter*, я отменил создание группы в главном меню Windows (выбрал «Don't Create»).

В следующих двух пунктах — «Host Platforms» и «Target Platforms» — надо указать две платформы: одну — для разработки, другую — для исполнения разработанных программ. О чем тут речь?

Напомню о трех IDE, предложенных фирмой Borland: одна из них — для слабеньких компьютеров на базе 80x86, другая — для мощных компьютеров и третья — для Windows 3.1. Эти три IDE представлены программами «TURBO.EXE», «BP.EXE» и «BPW.EXE» соответственно. Но нас интересует только IDE «BP.EXE». Ставить все три неразумно, к чему засорять винчестер? Итак, войдя в пункты меню «Host Platforms» и «Target Platforms», выберем вариант «Install» для «BP.EXE» и «Don't Install» для всех остальных (рис. 158).

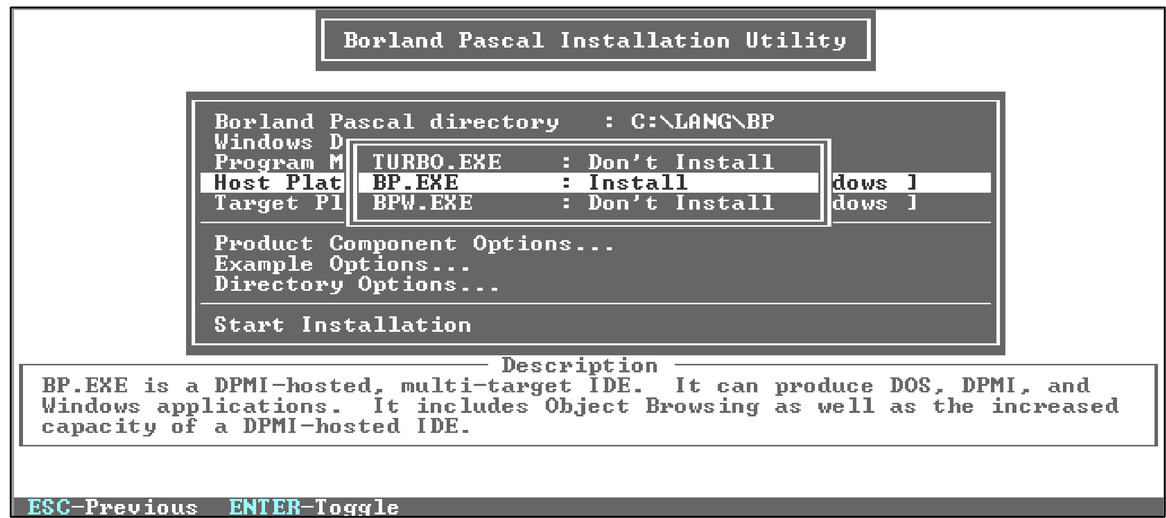


Рис. 158 – Выбор устанавливаемых вариантов IDE

Теперь, после настройки опций, запускаем процедуру установки. В этот момент окно программы будет таким (рис. 159).

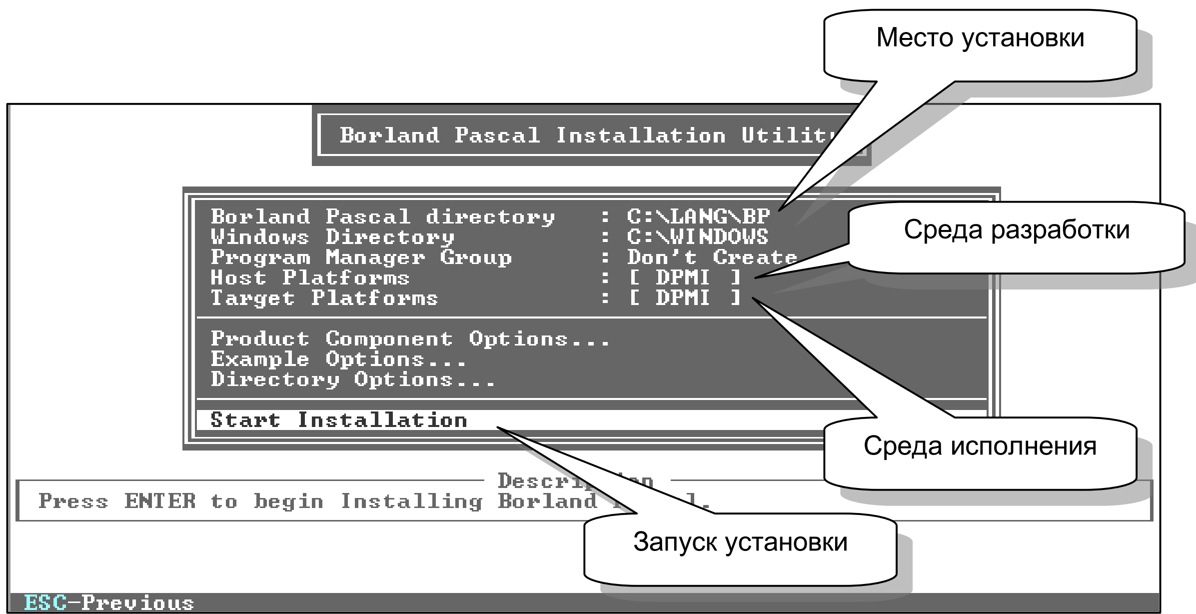


Рис. 159 – Значения опций перед запуском установки

Выбираем пункт «Start Installation» и жмем *Enter*. Программа установки создаст предопределенную структуру папок по указанному нами пути (рис. 160) и распакует туда файлы IDE.

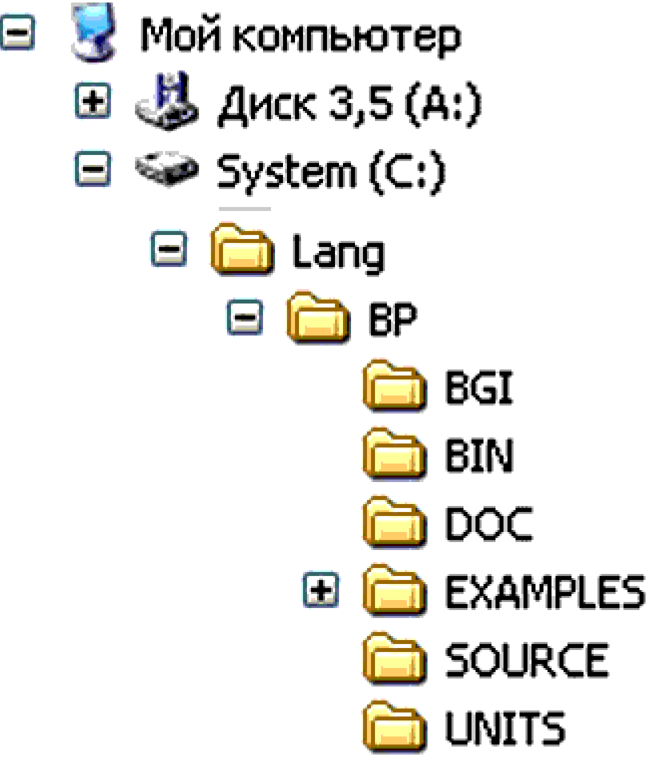


Рис. 160 – Структура папок после установки Borland Pascal

Содержание вложенных папок представлено в табл. 14.

Табл. 14 - Содержание папок Borland Pascal

Папка	Содержание
BGI	Драйверы для графических программ под MS-DOS
BIN	Исполняемые файлы IDE Borland Pascal, основные библиотеки и файлы справок
DOC	Файлы документации на английском языке
EXAMPLES	Примеры программ
SOURCE	Исходные файлы библиотеки Turbo Vision
UNITS	Библиотечные модули

Основные файлы размещены в папке «BIN», вот они.

```
BP.EXE          - исполняемый файл IDE Borland Pascal
DPMI16BI.OVL    - вспомогательный файл IDE
RTM.EXE         - вспомогательный файл IDE
TURBO.TPL       - системная библиотека
TURBO.TPH       - файл основной справочной системы
TVISION.TPH     - файл справочной системы Turbo Vision
```

На этом установка IDE Borland Pascal версии 7.0 завершена, и с нею уже можно работать. Но лучше сделать и третий шаг — обновление IDE до версии 7.1.

Третий шаг – обновление до версии 7.1

Запустите входящий в состав дистрибутива обновления пакетный файл «INSTALL.BAT». При запуске ему необходимо указать путь к папке, где уже установлена версия Borland Pascal 7.0. В моем случае команда запуска такова.

```
install c:\Lang\BP
```

При обновлении будут распакованы файлы из архивов «BP7BIN.ZIP» и «BP7ETC.ZIP», они заменят собою файлы в созданных ранее папках версии 7.0. Можно поступить иначе, указав при запуске несуществующую папку, например, так:

```
install c:\Lang\Temp
```

Тогда файлы распакуются в папку «Temp» с сохранением стандартной файловой структуры IDE (рис. 160). Затем скопируйте их поверх файлов старой версии.

Наконец, можно не запускать установщик, а распаковать архивы «BP7BIN.ZIP» и «BP7ETC.ZIP» вручную подходящим для этого распаковщиком и полученные файлы скопировать в надлежащие папки IDE. Так или иначе, обновление состоит в замене части файлов установленной ранее версии 7.0 файлами версии 7.1. Напомню, что отличить новые файлы можно по времени их создания, которое составляет «07:01».

Четвертый шаг – исправление библиотечных файлов

Скачайте в Интернете заплатку «T7TPLFIX.EXE» и поместить её в подкаталог «BIN», где расположены библиотеки «TURBO.TPL» и «TPP.TPL». Запустив программу-заплатку, вы увидите список предлагаемых действий. Для исправления библиотек нажмите клавишу *P*, а затем завершите программу

клавишей *Q*. После этого в каталоге «BIN» появятся два файла с расширениями «OLD» — это старые версии библиотек. Исправленные библиотеки сохранят прежние имена «TURBO.TPL» и «TPP.TPL».

Пятый шаг – русификация справочной системы

Русификация справки — дело добровольное. Если вы не очень сильны в английском, русская справка не помешает. Предположим, что русифицированный файл справочной системы вы уже скачали и сохранили где-нибудь под именем «Turbo_rus.tph». Войдите в папку «BIN», найдите там файл «Turbo.tph» и назовите его как-то иначе, например «Turbo_eng.tph», сохранив его, таким образом, на всякий случай. Затем скопируйте в папку «BIN» файл с русской справкой и назовите стандартным именем «Turbo.tph».

На этом установка IDE завершена. Распакованную и настроенную IDE сохраните на компакт-диске или в архивном файле — на случай потери части файлов. Такую «законсервированную» IDE можно затем переносить на другие компьютеры, не повторяя всех шагов установки.

Организация рабочей папки

Что такое рабочая папка? Это папка, где мы будем хранить свои программы, а их будет немало. Негоже разбрасывать свои файлы, где попало: ведь найти их будет трудно, а удалить по неосторожности — легко. Создайте для хранения программ папку в подходящем месте, например, по следующему пути:

C:\User\Pascal

В названиях папок я употребил только латинские буквы, причем длина каждого имени не превышает восьми букв. При работе с IDE Borland Pascal рекомендую поступать именно так, поскольку среда сделана для MS-DOS, «не понимающей» русских имен.

Во избежание проблем придерживайтесь и принятого в MS-DOS правила «8.3» (восемь точка три). Это значит, что имя файла должно содержать не более восьми символов, а расширение — не более трех. Пробелы и русские буквы в именах недопустимы.

Создание и настройка ярлыка

Для вызова IDE Borland Pascal запускается файл «BP.EXE». Но такой простейший способ нам не вполне годится. Почему? — сейчас поймете. Мы поступим иначе, — создадим на рабочем столе ярлык, запускающий IDE. Ярлык — это специальный файл, активизирующий другие файлы, например, программы. Через ярлык можно не только запустить программу, но и указать параметры для неё, достигая тем самым дополнительного эффекта. В нашем случае при запуске «BP.EXE» надо достичь следующих целей:

- указать текущую рабочую папку для программ;
- включить полноэкранный режим консольного окна;
- запустить русификатор консольного окна.

Рассмотрим эти пункты подробней.

Текущая папка — это папка, в которую пользователь переключился в данный момент времени. Перемещаясь по папкам своего компьютера посредством программы Explorer, вы, сами того не подозревая, переключаете текущую папку. Если запустить программу «BP.EXE» из того места, где она расположена (назовем эту папку стартовой), мы сделаем эту папку текущей, — в моем случае это папка «D:\Lang\BP\BIN». Но при работе в IDE мне нужна другая текущая папка — «C:\User\Pascal», именно там я буду хранить свои программы. Такого переключения папок можно добиться настройкой ярлыка.

Чего ещё хотим от ярлыка? — включить полноэкранный режим консольного окна. В этом режиме экран выглядит так, как в системе MS-DOS, — с крупными, «увесистыми» буквами. В любой момент окно консоли можно переключить в полноэкранный режим и обратно нажатием комбинации клавиш *Alt+Enter*. Впрочем, в некоторых версиях Windows полноэкранный режим не поддерживается. Тем не менее, я покажу соответствующую настройку ярлыка.

И, наконец, третья польза от ярлыка состоит в русификации консольного окна. Эту возможность обсудим позже, когда речь пойдет о русификации.

Приступим к изготовлению ярлыка. Откройте папку «BIN» и найдите там файл «BP.EXE». Нажав клавишу *ALT* и удерживая её, «схватите» левой кнопкой мыши файл «BP.EXE» и перетащите его на рабочий стол, — всё, ярлык готов! Система сама назначит ему имя, которое лучше изменить на более краткое, например «Pascal»: щелкните мышкой по названию ярлыка (или нажмите клавишу *F2*), напечатайте подходящее имя, а затем нажмите *Enter*.

Теперь настроим ярлык. Щелкните по нему правой кнопкой мыши и во всплывающем меню выберите пункт «Свойства», — откроется окно настройки свойств ярлыка (рис. 161).

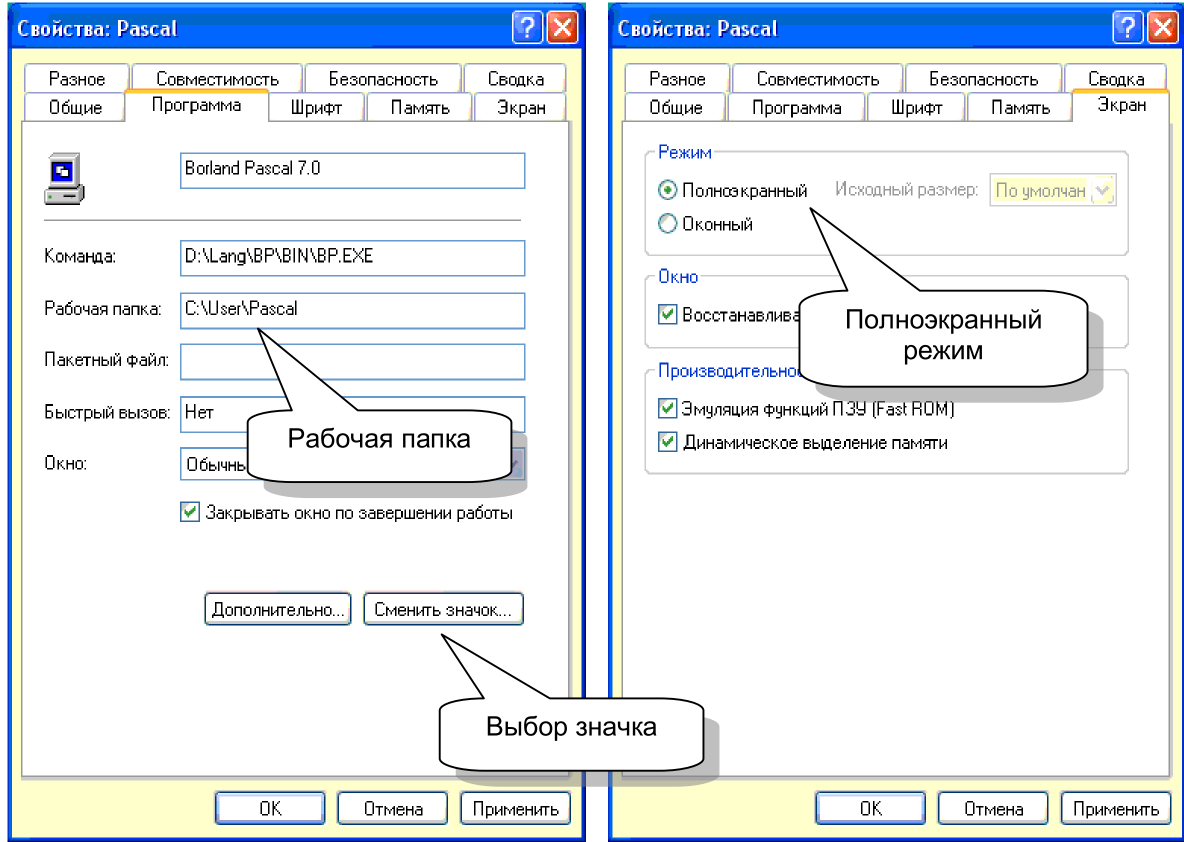


Рис. 161 – Настройка ярлыка в Windows XP

Окно содержит несколько вкладок, но нас интересуют лишь две из них — «Программа» и «Экран». На вкладке «Программа» укажите путь к рабочей папке; здесь же можно сменить значок ярлыка (если вам не нравится предложенный по умолчанию). На вкладке «Экран» можно установить режим «Полноэкранный». Но я рекомендую сделать это позже, после пробного запуска IDE, поскольку полноэкранный режим не всегда возможен. В завершение нажмите кнопку *OK*.

Пробный запуск IDE

Перед первым запуском IDE войдите в стартовую папку «BIN» и поищите там файлы с именами «BP.TP» и «BP.DSK» — они хранят настройки IDE. При первом запуске эти файлы будут лишь мешать нам. Если «злоумышленники» будут обнаружены, удалите их. Готово? Тогда смело «дергайте» за ярлык. Если всё сделано правильно, появится следующая картинка (рис. 162).

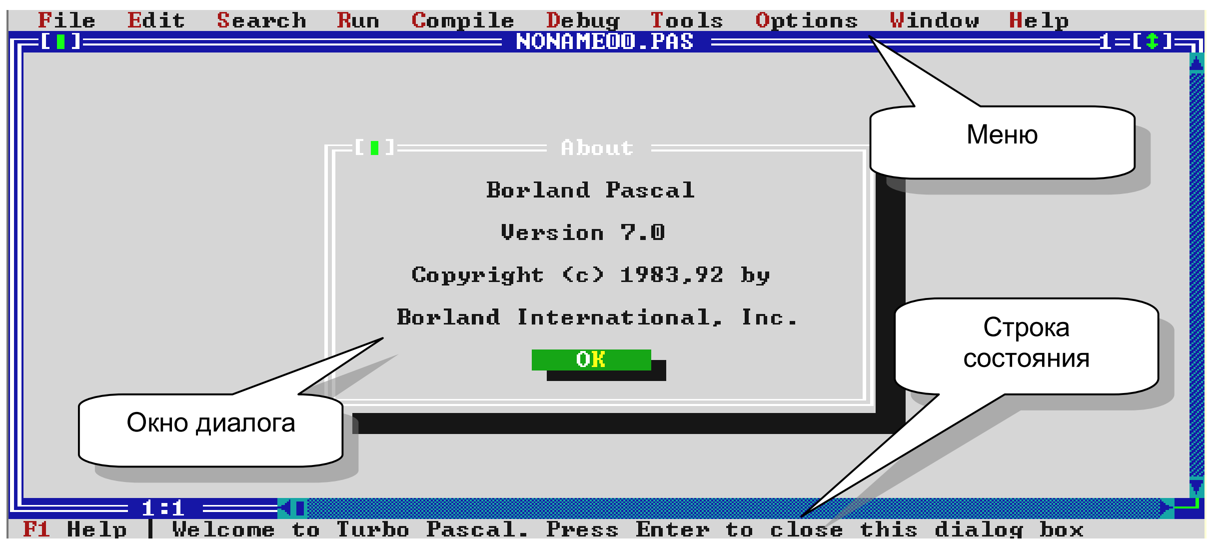


Рис. 162 – Первый запуск IDE Borland Pascal

Вы видите экран IDE Borland Pascal. В центре — диалоговое окно, рапортующее о версии IDE, закройте его нажатием *Enter* либо кнопкой «OK». Верхнюю строку занимает меню IDE, а нижнюю — строка состояния с полезными подсказками. Область в центре — это место для редактора, здесь будут располагаться окна с текстами программ.

Теперь проверим переключение между оконным и полноэкранным режимами. Нажмите пару раз комбинацию *Alt+Enter*. Если переключение срабатывает без видимых проблем, можно вернуться к настройке ярлыка и установить там полноэкранный режим (рис. 161).

Теперь выйдем из IDE. Попытка сделать это щелчком по крестику «не понравится» Windows, ответом будет «ругательное» сообщение (рис. 163).

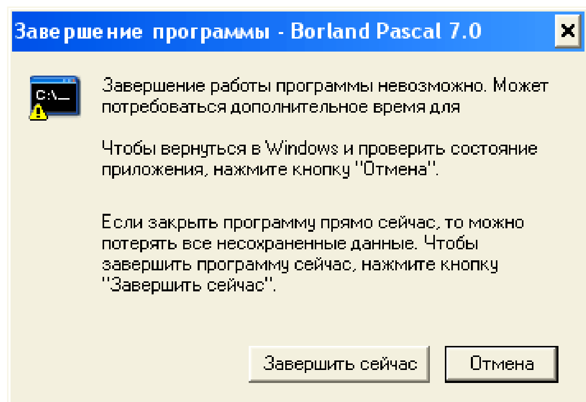


Рис. 163 – Сообщение о неправильном закрытии окна MS-DOS

Не сокрушайтесь, ведь есть два правильных способа выхода из IDE: нажатие комбинации *Alt+X*, либо пункт меню *File* → *Exit*.

Предварительная настройка IDE

IDE Borland Pascal содержит ряд опций, которые можно настроить по своей прихоти. Позднее вы покопаетесь в них, а сейчас организуем место хранения этих опций. Сделаем так, чтобы все последующие настройки IDE сохранялись в рабочей папке. Опции и состояние сеанса хранятся в трех служебных файлах: «BP.TP», «BP.DSK» и «BP.PSM». Сейчас надо решить лишь два вопроса: где разместить эти файлы и как это сделать.

Поиск своих служебных файлов IDE начинает с текущей папки. Не обнаружив их там, она обращается к стартовой папке, а если файлов нет и там, то устанавливает опции по умолчанию. Так, где же удобней хранить эти опции?

Лучше хранить их в рабочей папке вместе с текстами своих программ. Тогда опции можно настраивать по-своему в каждой рабочей папке, подгоняя под особенности данного проекта. Это удобно и тогда, когда одним компьютером пользуются несколько человек. Вдобавок при копировании проекта опции будут переноситься вместе с другими файлами. Вот такую предварительную настройку IDE мы сейчас и проделаем.

Запустите IDE и обратитесь к пункту меню *Options* → *Environment*→ *Preferences* (рис. 164). Меню активируется как мышкой, так и клавишей *F10*.

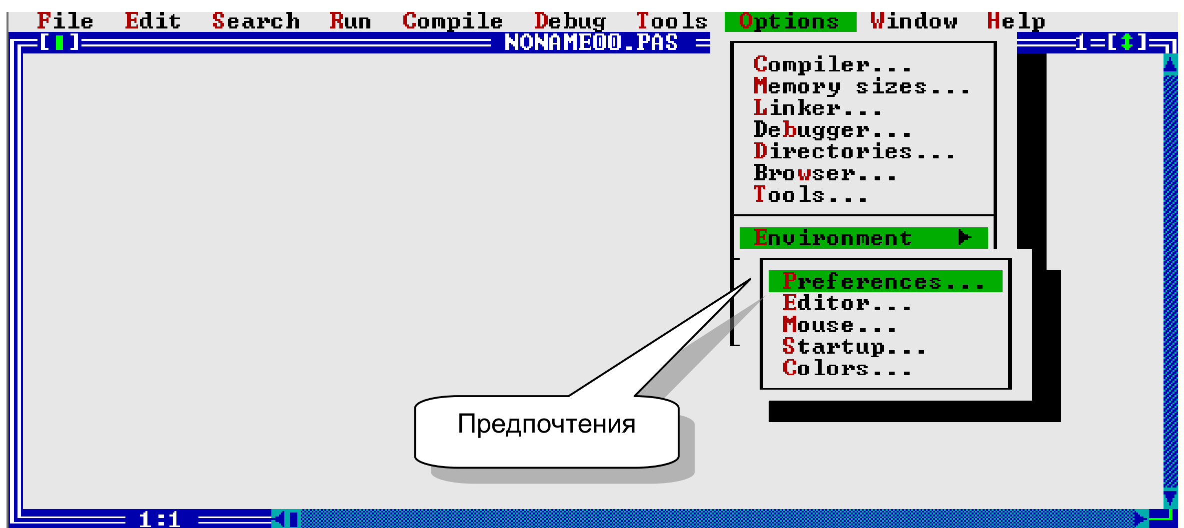


Рис. 164 – Выбор пункта меню ***Options* → *Environment* → *Preferences***

В окне «Preferences» (предпочтения) установите опции соответственно рис. 165, затем нажмите кнопку *OK*.

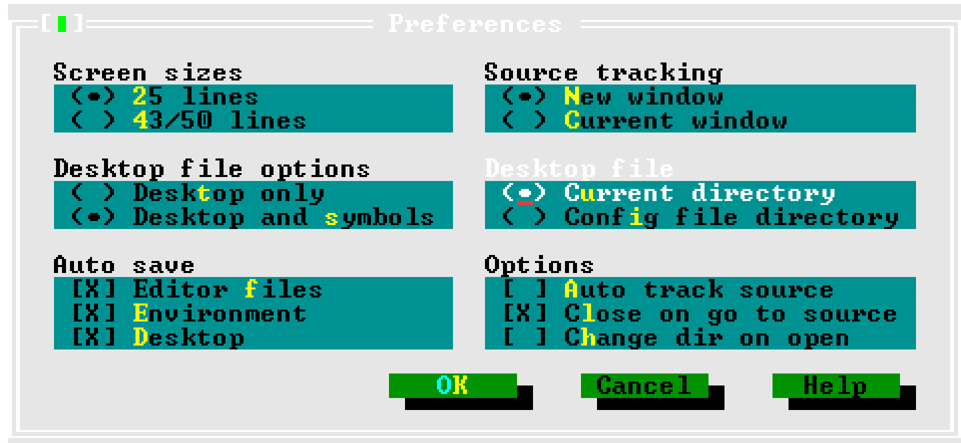


Рис. 165 – Настройка опций «Предпочтения»

Сохраните опции, обратившись к пункту *Options* → *Save as...* (рис. 166).

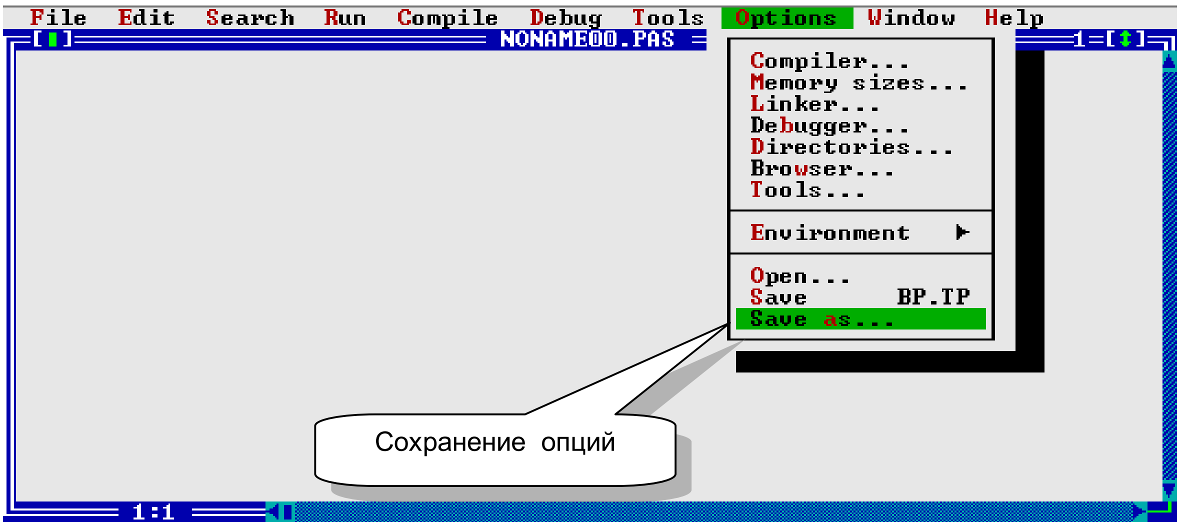


Рис. 166 – Выбор пункта меню *Options* → *Save as...*

В открывшемся окне «Save Options As» (рис. 167) введите имя сохраняемого файла «BP.TP» и нажмите *OK*. С этого момента все опции и состояние IDE будут автоматически сохраняться в рабочей папке, и восстанавливаться при следующем запуске IDE.

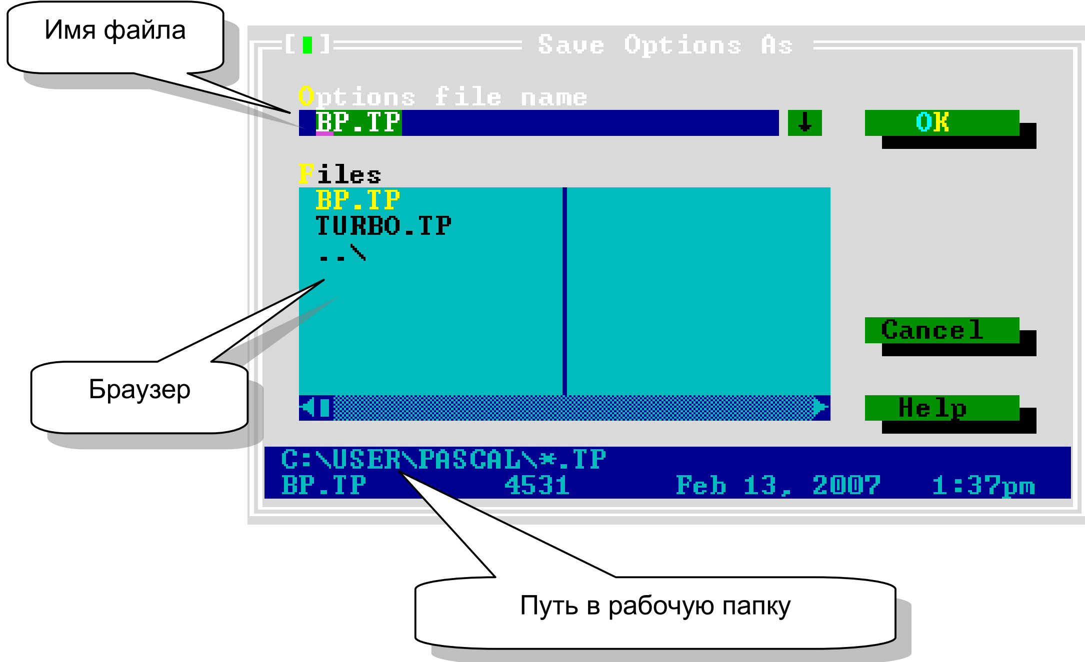


Рис. 167 – Сохранение опций в рабочей папке

Сохраняя файлы, следите за строчкой в нижней области окна. Там виден путь к текущей папке, в моем случае это «C:\User\Pascal». Если по каким-либо причинам текущая папка не совпадает с рабочей, перейдите в рабочую папку браузером «Files» в центре окна. Или напечатайте в строке ввода полный путь к файлу, например «C:\User\Pascal\BP.TP».

Русификация консольного окна

Проблемы с русским текстом в консольном окне обнаружат обладатели операционных систем Windows XP и более поздних. Сам по себе русский текст отображается верно, но не работает переключение ввода между русским и английским языками. Решением этой проблемы сейчас и займемся.

Любое препятствие можно преодолеть двояко: либо обойти его (умный в гору не пойдет!), либо устранить. Обойти проблему проще: не пользуйтесь русскими буквами — и баста! Так, если ваша программа вместо «Привет!» выведет на экран «Privet!» или «Hello!», никто не обидится. Но когда вам захочется пообщаться с компьютером на родном языке, препятствие придется устранять. Я расскажу о двух способах русификации.

Первый способ

Этот способ прост, но требует прав администратора (проверен на Windows XP).

1) В системной папке «C:\Windows\System32» (если система установлена в папку «C:\Windows») найдите и откройте редактором файл «Autoexec.nt», добавьте в конец файла следующую строку.

```
LH %SystemRoot%\system32\kb16.com ru
```

Сохраните этот файл.

2) Через главное меню откройте реестр Windows:

Пуск → Выполнить → Regedit

3) В левой половине окна распахните ветвь реестра (рис. 168):

HKEY_LOCAL_MACHINE → SYSTEM → CurrentControlSet → Control → Keyboard Layout → DosKeybCodes

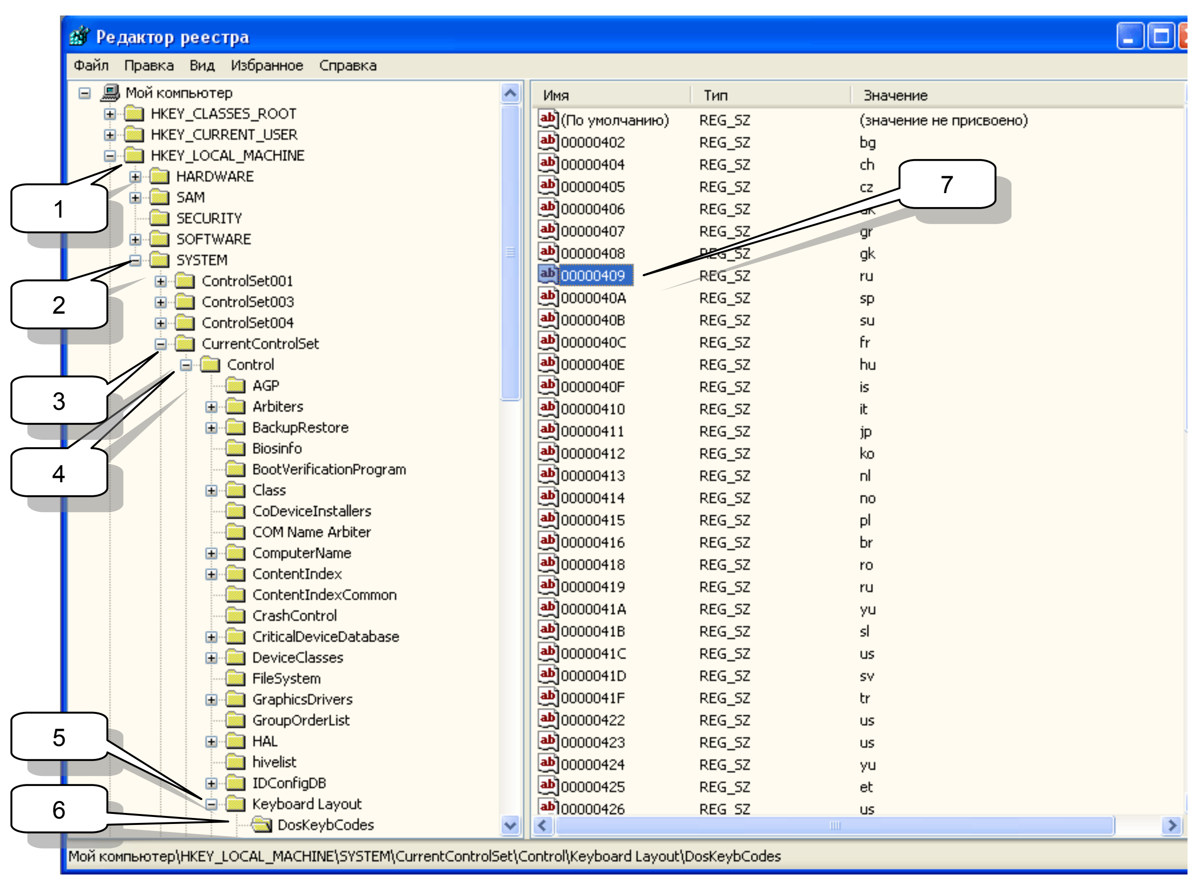


Рис. 168 - Ветвь реестра для настройки русификатора

4) Справа в колонке «Имя» найдите параметр «00000409» и дважды щелкните по нему. В открывшемся окне в поле «Значение» вместо «us» введите «ru» (две буквы без кавычек). Закройте редактор реестра. Теперь в консольном окне MS-DOS раскладка клавиатуры будет переключаться клавишами *Ctrl+Shift*.

Второй способ

Этот путь не требует прав администратора (проверен на Windows XP и Windows Vista).

Первое — скачайте в Интернете русификатор «RK.COM» или его аналог «RK-866.COM», а также драйвер «RNBOVDD.DLL». Дальше всё зависит от установленной операционной системы и ваших прав на компьютере. Если вы работаете в Windows XP и обладаете правами администратора, действуйте следующим образом.

Скопируйте упомянутые файлы в системную папку «C:\Windows\System32». В этой же папке разыщите файл «AUTOEXEC.NT», откройте его редактором и добавьте в конце следующую строку.

```
LH %SystemRoot%\system32\rk.com /R5
```

Или, если скачан файл «RK-866.COM», то соответственно

```
LH %SystemRoot%\system32\rk-866.com /R5
```

Сохраните файл «AUTOEXEC.NT» и откройте в этой же папке файл «CONFIG.NT». Проверьте наличие в нём следующих строк.

```
dos=high, umb
device=%SystemRoot%\system32\himem.sys
files=40
```

Убедитесь, что значение «FILES» составляет не мене 40, а иначе исправьте его и сохраните файл. На этом русификация завершена, переключение клавиатуры в консоли MS-DOS выполняется правой комбинацией *Ctrl+Shift*.

Теперь рассмотрим случаи, когда у вас нет прав администратора, или когда на компьютере установлена система Windows Vista. Здесь нужна дополнительная настройка ярлыка, и действовать надо так.

Создайте где-нибудь папку для хранения файлов русификатора, например, «D:\Lang\BP\Rus». Скопируйте в неё файл «RK.COM» (или «RK-866.COM»), а также два файла из системной папки «C:\Windows\System32»: файл конфигурации «CONFIG.NT» и файл автозапуска «AUTOEXEC.NT».

В конце вашей копии файла «AUTOEXEC.NT» добавьте строку

```
LH d:\Apps\Lang\BP\Rus\Rk.com /R5
или
LH d:\Apps\Lang\BP\Rus\Rk-866.com /R5
```

Теперь настройте запускающий IDE ярлык: щелкните по нему правой кнопкой мыши и выберите пункт «Свойства». На вкладке «Программа» щелкните по кнопке «Дополнительно» (рис. 169).

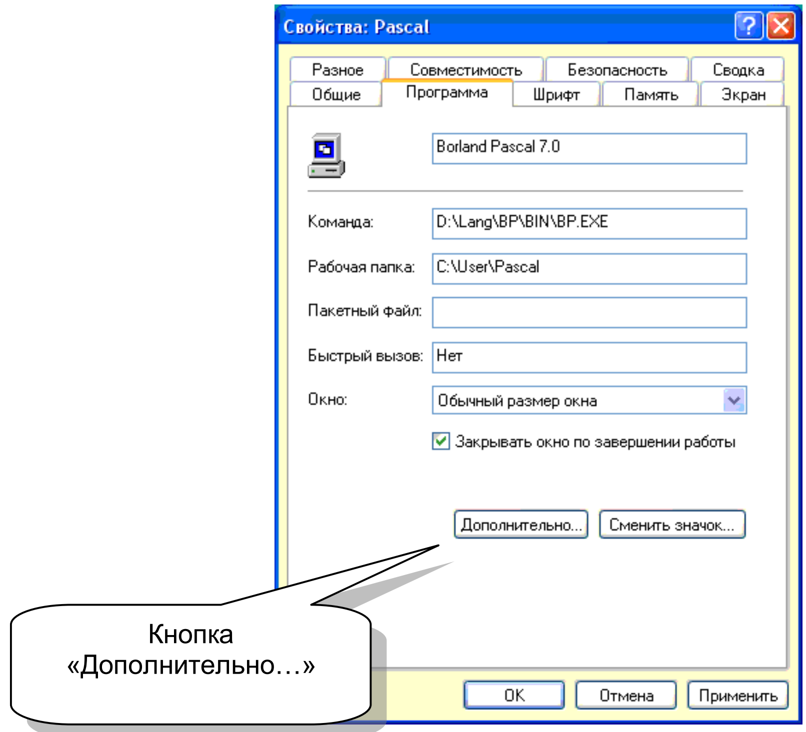


Рис. 169 – Настройка ярлыка для русификации консольного окна

Появится показанное ниже окно для ввода имен системных файлов «CONFIG.NT» и «AUTOEXEC.NT».

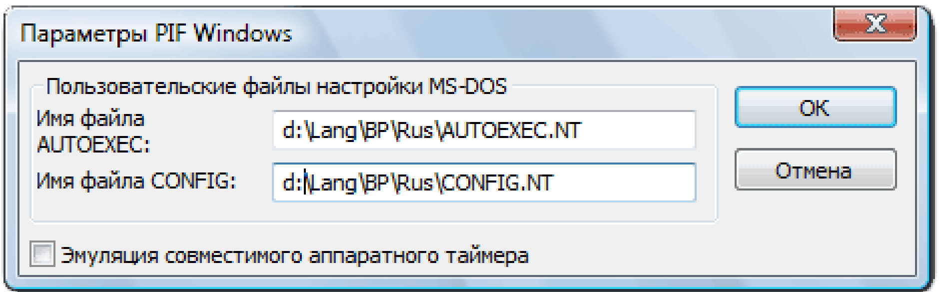


Рис. 170 – Указание пути к файлам настройки MS-DOS

Здесь, вместо файлов, предложенных по умолчанию, укажите файлы из вашей папки с русификатором (рис. 170), например:

```
D:\Lang\BP\Rus\Config.nt
D:\Lang\BP\Rus\Autoexec.nt
```

Кнопка *OK* завершит дело. К сожалению, этот вариант русификации действует только на данный ярлык. При запуске других программ MS-DOS русификатор не сработает, — для них надо создавать и настраивать свои ярлыки.

И последнее. При запуске IDE с русификатором в Windows XP иногда появляется сообщение о сбое (рис. 171).

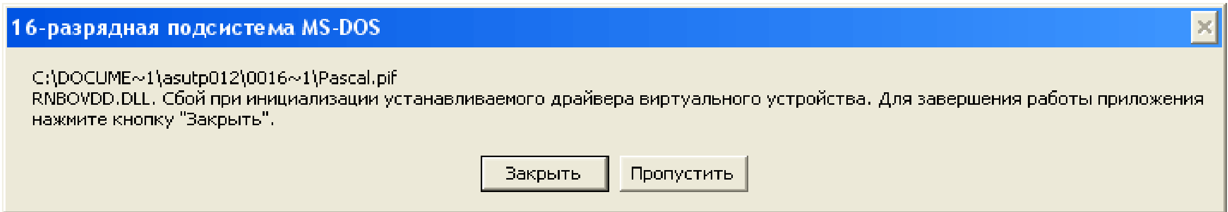


Рис. 171 – Сообщение о сбое

Нажмите здесь кнопку «Пропустить», пренебрегая этим сообщением.

Turbo Pascal School Pak

Итак, кропотливая работа по настройке IDE завершена. Но эту работу можно сделать проще и быстрее, если воспользоваться школьным пакетом «Turbo Pascal School Pak», который легко найти в Сети.

Чем хорош этот продукт? Вот главные его прелести:

- включает в себя предустановленную IDE Borland Pascal;
- весь интерфейс и справочная система – на русском;
- не требует русификации консольного окна;
- крайне прост в установке и работает в последних версиях Windows.

Turbo Pascal School Pak работает на так называемой виртуальной машине DOSBox (входит в состав пакета) — она эмулирует MS-DOS. В комплект входит и ряд дополнительных программ, включая Norton Commander.

Вот ещё одна особенность DOSBox: после щелчка в её окне мышь будет захвачена виртуальной машиной и не покинет это окно, пока вы не нажмете комбинацию *Alt+F11*.

Приложение Б
Консольная программа в среде Delphi

Программы, рассмотренные в этой книге, могут создаваться в среде Delphi в качестве консольных приложений. Здесь изложен порядок создания, настройки и русификации консольного приложения.

Создание пустого консольного приложения

При входе в среду программирования Borland Delphi автоматически создается пустое оконное приложение с именем Project1, – нам оно не нужно. Для создания консольного приложения поступайте так.

1) Выберите пункт меню *File → New → Other…* (рис. 172).

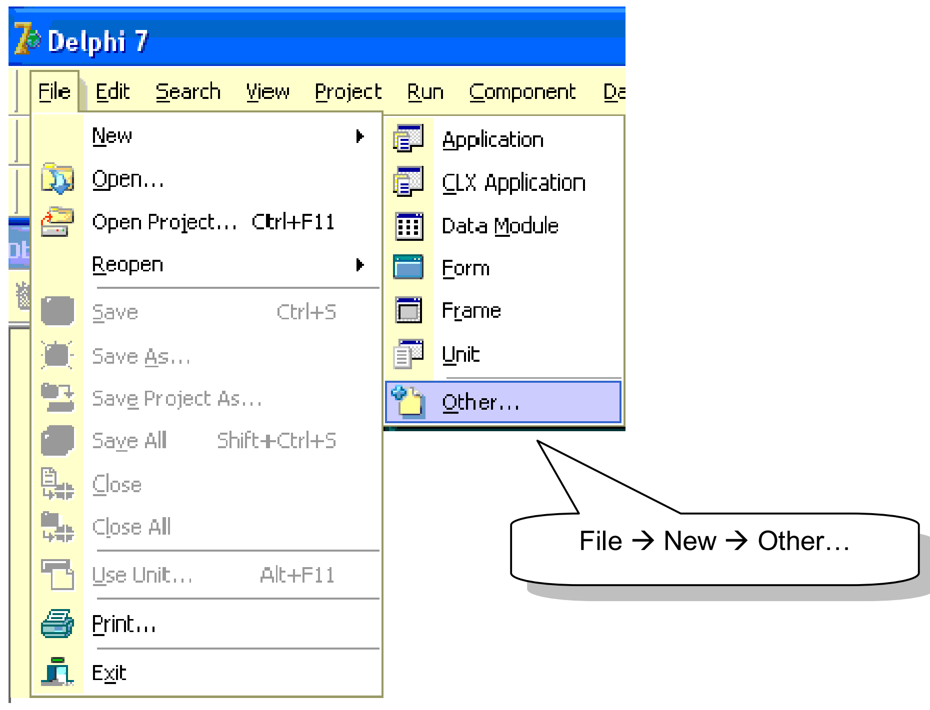


Рис. 172 – Пункт меню *File → New → Other…*

2) В открывшемся окне выберите вкладку «New», а в ней – тип приложения «Console Application», затем нажмите кнопку *OK* (рис. 173).

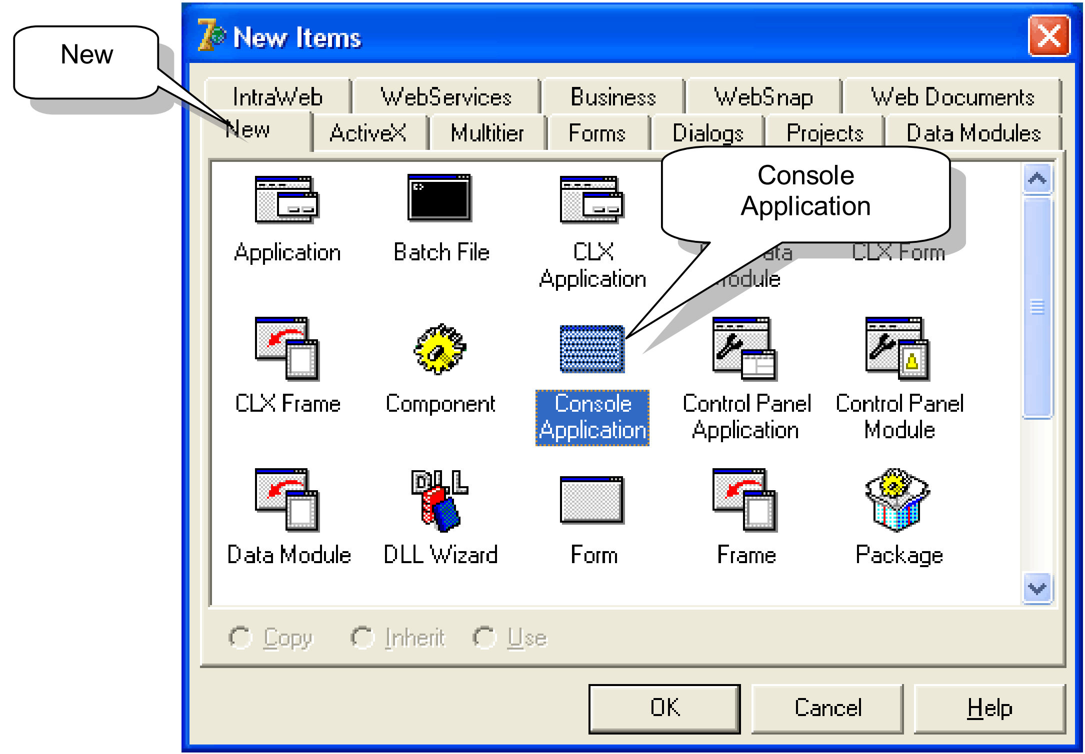


Рис. 173 – Окно выбора типа приложения

В результате на экране появится окно с заготовкой будущей программы.

```
program Project1;
{$APPTYPE CONSOLE}
uses  SysUtils;
begin
  { TODO User Console Main : Insert code here }
end.
```

В первой строке указано имя проекта **Project1**, затем директива **CONSOLE**, определяющая тип приложения, а в третьей строке – список подключаемых модулей **USES**. Комментарий, стоящий между ключевыми словами **BEGIN** и **END**, отмечает место, где будет располагаться главная программа, этот комментарий можно удалить.

Настройка и сохранение консольного приложения

Перед сохранением пустого консольного приложения настройте опции компилятора следующим образом:

1) Выберите пункт меню *Project* → *Options...* (рис. 174).

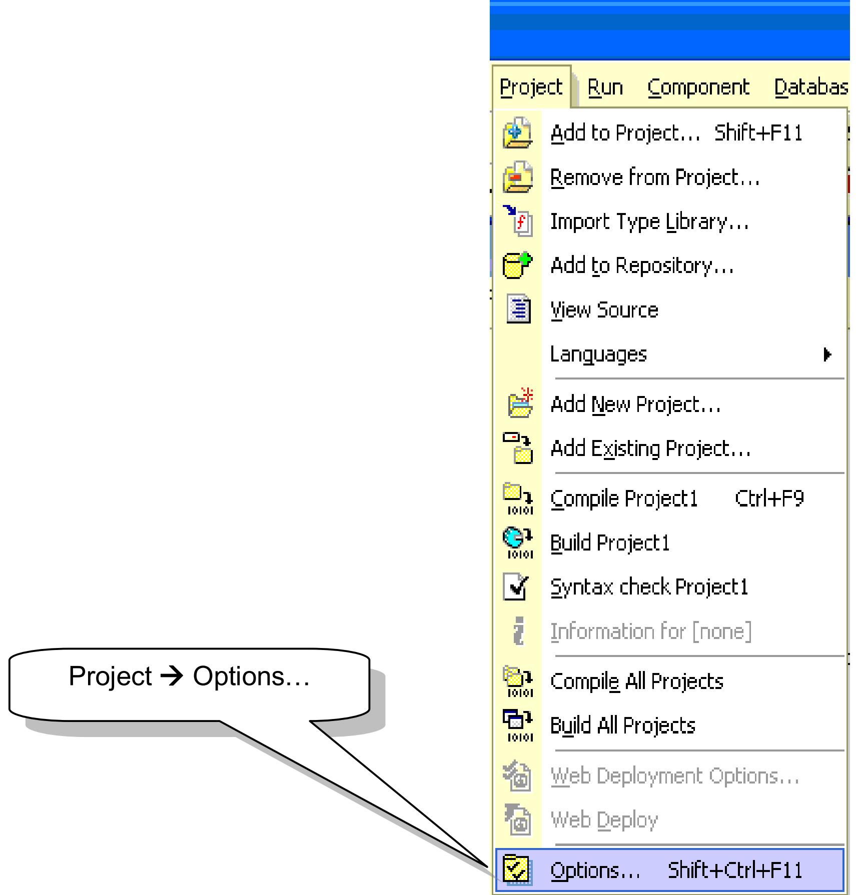


Рис. 174 – Выбор пункта меню *Project → Options…*

2) В открывшемся диалоге выберите вкладку «Compiler» и установите опции компилятора так, как показано на рис. 175. Эта настройка обеспечит совместимость с компилятором Borland Pascal. Перед нажатием кнопки *OK* установите флажок «Default», и тогда последующие проекты будут создаваться с этими же опциями.

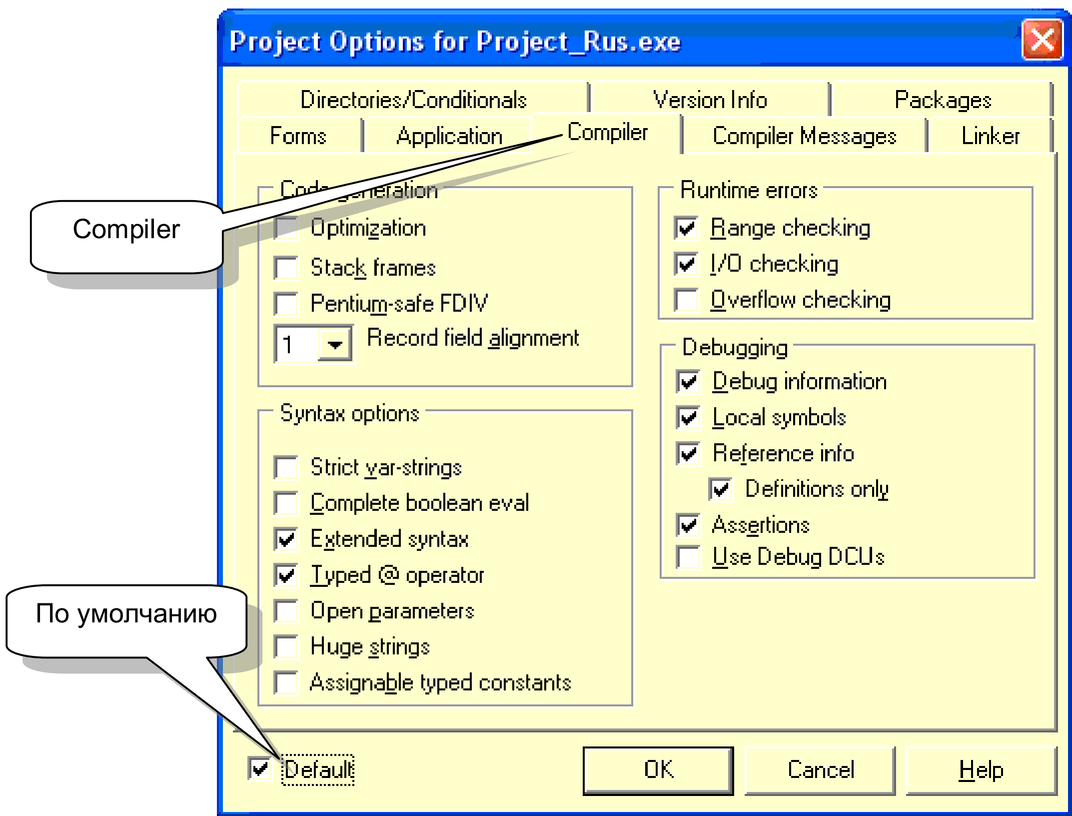


Рис. 175 – Окно настройки опций проекта, вкладка Compiler

3) Для сохранения приложения обратитесь к пункту меню *File → Save* или *File → Save Project As…* (рис. 176)

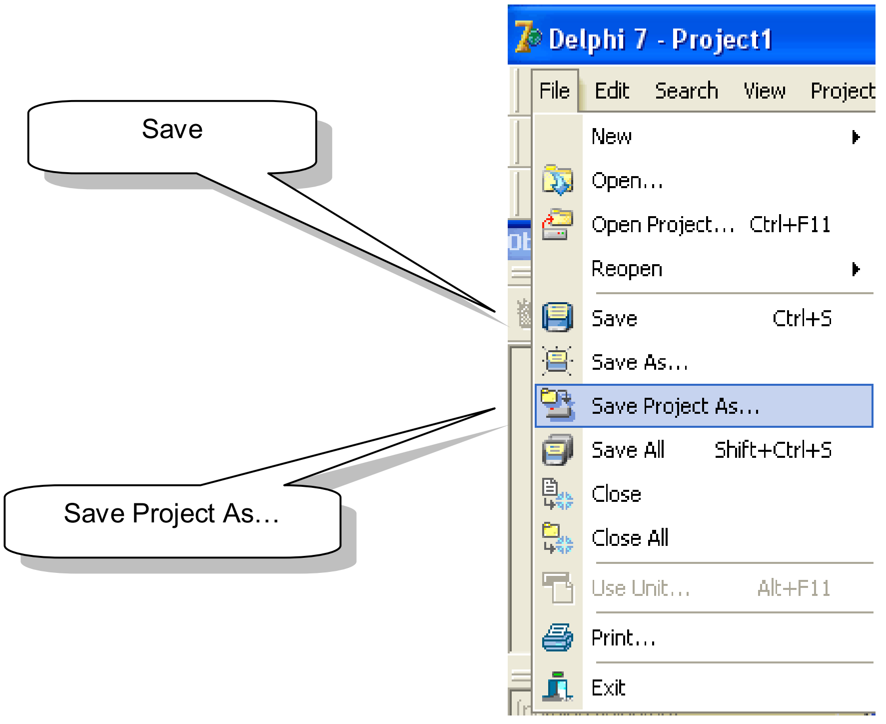


Рис. 176 – Выбор пункта меню для сохранения проекта

В открывшемся диалоге найдите нужную рабочую папку и укажите имя сохраняемого файла. Файлу проекта автоматически назначается расширение DPR (а не PAS, как в Borland Pascal).

Внимание! При сохранении файла проекту автоматически назначается имя этого файла (имя проекта указано после ключевого слова **Program**). Здесь вступают в силу ограничения на идентификаторы, действующие в Паскале. Имя файла должно начинаться с латинской буквы, состоять из латинских букв, цифр и знаков подчеркивания. Например, после сохранения проекта под именем My_Prj_1 окно с заготовкой программы станет таким.

```
program My_Prj_1;
{$APPTYPE CONSOLE}
uses  SysUtils;
begin
  { TODO User Console Main : Insert code here }
end.
```

С этого момента приступайте к вводу своей программы. Компиляция и запуск консольного приложения выполняются клавишей *F9*.

Русификация консольного приложения

Консольные приложения Delphi работают почти так же, как созданные в Borland (Free) Pascal. Но есть проблема с выводом русского текста на экран (именно на экран, а не в тестовый файл!). Запустив следующую программу, вместо русского приветствия вы увидите «абракадабру».

```
program My_Prj_1;
{$APPTYPE CONSOLE}
uses  SysUtils;
begin
      Writeln('Hello, World!');
      Writeln('Привет, Мартышка!');
      Readln;
end.
```

Причина – в несоответствии кодировок консольного окна и текстового файла.

Проблема решается вставкой в начале программы вызовов двух системных процедур, вот они:

```
      SetConsoleCP(1251);

      SetConsoleOutputCP(1251);
```

Процедуры спрятаны в модуле **Windows**, поэтому вам придется добавить его в список **Uses**. Пример такой программы представлен ниже.

```
program Rus;

{$APPTYPE CONSOLE}

uses SysUtils, Windows;

begin

{

Следующие вызовы процедур переключают консоль на кодовую страницу CP1251 (Win-1251). Если всё же русские буквы показываются неверно, откройте системное меню консоли (в левом верхнем углу окна) и выберите: Свойства -> закладка "Шрифт" -> Lucida Console.

}

 SetConsoleCP(1251);

 SetConsoleOutputCP(1251);

 Writeln('Привет, Мартышка! (Lucida Console CP1251)');

 Readln;

end.
```

Впрочем, при первом запуске и это не приведет к успеху (рис. 177).

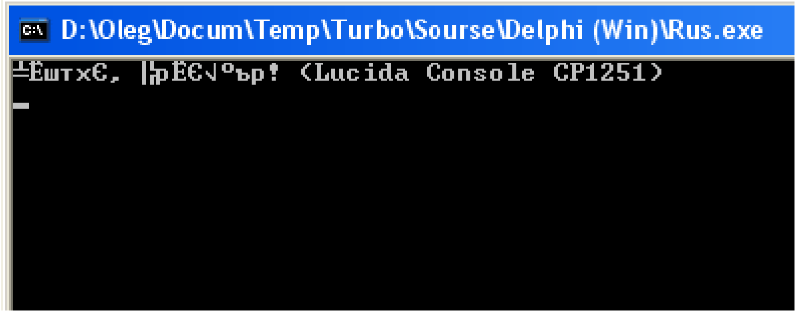


Рис. 177 - Вид консольного окна при первом запуске программы

Вам следует настроить шрифт консольного окна, выполнив следующие действия.

Щелкните системное меню консольного окна и выберите пункт «Свойства» (рис. 178).

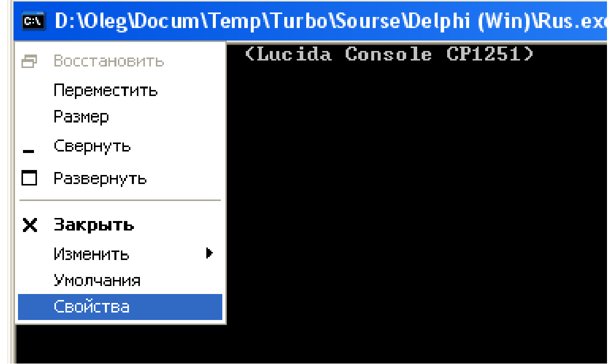
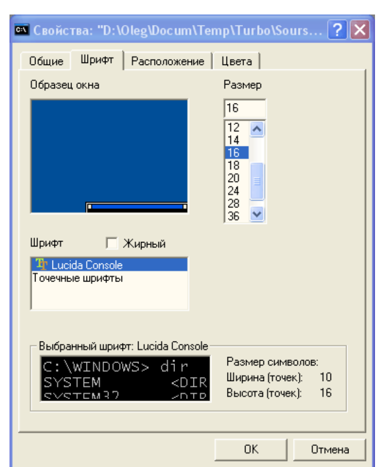


Рис. 178 - Выбор пункта меню «Свойства»

В появившемся окне выберите вкладку «Шрифт», а затем шрифт «Lucida Console» (рис. 179 слева). После нажатия кнопки *ОК* появится запрос, где можно подтвердить выбор шрифта для всех консольных окон (рис. 179 справа).



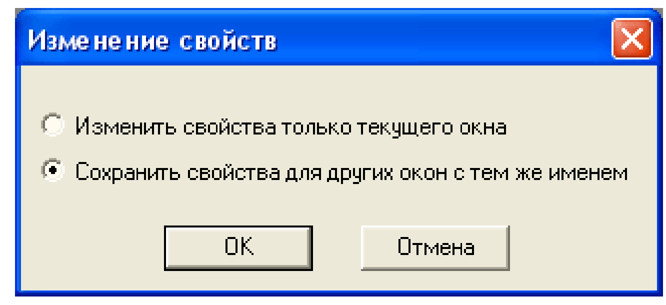


Рис. 179 - Вкладка «Шрифт» (слева) и окно подтверждения (справа)

Результат настройки не заставит себя ждать (рис. 180).

Рис. 180 - Русский текст в консольном окне

Приложение B
Особенности IDE Pascal ABCNet

Интегрированная среда разработки Pascal ABCNet пригодна для решения многих задач из этой книги. Я рекомендую её новичкам, делающим первые шаги в изучении Паскаля. Эта IDE проста в установке, не требует настройки, и отличается удобным русскоязычным оконным интерфейсом. IDE Pascal ABCNet создана в стенах Южного федерального университета, её можно бесплатно скачать с сайта http://pascalabc.net.

Ознакомимся с некоторыми особенностями этой IDE.

При первом вызове IDE появляется пустое окно, в которое можно ввести текст программы и сохранить под нужным именем (рис. 181). Если открыть несколько файлов с программами, каждый из них будет помещен в отдельной вкладке.

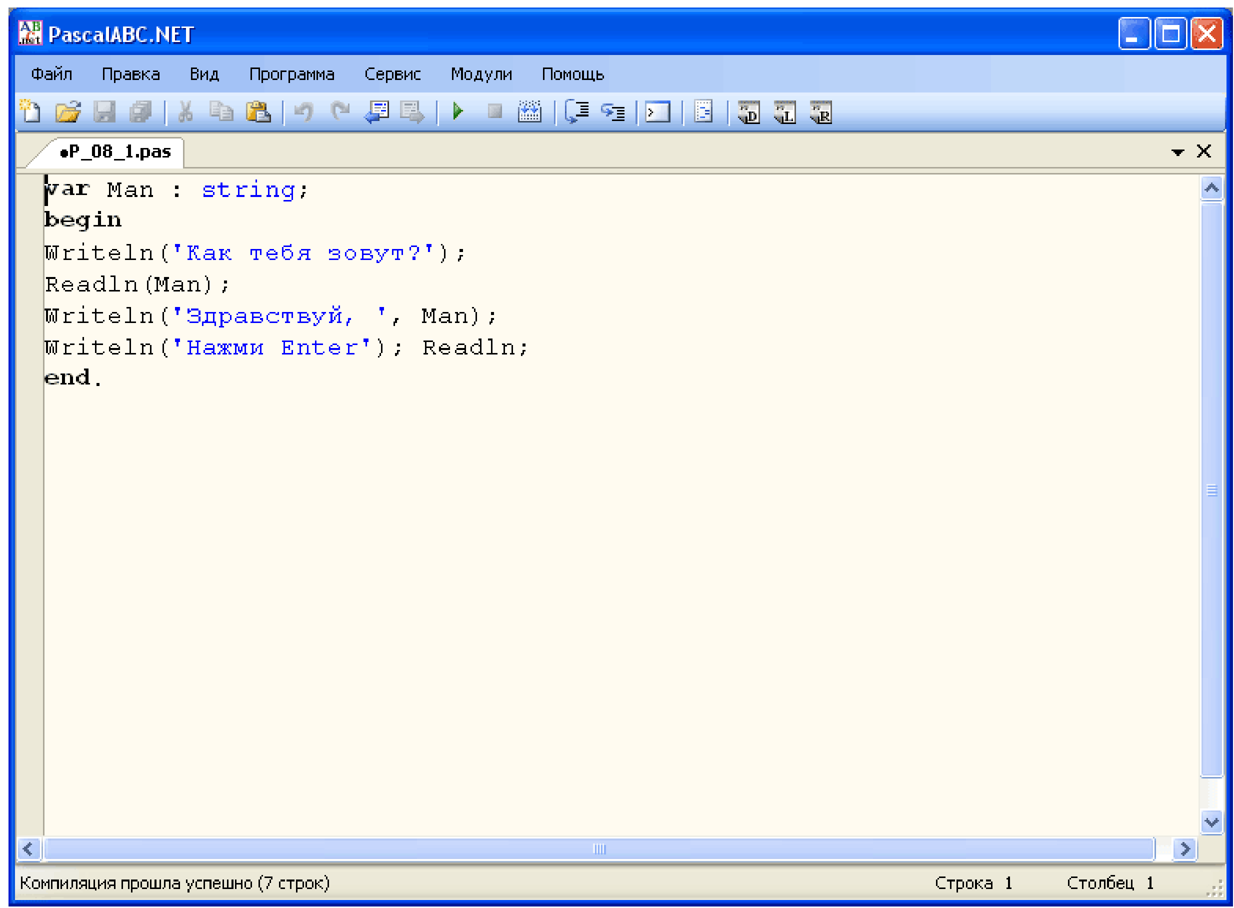


Рис. 181 - Окно IDE Pascal ABCNet

Готовая программа запускается клавишей *F9* или соответствующей кнопкой на панели инструментов.

Результаты, формируемые программой, выводятся в область консоли в нижней части окна (рис. 182). Здесь же расположено поле для ввода данных

пользователем. Для завершения ввода нажимается клавиша *Enter* или кнопка «Ввести». Кнопкой «Завершить» выполнение программы прекращается досрочно, то же самое случится при нажатии комбинации *Ctrl+F2*.

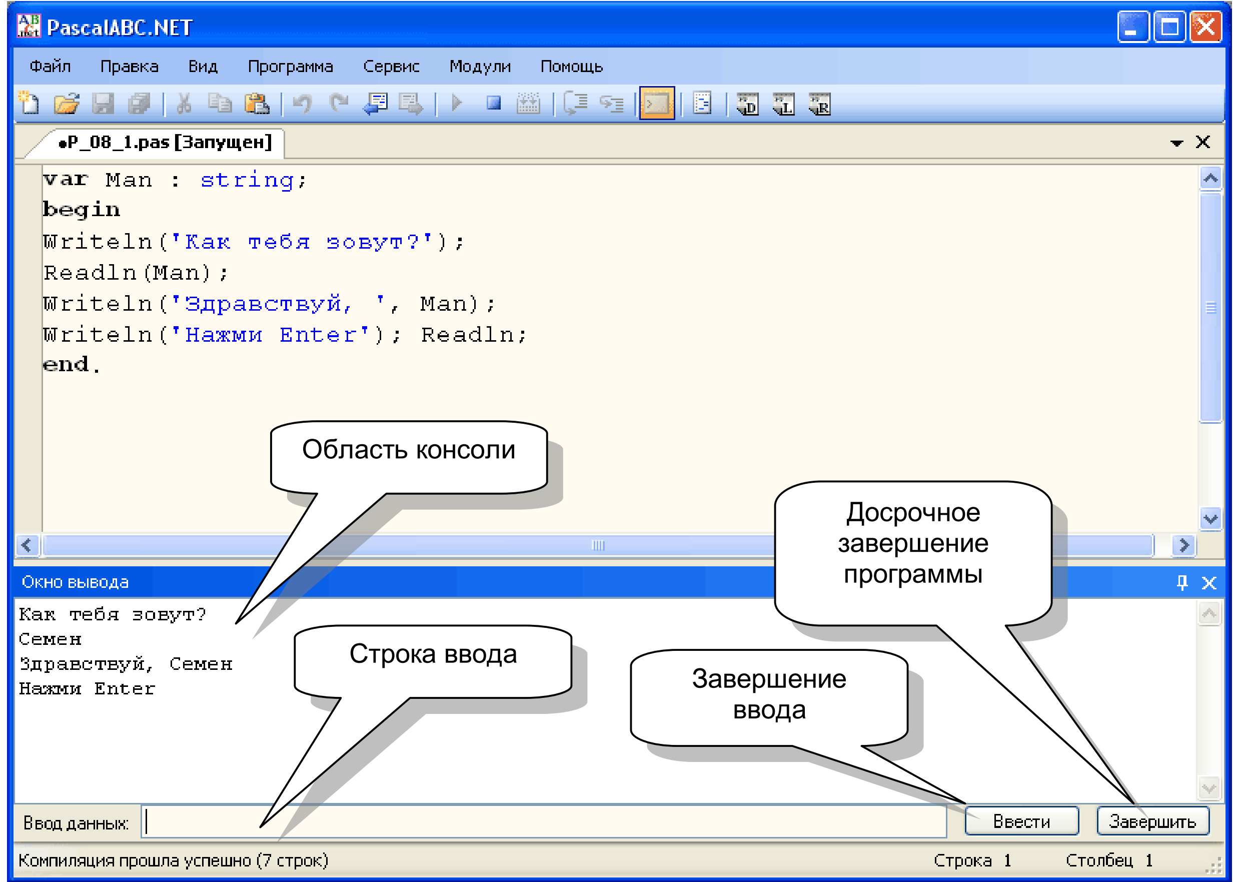


Рис. 182 - Протокол работы запущенной программы

Запуск программы в пошаговом режиме выполняется кнопками на панели инструментов, либо через меню. В пошаговом режиме доступен просмотр локальных переменных (рис. 183), а также другой информации о программе (на соседних вкладках).

Разработчики IDE Pascal ABCNet стремились, очевидно, к совместимости её со «стандартной» IDE Borland Pascal в той мере, насколько это возможно. Но объектная технология «точка Net» диктует своё, – полной совместимости не получилось.

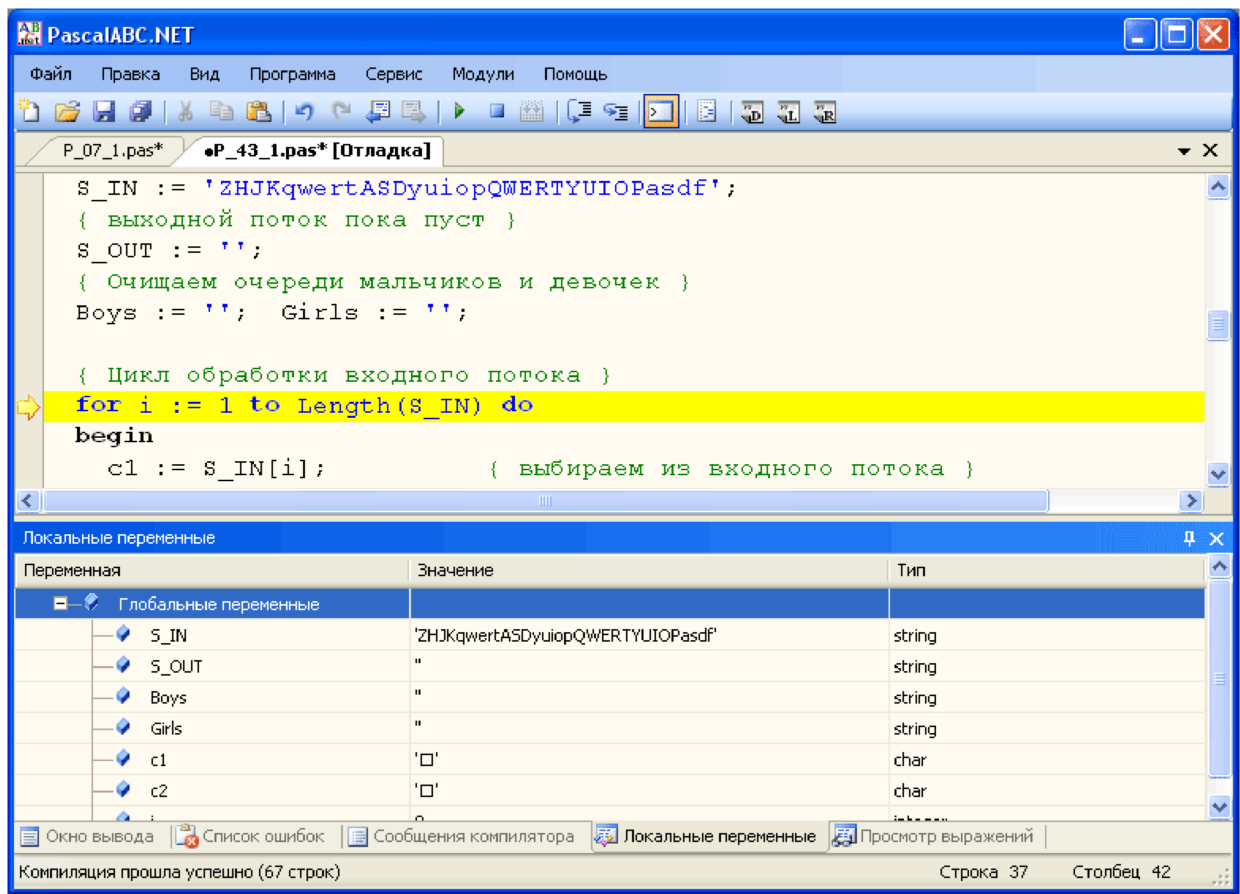


Рис. 183 - Просмотр переменных в пошаговом режиме

Так, например, в данной IDE не существует встроенной функции **Assigned**. Но вы можете написать её сами.

```
function Assigned (p: pointer): boolean;
begin
      Assigned:= p <> nil
end;
```

Ниже перечислены другие особенности Pascal ABCNet, которые следует учесть при переделке примеров данной книги.

- Отсутствует процедура **FillChar**. Записи и массивы нужно заполнять явным образом.
- Нельзя назначать файловым переменным пустое имя, связывая их, таким образом, с экраном и клавиатурой.
- Под символ отводится не один, а два байта (используется UNICODE).
- Строки являются объектами, поэтому доступ к нулевому элементу (байту длины) невозможен.

Приложение Г
Зарезервированные слова

Ключевые слова, которые нельзя применять по иному назначению.

Ключевое слово	Назначение
AND	Операция логического умножения «И»
ARRAY	Объявление массива
ASM	Начало блока инструкций на ассемблере
BEGIN	Начало блока операторов
CASE	Начало оператора множественного выбора
CONST	Начало секции объявления констант
CONSTRUCTOR	Объявление конструктора объекта
DESTRUCTOR	Объявление деструктора объекта
DIV	Операция целочисленного деления
DO	Элемент операторов **FOR, WITH, WHILE**
DOWNTO	Элемент оператора цикла **FOR-DOWNTO-DO**
ELSE	Элемент условных операторов **IF, CASE**
END	Завершение блоков **BEGIN-END, CASE-END, ASM-END, RECORD-END**
EXPORTS	Объявление списка экспорта для DLL
FALSE	Логическое значение «ЛОЖЬ»
FILE	Тип файловой переменной
FOR	Элемент оператора цикла **FOR-TO-DO**
FUNCTION	Объявление функции
GOTO	Безусловный переход на метку
IF	Элемент условного оператора **IF-THEN-ELSE**
IMPLEMENTATION	Объявление секции реализации модуля
IN	Проверка принадлежности элемента множеству
INHERITED	Квалификатор унаследованного метода объекта
INLINE	Оператор вставки ассемблерных кодов
INTERFACE	Объявление секции интерфейса модуля
LABEL	Объявление меток
LIBRARY	Объявление библиотечного модуля
MOD	Операция нахождения остатка от деления
NIL	Пустой указатель

Ключевое слово	Назначение
NOT	Логическое отрицание «НЕ»
OBJECT	Объявление типа «объект»
OF	Элемент оператора **CASE-OF-END**
OR	Логическое сложение «ИЛИ»
PACKED	Объявление упакованного массива (устарело)
PROCEDURE	Объявление процедуры
PROGRAM	Объявление программы
RECORD	Начало объявления записи **RECORD-END**
REPEAT	Элемент оператора цикла с проверкой в конце **REPEAT-UNTIL**
SET	Объявление множества **SET OF**
SHL	Операция сдвига влево
SHR	Операция сдвига вправо
STRING	Объявление строкового типа
THEN	Элемент условного оператора **IF-THEN-ELSE**
TO	Элемент оператора цикла **FOR-TO-DO**
TRUE	Логическое значение «ИСТИНА»
TYPE	Начало секции объявления типов
UNIT	Объявление имени модуля
UNTIL	Элемент оператора цикла с проверкой в конце **REPEAT-UNTIL**
USES	Объявление внешних модулей
VAR	Начало секции объявления переменных
WHILE	Элемент оператора цикла с проверкой в начале **WHILE-DO**
WITH	Элемент оператора раскрытия записи **WITH-DO**
XOR	Логическое сравнение («ИСКЛЮЧАЮЩЕЕ ИЛИ»)

Директивы, имена которых не рекомендуется использовать по иному назначению.

Директива	Назначение в Borland Pascal
ABSOLUTE	Назначение переменной абсолютного адреса в памяти
ASSEMBLER	Объявление процедуры на ассемблере
EXPORT	Список экспорта
EXTERNAL	Объявление внешних имен для ассемблера
FAR	Спецификатор процедуры с «дальним» вызовом
FORWARD	Предварительное определение процедуры или функции
INDEX	Указание индекса процедуры в DLL
INTERRUPT	Спецификатор процедуры обработки прерывания
NAME	Импорт процедуры из DLL по имени
NEAR	Спецификатор процедуры с «ближним» вызовом
PRIVATE	Начало секции приватных полей объекта
PUBLIC	Начало секции публичных полей объекта
RESIDENT	Спецификатор резидентного элемента DLL
VIRTUAL	Спецификатор виртуального метода объекта

Приложение Д
Ошибки компиляции

Номер ошибки	Сообщение	Пояснение
1	Out of memory	Недостаточно оперативной памяти для работы компилятора. Воспользуйтесь средствами расширения оперативной памяти MS-DOS.
2	Identifier expected	В данном месте программы должен находиться идентификатор. Возможно, имеется попытка использовать зарезервированное слово.
3	Unknown identifier	Неизвестный (не определенный ранее) идентификатор.
4	Duplicate identifier	Повторное определение идентификатора.
5	Syntax error	Синтаксическая ошибка (нарушены правила написания предложения).
6	Error in real constant	Ошибка в изображении вещественного числа.
7	Error in integer constant	Ошибка в изображении целого числа.
8	String constant exceeds line	Строковая константа превышает допустимый размер (255 символов). Вероятно, пропущена закрывающая кавычка.
10	Unexpected end of file	Неожиданное завершение программы. Вероятно, не сбалансировано число зарезервированных слов begin и end, неправильно оформлен файл.
11	Line too long	Компилируемая строка программы превышает 127 символов.
12	Type identifier expected	В данном месте программы требуется идентификатор типа.
13	Too many open files	Попытка открыть в среде программирования количество файлов больше, чем допускается в операционной системе. Максимальное число одновременно открываемых файлов определяется строкой FILES=NN в файле CONFIG.SYS.
14	Invalid file name	Неверный путь или имя файла.

Номер ошибки	Сообщение	Пояснение
15	File not found	Файл не найден ни в текущем, ни в заданном каталоге.
16	Disk full	Недостаточно места на диске, куда записывается информация.
17	Invalid compiler directive	Ошибка в директиве компилятора, или она используется в недопустимом месте.
18	Too many files	Слишком много файлов используется при компиляции.
19	Undefined type in pointer def	При определении типа-указателя используется неизвестный базовый тип.
20	Variable identifier expected	В данном месте программы должен быть идентификатор переменной.
21	Error in type	Ошибка в определении типа.
22	Structure too large	Размер данных превышает ограничение в 65520 байт.
23	Set base type out of range	Тип-множество имеет более 256 элементов или содержит элементы с порядковым номером за пределами 0..255.
24	File components may not be files or objects	Компонентами файловой переменной не могут быть файлы или объекты.
25	Invalid string length	Неверная длина строки (максимум 255 символов).
26	Type mismatch	Несовместимые типы в операциях присваивания, в выражениях или у индекса массива. Тип фактического параметра при обращении к подпрограмме не соответствует типу формального параметра.
27	Invalid subrange base type	В типе-диапазоне может использоваться только порядковый тип.
28	Lower bound > than upper bound	В типе-диапазоне нижняя граница больше, чем верхняя.
29	Ordinal type expected	Здесь может использоваться только порядковый тип.
30	Integer constant expected	Здесь можно использовать только константу целого типа.
31	Constant expected	Здесь можно использовать только константу.

Номер ошибки	Сообщение	Пояснение
32	Integer or real constant expected	Здесь можно использовать только числовую константу.
33	Pointer Type identifier expected	Здесь должен быть указатель.
34	Invalid function result type	Недопустимый тип результата функции.
35	Label identifier expected	Здесь должен быть идентификатор метки.
36	BEGIN expected	Здесь ожидается зарезервированное слово BEGIN.
37	END expected	Здесь ожидается зарезервированное слово END.
38	Integer expression expected	Выражение должно быть целого типа.
39	Ordinal expression expected	Выражение должно быть порядкового типа.
40	Boolean expression expected	Выражение должно быть логического типа.
41	Operand types do not match	Типы операндов не совместимы друг с другом.
42	Error in expression	Ошибка, в выражении (например, пропущен знак операции между операндами).
43	Illegal assignment	Неправильно присвоено значение переменной.
44	Field identifier expected	Требуется указать поле записи.
45	Object file too large	Размер объектного файла превышает 64 кбайта.
46	Undefined EXTERN	Не найдена внешняя процедура или функция.
47	Invalid object file record	Файл, по-видимому, не является объектным.
48	Code segment too large	Размер кодового сегмента превышает 65520 байт.
49	Data segment too large	Размер сегмента данных превышает 65520 байт.
50	DO expected	Здесь следует поместить зарезервированное слово DO.
51	Invalid PUBLIC definition	Неправильное использование директивы PUBLIC в подпрограмме, написанной на ассемблере.

Номер ошибки	Сообщение	Пояснение
52	Invalid EXTRN definition	Неправильное использование директивы EXTRN в подпрограмме, написанной на ассемблере.
53	Too many EXTRN definitions	Слишком много директив EXTRN.
54	OF expected	Здесь следует поместить зарезервированное слово OF.
55	INTERFACE expected	В модуле пропущено зарезервированное слово INTERFACE.
56	Invalid relocatable reference	Неправильная перемещаемая ссылка в подпрограмме, написанной на ассемблере.
57	THEN expected	Здесь следует поместить зарезервированное слово THEN
58	TO or DOWNTO expected	Здесь следует поместить зарезервированное слово TO или DOWNTO
59	Undefined forward	Заголовок подпрограммы объявлен с директивой FORWARD, но сама подпрограмма далее не описана.
61	Invalid typecast	Неверное приведение типов. При преобразовании типа величины исходного и результирующего типа имеют различные размеры.
62	Division by zero	Попытка деления на нуль.
63	Invalid file type	Эта процедура работы с файлом не поддерживает данный тип файла.
64	Cannot read or write variables of this type	Недопустимый тип параметра у процедур Read, Readln, Write, Writeln.
65	Pointer variable expected	Эта переменная должна быть указателем.
66	String variable expected	Эта переменная должна иметь строковый тип.
67	String expression expected	Эта выражение должно иметь строковый тип.
68	Circular unit reference	Циклическая ссылка модулей друг на друга. Необходимо поместить ссылку на модули (USES) в секции IMPLEMENTATION.
69	Unit name mismatch	Имена модуля и файла, в котором он находится, не совпадают.

Номер ошибки	Сообщение	Пояснение
70	Unit version mismatch	Один или несколько используемых модулей изменены после их компиляции (несоответствие версий модулей). Необходимо выполнить их повторную компиляцию.
71	Internal stack overflow	Внутренний стек компилятора переполнен из-за глубокой вложенности операторов.
72	Unit file format error	Ошибка в формате скомпилированного модуля. Возможно, он был скомпилирован предыдущей версией компилятора.
73	Implementation expected	Пропущено объявление исполнительной части модуля.
74	Constant and case types don't match	Недопустимое значение константы.
75	Record or object variable expected	Переменная должна иметь тип записи или объекта.
76	Constant out of range	Используемая константа имеет недопустимое значение .
77	File variable expected	Эта переменная должна быть файлового типа.
78	Pointer expression expected	Это выражение должно иметь тип указателя.
79	Integer or real expression expected	Это выражение должно быть числовым.
80	Label not within current block	Метка находится за пределами данного блока.
81	Label already defined	Повторное использование метки.
82	Undefined label in preceding stmt part	Метка объявлена, но ни один оператор ею не помечен.
83	Invalid @ argument	Неверный аргумент у операции взятия адреса @.
84	UNIT expected	В модуле пропущено зарезервированное слово Unit.
85	";" expected	Здесь пропущена точка с запятой.
86	":" expected	Здесь пропущено двоеточие.
87	"," expected	Здесь пропущена запятая.
88	"(" expected	Здесь пропущена открывающая круглая скобка.

Номер ошибки	Сообщение	Пояснение
89	")" expected	Здесь пропущена закрывающая круглая скобка.
90	"=" expected	Здесь пропущен знак равенства.
91	":=" expected	Здесь пропущен знак присваивания.
92	"[" or "(." expected	Здесь пропущена открывающая квадратная скобка или скобка с точкой.
93	"]" or ".)" expected	Здесь пропущена закрывающая квадратная скобка или скобка с точкой.
94	"." expected	Здесь пропущена точка.
95	".." expected	Здесь пропущены две точки.
96	Too many variables	Размер глобальных или локальных переменных подпрограммы превышает 64Кбайт.
97	Invalid FOR control variable	Неправильная переменная цикла FOR.
98	Integer variable expected	Здесь должна быть переменная целого типа.
99	Files types are not allowed here	Файловый или процедурный тип здесь не допускается.
100	String length mismatch	Длина строковой константы не соответствует размеру массива символов.
101	Invalid ordering of fields	Порядок следования полей в типизированных константах типов записи или объекта должен соответствовать порядку их следования при объявлении типа.
102	String constant expected	Здесь должна быть строковая константа.
103	Integer or real variable expected	Здесь должна быть числовая переменная целого или вещественной типа.
104	Ordinal variable expected	Здесь должна быть переменная порядкового типа.
105	INLINE error	Ошибка в подпрограмме с директивой Inline.
106	Character expression expected	Здесь должно быть выражение символьного типа.
107	Too many relocation items	Главная программа слишком велика. Следует выделить часть её в подпрограммы.

Номер ошибки	Сообщение	Пояснение
108	Overflow in arithmetic operation	Переполнение при выполнении арифметической операции с числами целого типа.
109	No enclosing For, While or Repeat statement	Стандартные процедуры Break или Continue используются вне циклов For, While или Repeat.
110	Cannot run a unit	Программный модуль выполнить нельзя, необходимо написать программу, вызывающую этот модуль.
111	Compilation aborted	Компиляция прервана пользователем
112	CASE constant out of range	Значение константы в операторе CASE вышло за пределы от -32768 до 32767.
113	Error in statement	Ошибка в операторе.
114	Cannot call an interrupt procedure	Нельзя непосредственно вызвать процедуру обработки прерывания.
116	Must be in 8087 mode to compile	Типы Single, Double, Extended и Comp можно использовать только при наличии арифметического сопроцессора.
117	Target address not found	При использовании команды меню «Search | Find error» нельзя найти оператор для заданного адреса.
118	Include files are not allowed here	В данном месте нельзя использовать файл, подключаемый директивой {$I filename}. Оператор должен полностью находиться в одном файле.
119	No inherited methods are accessible here	Зарезервированное слово inherited использовано за пределами метода объекта или внутри метода объекта, у которого нет предков.
121	Invalid qualifier	Неправильный квалификатор, попытка задать индекс у параметра, не являющегося массивом.
122	Invalid variable reference	Неправильная ссылка на переменную, вероятно, не разыменован указатель.
123	Too many symbols	Символические имена программы занимают больше, чем 64 Кбайт. Следует попробовать разделить программу на части.

Номер ошибки	Сообщение	Пояснение
124	Statement part too large	Блок программы занимает более 24 Кбайт. Необходимо разбить его на процедуры и функции.
126	Files must be var parameters	Файловую переменную можно передавать в подпрограмму только по ссылке VAR.
127	Too many conditional symbols	Слишком много символов у параметра условной компиляции.
128	Misplaced conditional directive	Пропущена часть директивы условной компиляции.
130	Error in initial conditional defines	Ошибка в директиве условной компиляции
131	Header does not match previous definition	Заголовок процедуры не соответствует заголовку, объявленному с директивой FORWARD, или заготовку в интерфейсной части модуля.
133	Cannot evaluate this expression	Нельзя вычислить константное выражение или выражение, используемое в окне отладки.
134	Expression incorrectly terminated	Неправильно завершено выражение.
135	Invalid format specifier	Неправильная спецификация формата.
136	Invalid indirect reference	Неправильная косвенная ссылка.
137	Structured variables are not allowed here	Недопустимая операция над структурированными данными.
138	Cannot evaluate without System unit	В библиотечном файле Turbo.tpl отсутствует модуль System.
139	Cannot access this symbol	Нет доступа к этому символу.
140	Invalid floating-point operation	Переполнение при операциях с вещественными числами или деление на ноль.
141	Cannot compile overlays to memory	Программа, использующая оверлеи, должна быть скомпилирована на диск.
142	Pointer or procedural variable expected	Здесь должен быть указатель или переменная процедурного типа.
143	Invalid procedure or function reference	Неправильный вызов процедуры или функции. Возможно, её требуется скомпилировать в режиме {$F+}.

Номер ошибки	Сообщение	Пояснение
144	Cannot overlay this unit	Попытка использовать модуль в оверлейной программе, который не скомпилирован с директивой {$O+}.
146	File access denied	Неправильное обращение к файлу (например, запись в файл, доступный только для чтения, или используется имя каталога, а не файл).
147	Object type expected	Здесь должна быть переменная типа объект.
148	Local object types are not allowed	Нельзя определять переменную типа объекта внутри подпрограммы (локально).
149	Virtual expected	Этот метод должен быть виртуальным.
150	Method identifier expected	Здесь должен быть идентификатор метода.
151	Virtual constructors are not allowed	Конструктор не может быть виртуальным.
152	Constructor identifier expected	Здесь должен быть идентификатор конструктора.
153	Destructor identifier expected	Здесь должен быть идентификатор деструктора.
154	Fail only allowed within constructors	Процедура Fail вызвана не из конструктора, что недопустимо.
155	Invalid combination of opcode and operands	Неправильный набор операндов у команды ассемблера.
156	Memory reference expected	В команде ассемблера должна быть ссылка на память.
157	Cannot add or subtract relocatable symbols	Нельзя складывать или вычитать перемещаемые операнды в ассемблере.
158	Invalid register combination	Неправильная комбинация регистров.
159	286/287 instructions are not enabled	Нельзя использовать эту команду для процессоров 80286/80287.
160	Invalid symbol reference	Этот параметр нельзя использовать как операнд в команде ассемблера.
162	ASM expected	Здесь должно быть зарезервированное слово ASM.

Приложение E
Ошибки исполнения

Номер ошибки	Сообщение	Пояснение
1	Invalid function number	Вызов несуществующей функции MS-DOS.
2	File not found	Не найден файл.
3	Path not found	Не найден каталог.
4	Too many open files	Слишком много открытых файлов. Максимальное число одновременно открываемых файлов определено в системном файле «CONFIG.NT» («CONFIG.SYS»).
5	File access denied	Отказано в доступе к файлу.
6	Invalid file handle	Неправильный описатель файла.
12	Invalid file access code	Неправильный режим доступа к файлу.
15	Invalid drive number	Неправильная буква (номер) устройства.
16	Cannot remove current directory	Нельзя удалить текущий каталог.
17	Cannot rename across drives	Нельзя переименовать файл заменой буквы устройства.
18	No more files	Процедура **`FindFirst`** или **`FindNext`** не нашла файл.
100	Disk read error	Попытка чтения за концом файла.
101	Disk write error	Ошибка (переполнение) диска.
102	File not assigned	Обращение с файлом, для которого не выполнена процедура **`Assign`**.
103	File not open	Файл не открыт.
104	File not open for input	Файл не открыт для чтения.
105	File not open for output	Файл не открыт для записи.
106	Invalid numeric format	Недопустимый формат целого числа.
150	Disk is write-protected	Запись на защищенный от записи диск.
151	Bad drive request struct length	Ошибка, формируемая драйвером.
152	Drive not ready	Внешнее устройство не готово.
154	CRC error in data	Ошибка при записи на внешнее устройство.

Номер ошибки	Сообщение	Пояснение
156	Disk seek error	Попытка чтения-записи за пределами файла.
157	Unknown media type	Нельзя распознать тип устройства.
158	Sector Not Found	Не найден сектор диска.
159	Printer out of paper	В принтере нет бумаги.
160	Device write fault	Ошибка на устройстве при записи.
161	Device read fault	Ошибка на устройстве при чтении.
162	Hardware failure	Ошибка устройства ввода-вывода.
200	Division by zero	Деление на ноль.
201	Range check error	Нарушение диапазона.
202	Stack overflow error	Переполнение стека программы.
203	Heap overflow error	Переполнение динамической памяти.
204	Invalid pointer operation	Вызов **`Dispose`** или **`FreeMem`** с неверным указателем.
205	Floating point overflow	Переполнение вещественного числа.
206	Floating point underflow	Потеря значимости вещественного числа.
207	Invalid floating point operation	Ошибка действия с вещественным числом.
208	Overlay manager not installed	Диспетчер оверлеев не инициализирован.
209	Overlay file read error	Ошибка при чтении оверлейного файла.
210	Object not initialized	Объект не инициализирован.
211	Call to abstract method	Вызван абстрактный метод объекта.
212	Stream registration error	Ошибка регистрации типа данных для потока.
213	Collection index out of range	Выход индекса за пределы коллекции.
214	Collection overflow error	Переполнение коллекции.
215	Arithmetic overflow error	Переполнение в арифметической операции.
216	General Protection fault	Запись в недоступную область памяти.

Приложение Ж
Директивы управления компиляцией

Директивы для настройки реакции на ошибки времени выполнения
(Runtime errors)

Директива компилятора	Флажок в окне настройки	Описание
$R	Range Checking	Проверка допустимых диапазонов для индексов массивов и чисел.
$S	Stack Checking	Проверка переполнения стека программы.
$I	I/O Checking	Проверка ошибок ввода-вывода.
$Q	Overflow Checking	Проверка переполнения при целочисленных вычислениях.

Директивы, управляющие синтаксическим контролем
(Syntax options)

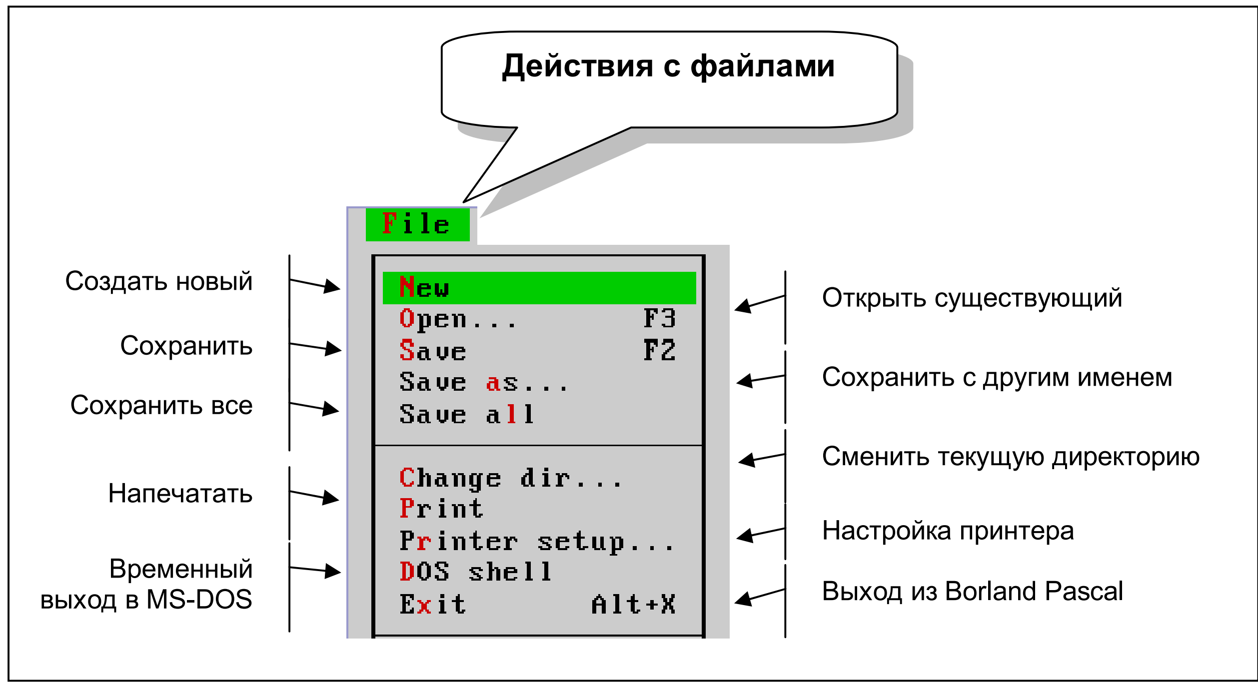
Директива компилятора	Флажок в окне настройки	Описание
$V	Strict Var Strings	Проверка совместимости типов для строк различной длины.
$B	Complete Boolean Eval	Вычисление полного булевого выражения.
$X	Extended Syntax	Разрешение вызова функций как процедур (возвращаемый результат игнорируется).
$T	Typed @ operator	Проверка совместимости типов указателей.
$P	Open parameters	Разрешение применения открытых параметров процедур и функций.

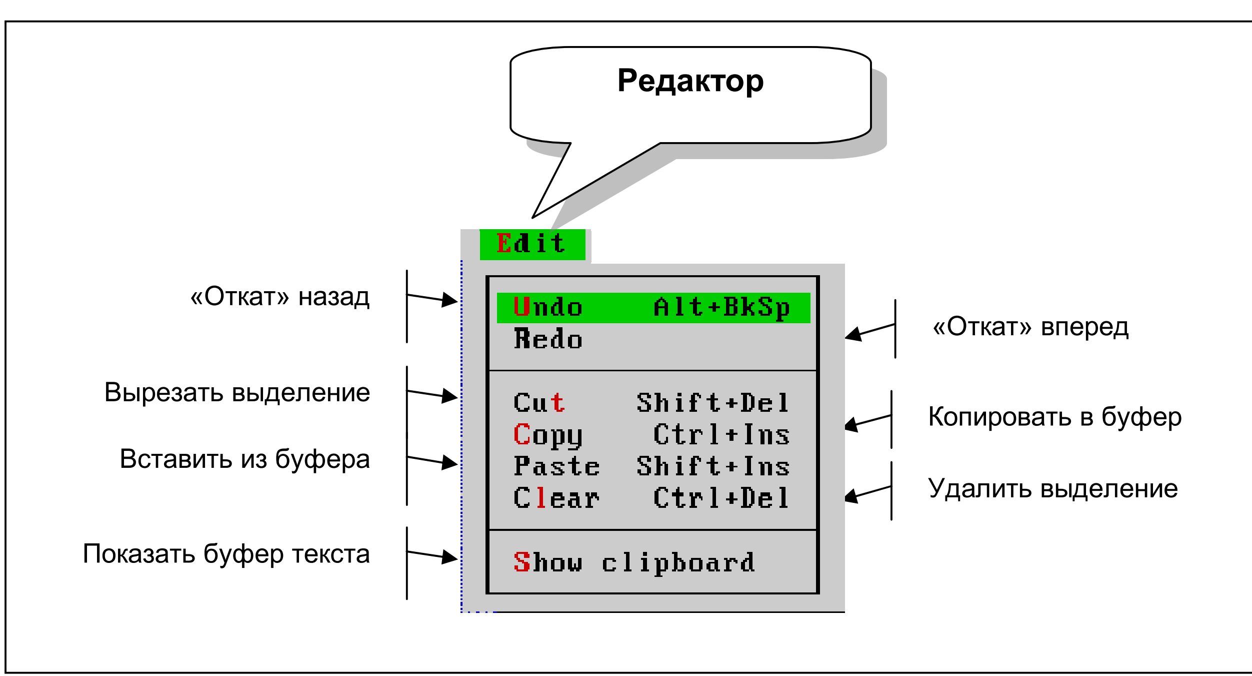
Директивы условной компиляции

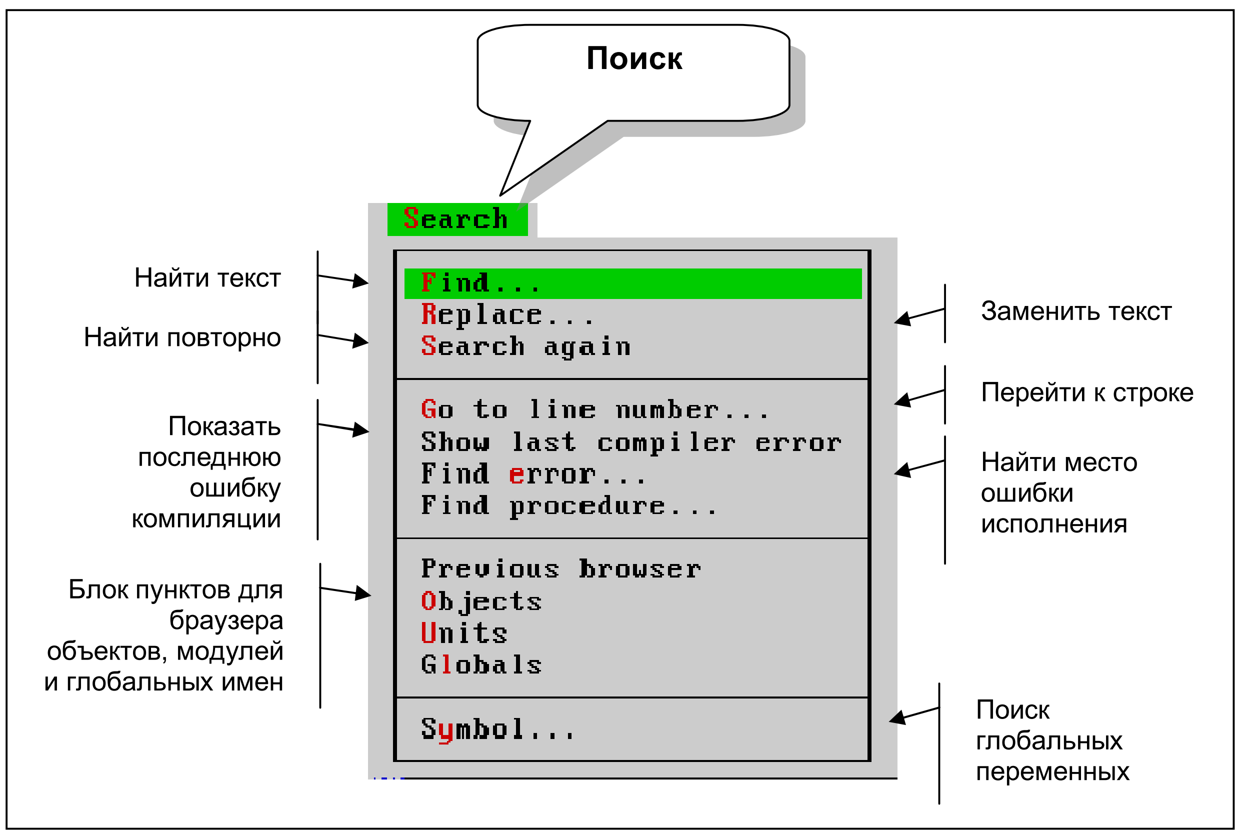
Директива компилятора	**Описание**
$DEFINE NNN	Определяет идентификатор с именем NNN, который может быть далее использован в директивах условной компиляции вида $IF…
$UNDEF NNN	Отменяет определение идентификатора NNN.
$IFDEF NNN	Разрешает компиляцию последующего текста, если идентификатор NNN был ранее определен. Компилируемый участок текста завершается директивами $ELSE или $ENDIF.
$IFNDEF NNN	Разрешает компиляцию последующего текста, если идентификатор NNN был ранее НЕ определен. Компилируемый участок текста завершается директивами $ELSE или $ENDIF.
$IFOPT	Разрешает компиляцию последующего текста, если указанная опция (директива) компилятора разрешена. Компилируемый участок текста завершается директивами $ELSE или $ENDIF.
$ELSE	Завершает положительную ветвь условной компиляции, последующий текст компилируется, если условие компиляции ложно. Компилируемый участок текста завершается директивой $ENDIF.
$ENDIF	Завершает директиву условной компиляции.

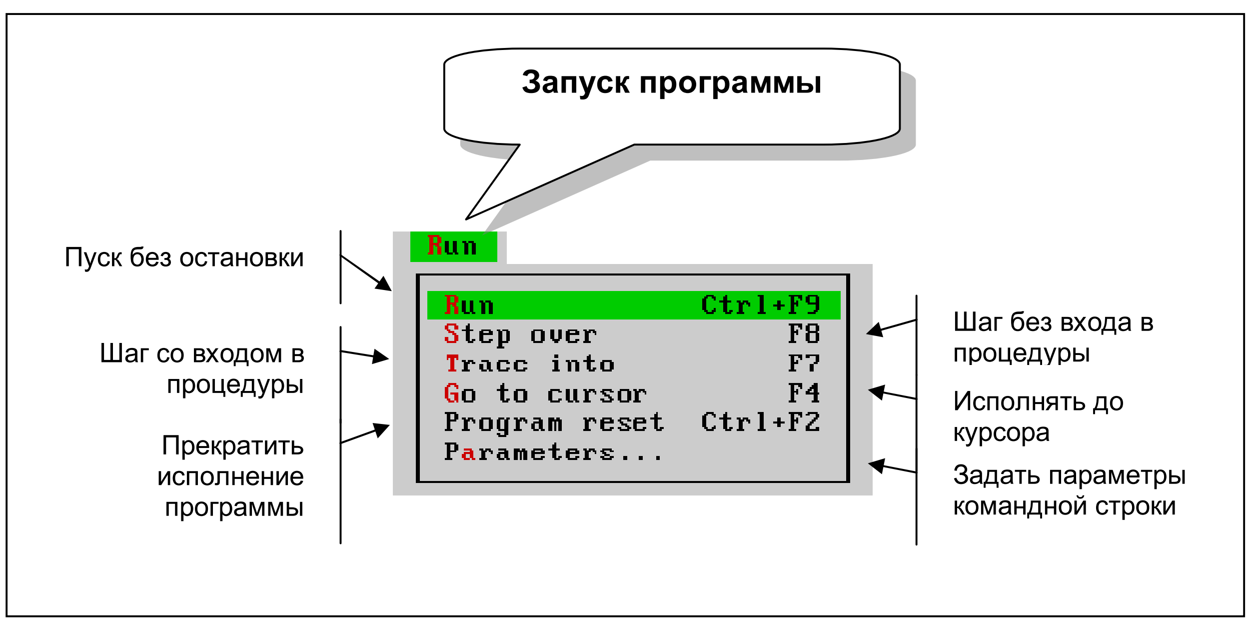
Приложение 3
Назначение пунктов меню

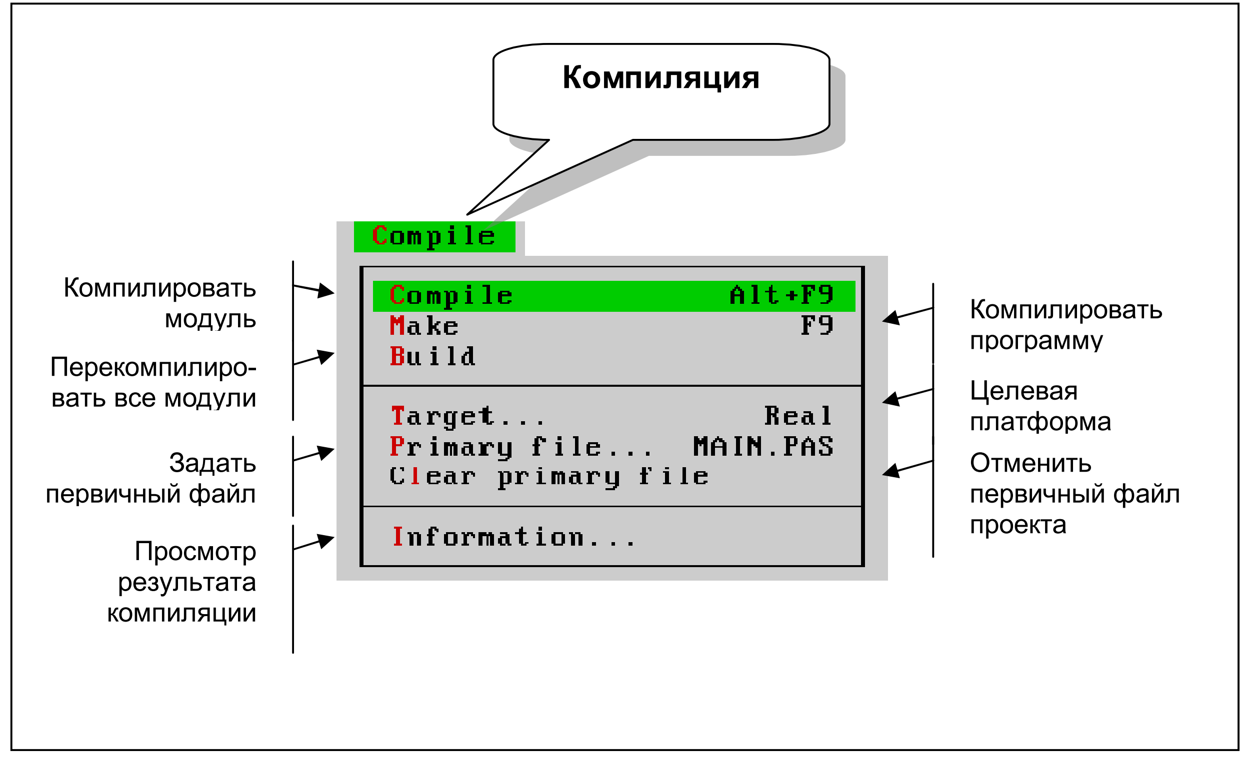
Это приложение содержит перевод пунктов меню IDE Free Pascal и Borland Pascal.

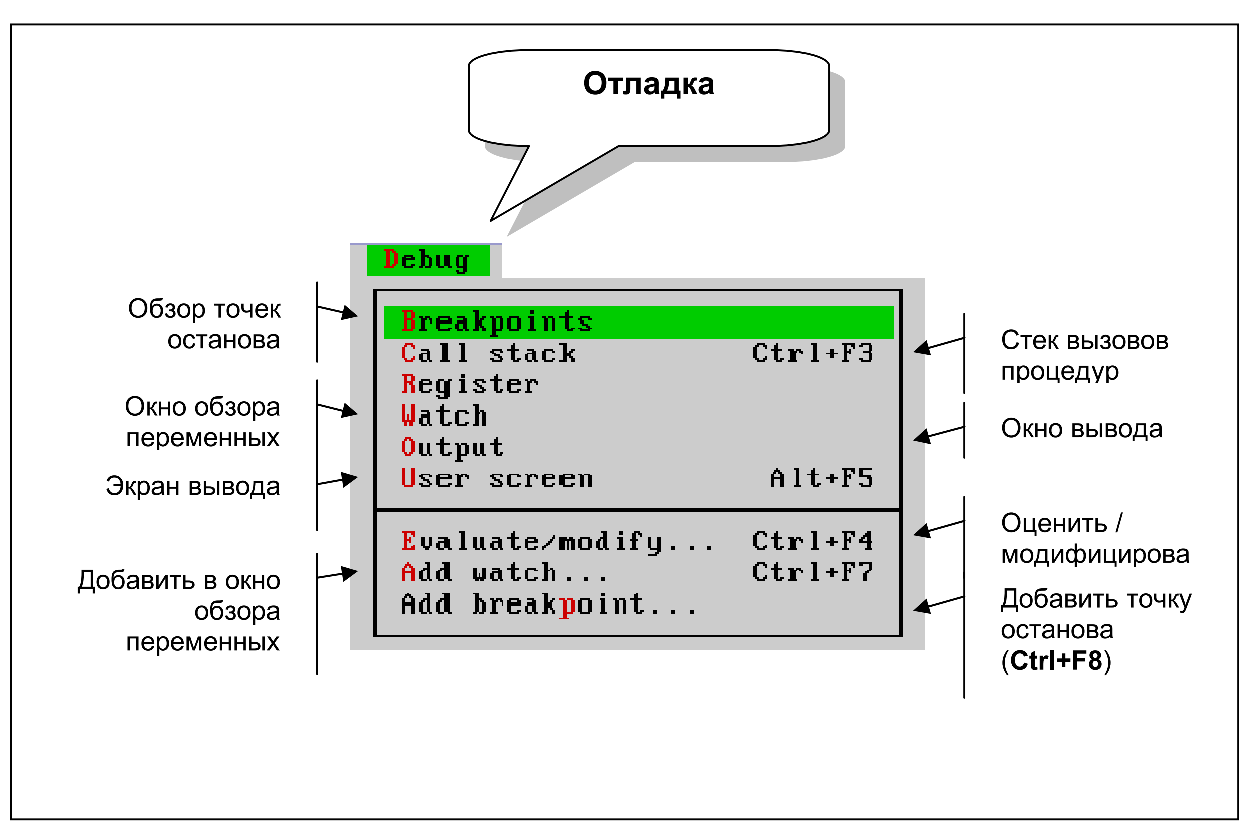


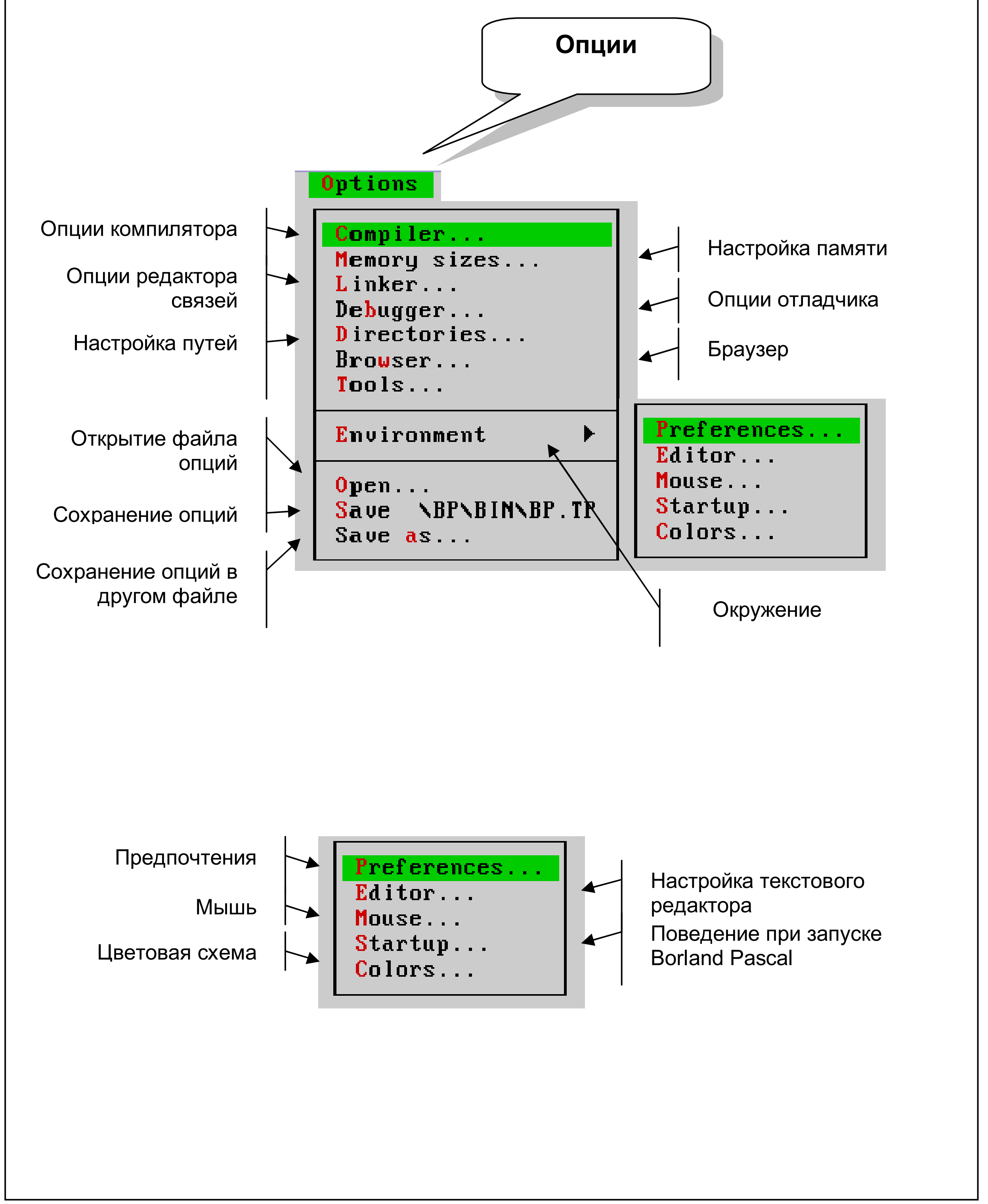


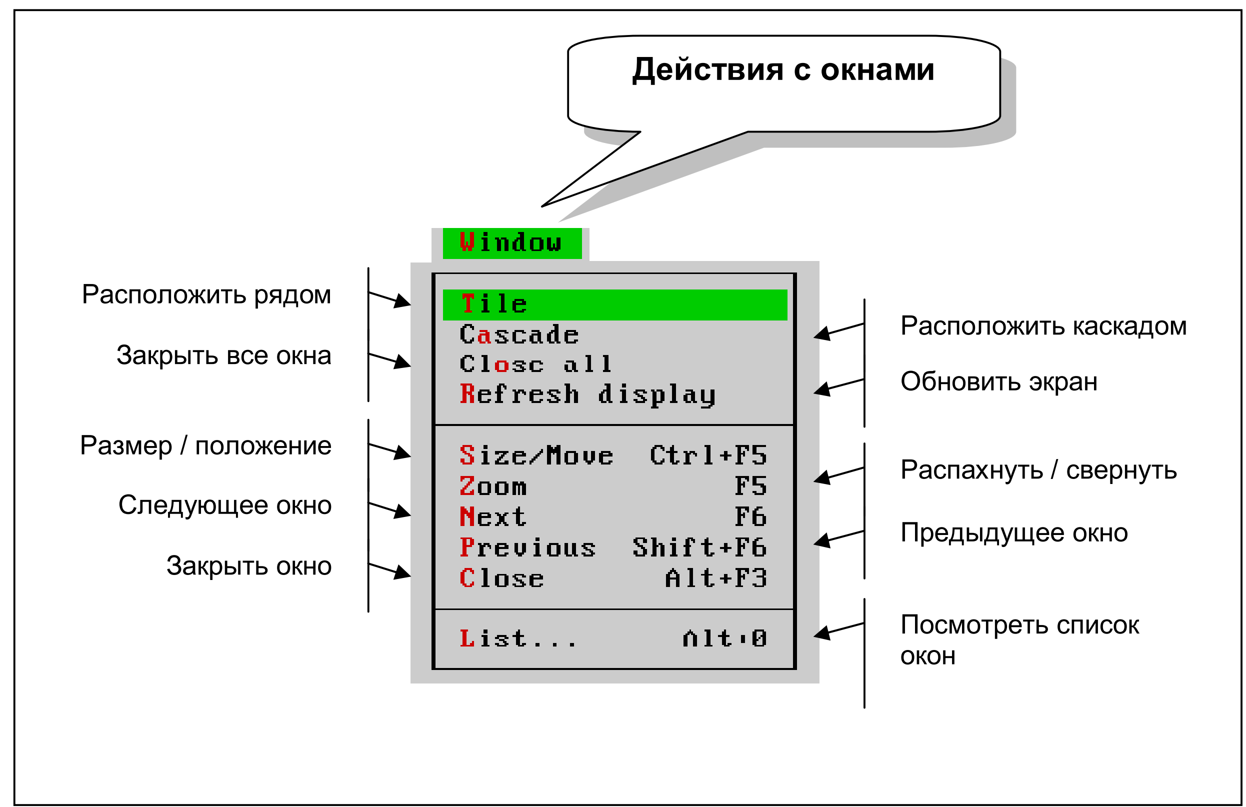

Поиск
Search
Найти текст
Найти повторно
Показать последнюю ошибку компиляции
Блок пунктов для браузера объектов, модулей и глобальных имен
Find...
Replace...
Search again
Go to line number...
Show last compiler error
Find error...
Find procedure...
Previous browser
Objects
Units
Globals
Symbol...
Заменить текст
Перейти к строке
Найти место ошибки исполнения
Поиск глобальных переменных

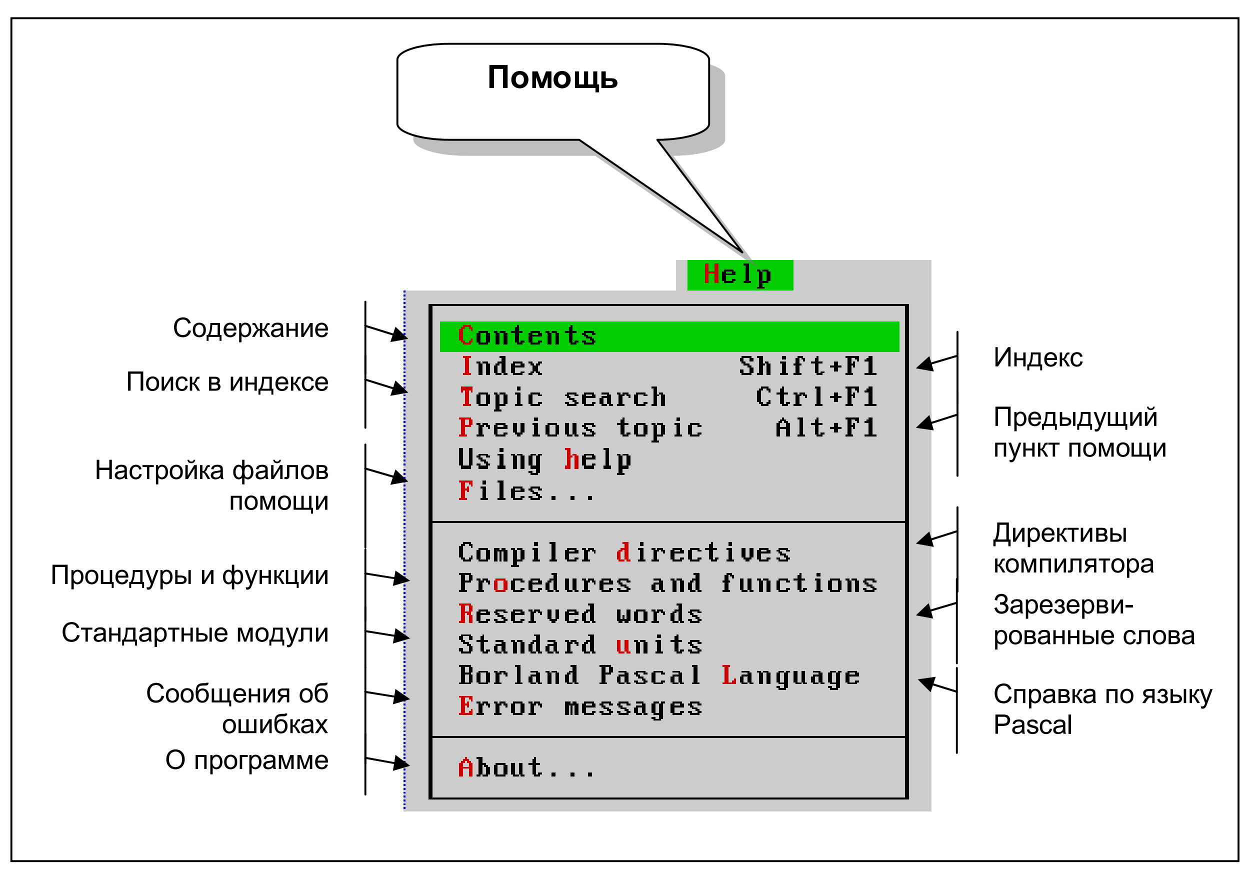

Запуск программы
Run
Пуск без остановки
Шаг со входом в процедуры
Прекратить исполнение программы
Run Ctrl+F9
Step over F8
Trace into F7
Go to cursor F4
Program reset Ctrl+F2
Parameters...
Шаг без входа в процедуры
Исполнять до курсора
Задать параметры командной строки


Компиляция
Compile
Compile Alt+F9
Make F9
Build
Target... Real
Primary file... MAIN.PAS
Clear primary file
Information...
Компилировать модуль
Перекомпилиро-вать все модули
Задать первичный файл
Просмотр результата компиляции
Компилировать программу
Целевая платформа
Отменить первичный файл проекта


Отладка
Debug
Breakpoints
Call stack Ctrl+F3
Register
Watch
Output
User screen Alt+F5
Evaluate/modify... Ctrl+F4
Add watch... Ctrl+F7
Add breakpoint...
Обзор точек останова
Окно обзора переменных
Экран вывода
Добавить в окно обзора переменных
Стек вызовов процедур
Окно вывода
Оценить / модифицирова
Добавить точку останова (Ctrl+F8)


Опции
Options
Compiler...
Memory sizes...
Linker...
Debugger...
Directories...
Browser...
Tools...
Environment
Open...
Save \BP\BIN\BP.TP
Save as...
Preferences...
Editor...
Mouse...
Startup...
Colors...
Опции компилятора
Опции редактора связей
Настройка путей
Открытие файла опций
Сохранение опций
Сохранение опций в другом файле
Настройка памяти
Опции отладчика
Браузер
Окружение
Preferences...
Editor...
Mouse...
Startup...
Colors...
Предпочтения
Мышь
Цветовая схема
Настройка текстового редактора
Поведение при запуске Borland Pascal


Действия с окнами
Window
Tile
Cascade
Close all
Refresh display
Size/Move Ctrl+F5
Zoom F5
Next F6
Previous Shift+F6
Close Alt+F3
List... Alt+0
Расположить рядом
Закрыть все окна
Размер / положение
Следующее окно
Закрыть окно
Расположить каскадом
Обновить экран
Распахнуть / свернуть
Предыдущее окно
Посмотреть список окон


Помощь
Help
Contents
Index Shift+F1
Topic search Ctrl+F1
Previous topic Alt+F1
Using help
Files...
Compiler directives
Procedures and functions
Reserved words
Standard units
Borland Pascal Language
Error messages
About...
Содержание
Поиск в индексе
Настройка файлов помощи
Процедуры и функции
Стандартные модули
Сообщения об ошибках
О программе
Индекс
Предыдущий пункт помощи
Директивы компилятора
Зарезервированные слова
Справка по языку Pascal

Приложение И
Стандартная кодировка символов MS-DOS

Коды управляющих символов (0–31)

Код	Обозначение символа	Назначение или выполняемое действие
0	NUL	Пустой символ
1	SOH	Начало заголовка
2	STX	Начало текста
3	ETX	Конец текста
4	EOT	Конец передачи
5	ENQ	Запрос
6	ACK	Подтверждение
7	BEL	Сигнал (звонок)
8	BS	Забой (шаг назад)
9	HT	Горизонтальная табуляция
10	LF	Перевод строки
11	VT	Вертикальная табуляция
12	FF	Новая страница (прогон формата)
13	CR	Возврат каретки
14	SO	Выключить сдвиг
15	SI	Включить сдвиг
16	DLE	Ключ связи данных
17	DC1	Управление устройством
18	DC2	Управление устройством
19	DC3	Управление устройством
20	DC4	Управление устройством
21	NAK	Отрицательное подтверждение
22	SYN	Синхронизация
23	ETB	Конец передаваемого блока
24	CAN	Отказ
25	EM	Конец среды
26	SUB	Замена
27	ESC	Ключ
28	FS	Разделитель файлов
29	GS	Разделитель группы
30	RS	Разделитель записей
31	US	Разделитель модулей

Символы с кодами 32–127

Код	Символ	Код	Символ	Код	Символ	Код	Символ
32	Пробел	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	Del

Символы с кодами 128–255 (Кодовая таблица 866 – MS-DOS)

Код	Символ	Код	Символ	Код	Символ	Код	Символ
128	А	160	а	192	└	224	р
129	Б	161	б	193	┴	225	с
130	В	162	в	194	┬	226	т
131	Г	163	г	195	├	227	у
132	Д	164	д	196	─	228	ф
133	Е	165	е	197	┼	229	х
134	Ж	166	ж	198	╞	230	ц
135	З	167	з	199	╟	231	ч
136	И	168	и	200	╚	232	ш
137	Й	169	й	201	╔	233	щ
138	К	170	к	202	╩	234	ъ
139	Л	171	л	203	╦	235	ы
140	М	172	м	204	╠	236	ь
141	Н	173	н	205	═	237	э
142	О	174	о	206	╬	238	ю
143	П	175	п	207	╧	239	я
144	Р	176	░	208	╨	240	Ё
145	С	177	▒	209	╤	241	ё
146	Т	178	▓	210	╥	242	Є
147	У	179	│	211	╙	243	є
148	Ф	180	┤	212	╘	244	Ї
149	Х	181	╡	213	╒	245	ї
150	Ц	182	╢	214	╓	246	Ў
151	Ч	183	╖	215	╫	247	ў
152	Ш	184	╕	216	╪	248	°
153	Щ	185	╣	217	┘	249	∙
154	Ъ	186	║	218	┌	250	·
155	Ы	187	╗	219	█	251	√
156	Ь	188	╝	220	▄	252	№
157	Э	189	╜	221	▌	253	¤
158	Ю	190	╛	222	▐	254	■
159	Я	191	┐	223	▀	255	

Примечание. Символы с кодами 128-255 предназначены для национальных алфавитов и символов псевдографики.

Приложение К
Некоторые встроенные процедуры и функции

Работа с текстовыми файлами

Assign(F, Name)	Назначает файловой переменной **F** имя файла **Name.**
Reset(F)	Открывает файл **F** для чтения устанавливает позицию чтения в начало файла.
Rewrite(F)	Открывает файл для записи; в существующем файле старое содержимое стирается.
Write(F, …)	Записывает данные в файл на текущей строке.
Writeln(F, …)	Записывает данные в файл на текущей строке и добавляет признак конца строки.
Eoln(F), SeekEoln(F)	Возвращает **TRUE**, если позиция чтения находится в конце строки (**SeekEoln** игнорирует пробелы и табуляции).
Eof(F), SeekEof(F)	Возвращает **TRUE**, если позиция чтения находится в конце файла (**SeekEof** игнорирует пустые строки).
Read(F, …)	Читает данные из файла, пропуская признаки конца строки.
Readln(F, …)	Читает данные в текущей строке и переводит позицию чтения в начало следующей строки.
Close(F)	Закрывает ранее открытый для чтения или записи файл **F.**

Генерация случайных чисел

Random(N)	Возвращает псевдослучайное целое число в диапазоне от 0 до **N-1**
Randomize	Изменяет псевдослучайную последовательность функции **Random** при каждом запуске программы.

Обработка строк

Length(S)	Возвращает длину строки **S**
Pos(S1, S2)	Возвращает позицию строки **S1** в строке **S2**
Insert(S1, S2, Index)	Вставляет строку **S1** в строку **S2** начиная с позиции **Index**
Copy(S, Index, Count)	Возвращает часть строки **S**, начиная с позиции **Index** длиной **Count** символов
Delete(S, Index, Count)	Удаляет часть строки **S**, начиная с позиции **Index** длиной **Count** символов
UpCase(Ch)	Переводит латинские буквы в верхний регистр

Действия с переменными любого типа

SizeOf(…)	Возвращает объем памяти, занимаемый переменной (или типом данных)
FillChar(X, Size, Val)	Заполняет переменную **X** значением **Val**. Параметр **Size** определяет количество заполняемых байтов.

Действия с переменными порядковых типов

Ord(X)	Возвращает код символа **X**
Chr(N)	Возвращает символ с кодом **N**
Succ(N)	Возвращает следующее значение порядкового типа
Pred(N)	Возвращает предыдущее значение порядкового типа
Inc(N)	Увеличивает число N на единицу
Dec(N)	Уменьшает число N на единицу

Усечение и округление действительных переменных

Trunc(R)	Возвращает целую часть действительного числа **R**
Round(R)	Возвращает округленное действительное число **R**

Динамические переменные и куча

New(P)	Создает новую динамическую переменную **P**
Dispose(P)	Уничтожает динамическую переменную **P**
MemAvail	Возвращает общий объем свободной памяти в куче
MaxAvail	Возвращает размер наибольшего свободного блока памяти в куче

Приложение Л
Перечень программ

Глава	Файл	Содержание программы
5	P_05_1	Вывод сообщения «Привет!»
7	P_07_1	Вывод приветствие на нескольких строках
8	P_08_1	Приветствие по имени
9	P_09_1	Приветствие по имени и фамилии
	P_09_2	Приветствие по имени и фамилии (второй вариант)
	P_09_3	Вывод «СПАРТАК – чемпион!»
10	P_10_1	Проверка пароля, версия 1
	P_10_2	Проверка пароля, версия 2
11	P_11_1	Проверка пароля, версия 3
12	P_12_1	Проверка пароля в цикле, версия 1
	P_12_2	Проверка пароля в цикле, версия 2
	P_12_2	Проверка пароля в цикле, версия 3
13	P_13_1	Ввод данных со спутника (булевы переменные)
14	P_14_1	Экзамен по таблице умножения, версия 1
15	P_15_1	Случайные числа, версия 1
	P_15_2	Случайные числа, версия 2
	P_15_3	Экзамен по таблице умножения, версия 2
16	P_16_1	Вопрос-ответ, версия 1
17	P_17_1	Экзамен по таблице умножения, версия 3
18	P_18_1	Распечатка строки по вертикали, версия 1
	P_18_2	Распечатка строки по вертикали, версия 2
19	P_19_1	Процедура без параметров
	P_19_2	Процедура с параметром
20	P_20_1	Замена символов в строке (заготовка)
	P_20_2	Процедура замены символов в строке
22	P_22_1	Процедура обмена
23	P_23_1	Подсчет символов в строке

Глава	Файл	Содержание программы
23	P_23_2	Замена символов в строке
24	P_24_1	Криптография (шифрование строки)
25	P_25_1	Распечатка текстового файла, версия 1
	P_25_2	Распечатка текстового файла, версия 2
26	P_26_1	Запись в текстовый файл
	P_26_2	Шифрование файла
27	P_27_1	Проверка наличия заданного файла
29	P_29_1	Полицейская база данных, версия 1
	P_29_2	Полицейская база данных, версия 2
30	P_30_1	Обработка классного журнала (первый этап)
31	P_31_1	Обработка классного журнала (второй этап)
37	P_37_1	Вывод множества в текстовый файл
	P_37_2	Ввод и вывод множеств
	P_37_3	Задача о кружках, версия 1
	P_37_4	Задача о кружках, версия 2
38	P_38_1	Задача о кружках, версия 3
	P_38_2	Подвиг контрразведчика
	P_38_3	Поиск стран-соседей
	P_38_4	Решето Эратосфена
40	P_40_1	Программа «вопрос-ответ», версия 2 (с массивом)
	P_40_2	Полицейская база данных, версия 3 (с массивом)
	P_40_3	Подсчет букв в файле
41	P_41_1	«Пузырьковая» сортировка массива чисел
	P_41_2	Пиратская дележка по справедливости
	P_41_3	Футбольный чемпионат, версия 1
42	P_42_1	Сравнение методов поиска
43	P_43_1	«Фермерская» сортировка
	P_43_2	«Быстрая» сортировка
	P_43_3	Сравнение методов сортировки

Глава	Файл	Содержание программы
44	P_44_1	Структура строки
	P_44_2	Поиск в строке слова «PASCAL»
	P_44_3	Замена в строке слова «Pascal»
45	P_45_1	Запись в танцевальный кружок, версия 1
	P_45_2	Моделирование сортировочной станции
46	P_46_1	Печать сверхбольшого числа
	P_46_2	Сложение сверхбольших чисел
47	P_47_1	Преобразование из десятичной системы
	P_47_2	Преобразование в десятичную систему
48	P_48_3	Логические операции с числами
49	P_49_1	Подсчет пересечений границ между странами
	P_49_2	Реклама «крестики-нолики»
50	P_50_1	Футбольный чемпионат, версия 2
	P_50_2	Футбольный чемпионат, версия 3
51	P_51_1	Принцип действия указателей
	P_51_2	Действия с указателями, размеры указателей
53	P_53_1	Ввод и вывод массив указателей
	P_53_2	Сортировка массива указателей
54	P_54_1	Ввод и вывод списка
	P_54_2	Поиск в списке
	P_54_3	Сортированный список
	P_54_4	Поиск в сортированном списке
55	P_55_1	Частотный анализатор текста
56	P_56_1	Перестановка строк файла
	P_56_2	Запись в танцевальный кружок, версия 2
57	P_57_1	Ввод и вывод графа
58	P_58_1	Обход графа в ширину
	P_58_2	Поиск кратчайшего пути в графе
59	P_59_1	Перестановка строк файла (используется модуль MyLibr)

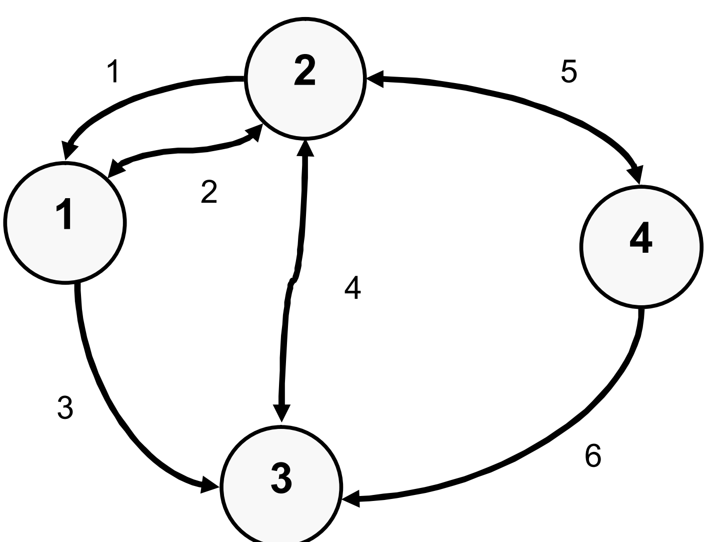
Глава	Файл	Содержание программы
59	MyLibr	Библиотечный модуль к программе P_59_1
61	P_61_1	Демонстрация работы Turbo Vision
	P_61_2	Программа с объектом типа «человек»
	P_61_3	Демонстрация наследования и полиморфизма

Приложение М
Пример олимпиадной задачи

Представлена одна из задач XVII районной (городской) олимпиады по информатике Московской области 2004 г.

Дано

Трамвайная сеть города состоит из ***N*** трамвайных остановок, пронумерованных числами от 1 до ***N***. Остановки соединяются друг с другом ***M*** перегонами, пронумерованными числами от 1 до ***M***. На трамвайных остановках есть стрелки для перехода трамвая с любого ведущего к остановке перегона на любой другой перегон, ведущий от нее. Все перегоны имеют одинаковую длину, но принадлежат к двум типам: односторонние и двухсторонние. По односторонним перегонам трамваи могут двигаться только в одном направлении; по двусторонним — в обоих, но вдвое медленнее, чем по односторонним.



Требуется

По заданной схеме трамвайной сети города найти кратчайший по времени путь между двумя заданными остановками, при условии, что трамваи никогда не мешают друг другу (в городе один трамвай). Входные данные гарантируют, что путь между остановками всегда существует.

Входные данные

В первой строке входного файла приведено количество остановочных пунктов ***N*** (2≤ ***N***≤ 100) и число перегонов ***M*** (1 ≤ ***M*** ≤ 30000). Далее идут ***M*** строк с описаниями перегонов по одному описанию в строке. Каждое описание состоит из четырех чисел, разделенных пробелом: номера перегона; двух номеров остановок, которые соединяет данный перегон; тип перегона (1 — если перегон односторонний и 2 — если двусторонний). Если перегон односторонний, то

движение трамваев по нему разрешается от первого остановочного пункта в описании ко второму. Далее следует строка с двумя номерами остановок, между которыми следует найти кратчайший по времени путь (от исходной остановки к конечной)

Выходные данные

В выходной файл «output.txt» следует вывести список номеров остановочных пунктов и перегонов между ними в порядке их прохождения трамваем. В случае нескольких возможных правильных ответов вывести любой из них.

Контрольный пример

Входные данные	**Вывод**
4 6 **1 2 1 1** **2 2 1 2** **3 1 3 1** **5 2 4 2** **4 2 3 2** **6 4 3 1** **1 4**	**1** **2** **2** **5** **4**

Библиография

1. Алексеев А. В. ***Олимпиады школьников по информатике. Задачи и решения.*** — Красноярск: Красноярское книжное издательство, 1995.
2. Андреева Е. В., Фалина И. Н. ***Информатика: Системы счисления и компьютерная арифметика.*** — М.: Лаборатория Базовых Знаний, 1999.
3. Ахо А., Хопкрофт Дж., Ульман Дж. ***Построение и анализ вычислительных алгоритмов***. — М.: Мир, 1979.
4. Бабушкина И. А., Бушмелева Н. А., Окулов С. М., Черных С.Ю. ***Конспекты занятий по информатике (практикум по Паскалю).*** — Киров: Изд-во ВятГПУ, 1997.
5. Бадин Н. М., Волченков С. Г., Дашниц Н. Л., Корнилов П. А. ***Ярославские олимпиады по информатике***. — Ярославль: Изд-во ЯрГУ, 1995.
6. Беров В. И., Лапунов А. В., Матюхин В. А., Пономарев А. А. ***Особенности национальных задач по информатике***. — Киров: Триада-С, 2000.
7. Брудно А. Л., Каплан Л. И. ***Олимпиады по программированию для школьников*** / Под ред. Б. И. Наумова /. — М.: Наука, 1985. 96 с.
8. Брудно А. П., Каплан Л. И. ***Московские олимпиады по программированию.*** — М.: Наука, 1990.
9. Вирт Н. ***Алгоритмы+структуры данных=Программы.*** — М.: Наука, 1989.
10. Дагене В. А., Григас Г. К., Аугутис К. Ф. ***100 задач по программированию.*** — М.: Просвещение, 1993.
11. Долинский М. С. ***Решение сложных и олимпиадных задач по программированию: Учебное пособие***. — СПб.: Питер, 2006. — 366 с
12. Емеличев В. А., Мельников О. И., Сарванов В. И., Тышкевич Р. И. ***Лекции по теории графов.*** — М.: Наука, 1990.
13. Епанешников А., Епанешников В.. ***Программирование в среде Turbo Pasca 7.0.*** — М.: «Диалог-МИФИ», 1995.
14. Йодан Э. ***Структурное проектирование и конструирование программ.*** — М.: Мир, 1979.
15. Кирюхин В. М., Лапунов А. В., Окулов С.М. ***Задачи по информатике. Международные олимпиады 1989-1996.*** — М.: «ABF», 1996.
16. Кнут Д. Искусство программирования. Т. 1-3, ***Получисленные алгоритмы. Сортировка и поиск.*** — М.: Издательский дом «Вильямс», 2000.
17. Кормен Т., Лейзерстон Ч., Ривест Р. ***Алгоритмы: Построение и анализ.*** — М.: МЦНМО, 2001.
18. Кристофидес Н. Теория графов. ***Алгоритмический подход.*** — М.: Мир, 1978.
19. Лапунов А. В., Окулов С. М. ***Задачи международных олимпиад по информатике.*** — Киров, Изд-во ВятГПУ, 1993.

20. Меньшиков Ф. В. ***Олимпиадные задачи по программированию*** (+CD). — СПб.: Питер, 2006.

21. Овсянников А., Овсянникова Т., Марченко А., Прохоров Р. ***Избранные задачи олимпиад по информатике.*** — М.: Тровант, 1997.

22. Окулов С. М. ***Задачи кировских олимпиад по информатике.*** — Киров, Изд-во ВятГПУ, 1993.

23. Окулов С. М. ***Конспекты занятий по информатике (алгоритмы на графах): Учебное пособие.*** — Киров, Изд-во ВятГПУ, 1996.

24. Окулов С. М. ***Основы программирования.*** — М.: Лаборатория Базовых Знаний, 2001.

25. Окулов С. М., Пестов А. А. ***100 задач по информатике.*** — Киров, Изд-во ВятГПУ, 2000.

26. Окулов С. М., Пестов А. А., Пестов О. А. ***Информатика в задачах.*** — Киров, Изд-во ВятГПУ, 1998.

27. Поляков Д.Б., Круглов И.Ю. ***Программирование в среде Тубо Паскаль (версия 5.5)*** — М.: Издательство МАИ 1992.

28. Препарата Ф., Шеймос М. ***Вычислительная геометрия. Введение.*** — М.: Мир, 1989.

29. Седжвик Роберт. ***Фундаментальные алгоритмы на С++. Анализ. Структуры данных. Сортировка. Поиск.*** — Москва, С-Петербург, Киев: Диасофт, 2003.

30. Скиена С. С, Ревилла М. А. ***Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям*** Пер. с англ. - М: КУДИЦ-ОБРАЗ, 2005. — 416 с.

31. Уэзерелл Ч. ***Этюды для программистов.*** — М.: Мир, 1982. 288 с.

32. Фаронов В.В. ***Турбо Паскаль 7.0. Начальный курс: Учебное пособие*** — М.: КНОРУС, 2007, 576 с.

33. Фаронов В.В. Турбо ***Паскаль 7.0. Практика программирования. Учебное пособие*** — М.: КНОРУС, 2007, 416 с.

34. Федоров А. ***Borland Pascal: практическое использование Turbo Vision 2.0*** — Киев: Диалектика 1993.

35. Федоров А. ***Особенности программирования на Borland Pascal*** — Киев: Диалектика 1994.

36. Хаггарти Р. ***Дискретная математика для программистов*** — М., Техносфера, 2003

37. Шень А. ***Программирование: теоремы и задачи.*** — М., МЦНМО, 1995.

38. Шпак Ю.А. ***Turbo Pascal 7.0 на примерах***. — К., «Юниор», 2003.