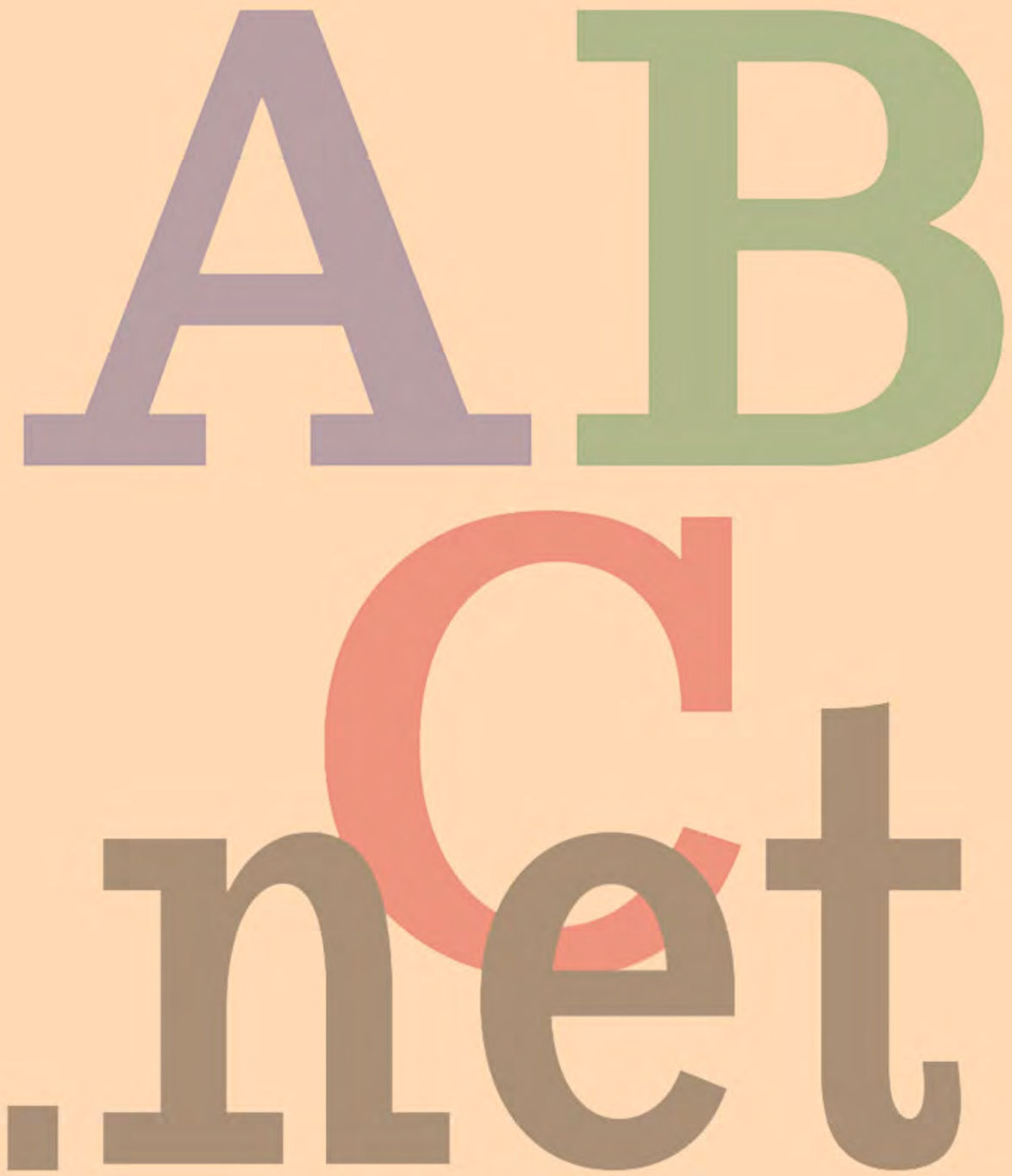
Александр Осипов

PascalABC.NET: Введение в современное программирование



Лицензия

Александр Осипов

PascalABC.NET:

Введение в современное программирование

Ростов-на-Дону, 2019

УДК 004.043(07)
ББК 32.973.2-018я7
О741

Осипов Александр Викторович

PascalABC.NET: Введение в современное программирование. – Ростов-на-Дону, 2019 – 572с. : ил.

Книга является первым изданием, содержащим полное описание языка программирования PascalABC.NET (версия 3.5), разработанного в Южном федеральном университете и завоевавшего широкое признание в образовательных учреждениях. Помимо описания языка, рассматриваются особенности использования его конструкций, даются рекомендации по выбору языковых средств в зависимости от решаемой задачи. Рассмотрение конструкций языка сопровождается примерами. Приводимые в конце каждой части книги задания способствуют лучшему усвоению материала. Тексты программ, с которыми предлагается поработать самостоятельно, размещены в прилагаемом к книге архивном файле. Для широкого круга читателей, желающих изучить язык PascalABC.NET и освоить приемы современного программирования.

Предисловие разработчика PascalABC.NET

Перед вами – первая большая книга по языку программирования PascalABC.NET. Это большой труд: написание данной книги заняло у автора более года. Её название действительно отражает суть – книга учит современному программированию. Это программирование с использованием современных языковых конструкций, современной системы типов и мощного набора библиотечных средств.

Идея создания PascalABC.NET возникла 15 лет назад, когда мы осознали, что нет хорошего современного языка программирования для обучения школьников. В качестве основного языка программирования для обучения в то время мы использовали язык Паскаль. Но язык Delphi Pascal в некоторый момент остановился в развитии, а его бесплатный аналог – язык Free Pascal – в основном просто копировал своего «старшего брата», введя лишь небольшое количество собственных расширений. Мы ощущали также нехватку простой и одновременно мощной среды программирования, хорошо русифицированной и с интерактивными подсказками по коду.

Именно в это время активно развивалась тогда еще молодая платформа Microsoft .NET, предлагавшая отличный набор библиотек и многочисленные современные языковые конструкции. Поэтому неудивительно, что команда разработчиков PascalABC.NET выбрала в качестве платформы реализации именно Microsoft .NET.

Синтаксически язык PascalABC.NET близок к языку Delphi (совместимость – 90-95%), однако включает множество конструкций языка C# , такие как обобщения, лямбда-выражения, кортежи, интерфейсы, сопоставление с образцом. Кроме того, PascalABC.NET вводит ряд собственных расширений языка Паскаль, нацеленных на написание компактных интуитивно понятных программ.

Язык PascalABC.NET активно совершенствуется и в настоящее время по мощности языка существенно опережает язык Delphi Pascal, который практически не развивается как язык программирования вот уже более 10 лет.

Современный PascalABC.NET нацелен на создание небольших и средних программных проектов, выигрывая у своего «старшего брата» – языка C# – по лаконичности записи программ и компактности инфраструктуры, создаваемой в процессе компиляции. Так, при написании простой консольной программы на C# в Visual Studio 2017 создаётся 11 папок и 10 файлов, а в PascalABC.NET – 1 файл.

Язык PascalABC.NET также многое взял от языка Python. Это прежде всего легковесные языковые и библиотечные средства, обеспечивающие компактность и интуитивную понятность программ. Но в отличие от языка Python, язык PascalABC.NET имеет статическую типизацию, что позволяет фиксировать множество ошибок на этапе компиляции и делает программу более безопасной.

В сфере обучения современному программированию язык PascalABC.NET имеет отличные показатели. Прежде всего, это новые примитивы программирования, такие как кортежи, срезы, лямбда-выражения и последовательности, позволяющие писать выразительные программы и концентрироваться на сути задачи, не отвлекаясь на ненужные технические сложности. Это – библиотечные средства, которые предоставляет платформа Microsoft .NET, а также модули стандартной библиотеки PascalABC.NET. Кроме того, PascalABC.NET – это простая и мощная среда разработки программ. В силу своей популярности у школьников PascalABC.NET принят как один из языков на школьных олимпиадах по программированию.

Разноуровневость конструкций языка позволяет выбирать различные траектории обучения современному программированию. В Южном федеральном университете накоплен богатый опыт обучения на основе PascalABC.NET. Так, детская компьютерная школа мехмата ЮФУ (сайт http://компьютернаяшкола.рф) каждый год выпускает более 300 учащихся по направлениям «Юный программист» (1,2,3 ступени), где PascalABC.NET является первым и основным языком программирования (в сочетании с Python и C#).

Язык PascalABC.NET взят за основу в курсе «Основы программирования» у студентов 1 курса мехмата ЮФУ по направлению «Фундаментальная информатика и информационные технологии». Теоретическая и практическая база, полученные при изучении этого курса, позволяют студентам эффективно изучать впоследствии такие языки как C#, C++, Java, Haskell, Python, Ruby и многие другие.

Возвращаясь к книге, хотелось бы отметить поистине титанический труд автора и блестящее изложение, изобилующее цитатами, историческими экскурсами, кропотливо подобранные термины и великолепное оформление. Книгу можно рассматривать и как справочник, и как учебник по многочисленным возможностям языка и библиотек. Но, наверное, ближе всего данная книга к некоторому авторскому познанию языка программирования PascalABC.NET. Её очень интересно читать даже нам – разработчикам языка, а также преподавателям, которые учат программировать, используя этот язык.

Следует отметить также, что автор писал эту книгу в тесном взаимодействии с разработчиками, в результате чего были устранены некоторые ошибки реализации библиотечных функций, добавлены некоторые возможности языка, а также ряд библиотечных функций был стандартизирован и усовершенствован.

Долгое время говорили, что основным недостатком PascalABC.NET является отсутствие по нему литературы. Теперь это не так. И хочется пожелать всем читателям множества приятных минут от общения с этой замечательной книгой.

С наилучшими пожеланиями,
руководитель проекта PascalABC.NET
Михалкович Станислав Станиславович,
мехмат Южного федерального университета

Предисловие рецензента

В последнее время, когда в нашем вузе полностью отказались от Delphi и перешли на PascalABC.NET, стало очень трудно рекомендовать студентам и школьникам литературу по языку программирования. Книги по Delphi устаревают, а по Turbo Pascal — устарели уже давно. Поэтому я очень обрадовался, когда узнал, что есть подвижник, взявшийся описать PascalABC.NET. Книга получилась объёмной. Вряд ли её можно назвать учебником: не только из-за количества вмещённых сведений, но и потому ещё, что плотность информации зачастую очень велика. В некоторых случаях автор говорит с читателем, который только начинает знакомство с программированием, а иногда — с довольно опытным программистом, который занялся более глубоким изучением вопроса. В то же время читатель держит в руках не справочник, который повторяет встроенную подсказку, добавляя к ней побольше текста и примеров. Мы имеем дело с чем-то средним. Наверное, больше всего подходит термин «учебное пособие», что сразу намечает круг читателей, которым могла бы пригодиться книга. Это может быть любознательный школьник старших классов или студент, который изучает программирование и хочет более глубокого погружения в свежий материал. Или учитель информатики, желающий отказаться от стандартов программирования девяностых годов и стремящийся шагать в ногу со временем. Для каждого из них книга может быть замечательным подспорьем в освоении многогранного PascalABC.NET.

Хотелось бы сказать несколько слов о термине «современное программирование», вынесенном в заголовок книги. Сегодня в школе сложилась система, ориентированная на обучение старому стилю программирования, по крайней мере если говорить о программировании на Паскале. Прямым доказательством этому являются задания на Паскале, встречающиеся в ЕГЭ: они написаны на языке тридцатилетней давности, потому что именно он изучается в школах. В таком окружении тексты на Python выглядят более современно. Так можно ли писать на Паскале современный код? Автор каждой строкой кода убеждает нас, что да, можно. И нужно, поскольку современный стиль позволяет сократить код, сделать его более читаемым, более гибким, переносимым и масштабируемым. Современный стиль означает, что программирование ведётся на более высоком уровне по сравнению со старыми языками. Этому тоже можно научиться, занимаясь по книге.

В заключение заметим, что такое удивительное явление, как PascalABC.NET, давно требует столь же масштабной информационной поддержки. Работу А. В. Осипова можно считать на сегодня самой смелой и успешной попыткой охватить как можно бóльшую часть как самого языка, так и прилегающих к нему «территорий». Хочется пожелать автору книги и разработчикам языка дальнейших успехов в столь нелёгком и, к сожалению, недостаточно оценённом труде, а также сил для реализации всего задуманного!

Дженжер В. О., к. ф.-м. н., доцент, заведующий кафедрой информатики, физики и МПИФ Оренбургского государственного педагогического университета

От автора

Во всякой книге предисловие есть первая и вместе с тем последняя вещь; оно или служит объяснением цели сочинения, или оправданием и ответом на критики. Но обыкновенно читателям дела нет до нравственной цели и до журнальных нападок, и потому они не читают предисловий.

М.Ю.Лермонтов. «Герой нашего времени»

Идея написать эту книгу появилась в результате довольно продолжительных дискуссий на Интернет-форуме Южного федерального университета, посвященных продвижению PascalABC.NET. Вызывает удивление, что при наличии такой мощной современной системы программирования на базе языка Pascal, в учебных заведениях продолжают преподаваться устаревшие Free Pascal, Delphi и даже Turbo Pascal. Причем, достаточно часто входные языки Turbo Pascal и Free Pascal изучаются на компьютерах с установленным PascalABC.NET! Откуда же такое стойкое неприятие развитого языка, позволяющего писать программы короче и эффективнее, упростить преподавание и заложить базу для перехода к современным промышленным языкам? Мы долго обсуждали на форуме сложившуюся ситуацию и пришли к мнению, что причин много. В частности, PascalABC.NET нигде достаточно подробно не был описан. Существуют только официальная справка и небольшое количество презентаций по его возможностям. Есть некоторое количество книг, но все они охватывают лишь небольшие подмножества языка, поскольку написаны в период, когда его нынешняя редакция переживала свое рождение. Неудивительно, что преподаватели лишены возможности в достаточное объеме ознакомиться с PascalABC.NET – как же в таких условиях строить обучение на его базе? Практически отсутствуют рекомендации по использованию многих возможностей и особенностей языка. При попытке самостоятельно освоить PascalABC.NET неожиданно обнаруживается потребность в знании множества современных концепций программирования. Желательно наличие опыта работы с каким-либо языком, включенным в Microsoft.NET, умение использовать LINQ. А еще - знать начала функционального программирования. И мы пришли к пониманию, что нужна книга, достаточно полно описывающая возможности PascalABC.NET и особенности программирования на нем. Создание электронного варианта такой книги заняло у автора почти полтора года. Насколько она окажется полезной – судить вам.

Я выражаю большую признательность всем участникам форума, подтолкнувшим меня к идее написать эту книгу. Считаю своим долгом поблагодарить Андрея Пидорченко, высказавшего множество критических замечаний на раннем этапе работы. Особые слова благодарности хочу сказать Станиславу Михалковичу и Вадиму Дженжеру, которые потратили много времени и сил, проявили героическое терпение и оказали неоценимую помощь в работе над книгой. Без них, скорее всего, ее бы не было.

А.Осипов

Оглавление

Введение 27
Немного истории 27
Зачем изучать PascalABC.NET? 29
О требовании эффективности программ 29
Чего вы здесь не найдете 31
Как пользоваться этой книгой 31
Парадигмы программирования 32
Наша первая программа 33
Понятие о типах данных 35
Часть 1 Арифметика целых чисел 37
1.1 Целые типы в PascalABC.NET 38
1.2 Константы 39
1.3 Переменные 42
1.4 Арифметические выражения 44
1.5 Оператор присваивания 50
1.6 «Сюрпризы» целочисленной арифметики 51
1.7 Явное приведение типа 52
1.8 Ввод целочисленных данных 54
1.9 Инкремент и декремент 56
1.10 Для самостоятельного решения 57
Часть 2 Вещественные числа 59
2.1 Типы вещественных чисел 60
2.2 Литералы вещественного типа 61
2.3 Арифметические выражения вещественного типа 61
2.4 Точность машинной арифметики 64
2.5 Вывод вещественных значений 65
2.6 Ввод вещественных значений 66
2.7 Обращение к системным библиотекам .NET 66
2.8 Для самостоятельного решения 66

Часть 3 Логика в программе 69
3.1 Логический тип данных 70
3.2 Логические выражения....70
3.3 Условный оператор 74
3.4 Условная операция....76
3.5 Оператор выбора 79
3.6 Циклы 81
3.7 Для самостоятельного решения 97
Часть 4 Подпрограммы....99
4.1 Общие сведения....100
4.2 Процедуры 103
4.3 Функции....108
4.4 Рекурсия 111
4.5 Опережающее объявление подпрограмм....111
4.6 Перегрузка имен подпрограмм 112
4.7 Подпрограмма в качестве параметра 113
4.8 Процедурная переменная 114
4.9 Лямбда-выражения....116
4.10 Первое понятие класса 120
4.11 Для самостоятельного решения 122
Часть 5 Последовательности....123
5.1 Последовательности sequence....125
5.2 Множества set of T 160
5.3 Динамические массивы array of T 166
5.4 Статические массивы array [] of T 193
5.5 Для самостоятельного решения 196
Часть 6 Символы и строки 199
6.1 Символьный тип данных 201
6.2 Строковый тип данных....212
6.3 Для самостоятельного решения 250

Часть 7 Типы данных....253
7.1 Записи....255
7.2 Перечислимый тип....263
7.3 Диапазонный тип....265
7.4 Эквивалентность и совместимость типов....265
7.5 Пример: работа с таблицей....268
7.6 Обобщенный тип....273
7.7 Указатели....275
7.8 Тип данных BigInteger....278
7.9 Тип данных decimal....280
7.10 Тип данных DateTime....281
7.11 Для самостоятельного решения....283
Часть 8 Многомерные массивы....285
8.1 Динамические двухмерные массивы (матрицы)....286
8.2 Статические двухмерные массивы....298
8.3 Массивы размерности выше двух....299
8.4 Примеры решения задач с матрицами....299
8.5 Для самостоятельного решения....302
Часть 9 Файлы....305
9.1 Файловый тип данных....307
9.2 Связь программы с файлом....311
9.3 Открытие и закрытие файлов....312
9.4 Запись данных в файл....321
9.5 Чтение данных из файла....331
9.6 Операции с файловым указателем....343
9.7 Прочие операции с файлами....345
9.8 Рекомендации по работе с файлами....346
9.9 Для самостоятельного решения....346

Часть 10 Стандартные коллекции....349
10.1 Линейные списки....351
10.2 Двусвязный список LinkedList....362
10.3 Список List....370
10.4 Множество HashSet....376
10.5 Словарь Dictionary....385
10.6 Коллекции с упорядоченностью элементов....389
10.7 Для самостоятельного решения....396
Часть 11 Модули и библиотеки....397
11.1 Модули....399
11.2 Библиотеки dll....408
11.3 Документирующие комментарии....410
11.4 Пример программы....411
11.5 Стандартные модули в PascalABC.NET....421
11.6 Для самостоятельного решения....424
Часть 12 Обработка ошибок в программе....427
12.1 Предотвращение исключений....429
12.2 Обработка исключений....434
Часть 13 Объектно-ориентированное программирование....445
13.1 Базовые понятия....446
13.2 Наследование....466
13.3 Полиморфизм....472
13.4 Операции as и is....482
13.5 Сопоставление с образцом....488
13.6 Деконструкторы....490
13.7 Анонимные классы....492
13.8 Методы расширения....494
13.9 Интерфейсы....498
13.10 Статический класс System.Console....511
13.11 Класс – обработчик исключений....513
13.12 Для самостоятельного решения....518

Часть 14 Ответы и решения.....521
14.1 К части 1 522
14.2 К части 2 523
14.3 К части 3 523
14.4 К части 4 527
14.5 К части 5 531
14.6 К части 6 533
14.7 К части 7 534
14.8 К части 8 538
14.9 К части 9 540
14.10 К части 10 544
14.11 К части 11 547
14.12 К части 13 549
Приложения 557
Предметный указатель.....565

Содержание

Введение 27

Немного истории 27
Зачем изучать PascalABC.NET? 29
О требовании эффективности программ 29
Чего вы здесь не найдете 31
Как пользоваться этой книгой 31
Парадигмы программирования 32
Наша первая программа 33
Понятие о типах данных 35

Часть 1 Арифметика целых чисел 37

1.1 Целые типы в PascalABC.NET 38
1.2 Константы 39
1.3 Переменные 42
1.4 Арифметические выражения 44
1.4.1 Арифметические операции 44
1.4.2 Приоритет арифметических операций 45
1.4.3 Игры с унарным плюсом и минусом 46
1.4.4 Стандартные целочисленные функции 47
1.5 Оператор присваивания 50
1.6 «Сюрпризы» целочисленной арифметики 51
1.7 Явное приведение типа 52
1.8 Ввод целочисленных данных 54
1.9 Инкремент и декремент 56
1.10 Для самостоятельного решения 57

Часть 2 Вещественные числа 59

2.1 Типы вещественных чисел 60
2.2 Литералы вещественного типа 61
2.3 Арифметические выражения вещественного типа 61
2.3.1 Деление и возведение в степень 61
2.3.2 Приведение типа 62
2.3.3 Об арифметическом переполнении 63
2.3.4 Некоторые математические функции 64

2.4 Точность машинной арифметики ... 64
2.5 Вывод вещественных значений ... 65
2.6 Ввод вещественных значений ... 66
2.7 Обращение к системным библиотекам .NET ... 66
2.8 Для самостоятельного решения ... 66

Часть 3 Логика в программе ... 69

3.1 Логический тип данных ... 70
3.2 Логические выражения ... 70
3.2.1 Операции отношения ... 71
3.2.2 Логические операции ... 71
3.2.3 О «короткой схеме» ... 73
3.3 Условный оператор ... 74
3.4 Условная операция ... 76
3.5 Оператор выбора ... 79
3.6 Циклы ... 81
3.6.1 Цикл с заданным числом повторений (loop) ... 82
3.6.2 Цикл с параметром (for) ... 83
3.6.3 Цикл с предусловием (while) ... 86
3.6.4 Цикл с постусловием (repeat) ... 88
3.6.5 Изменение нормального хода выполнения цикла ... 89
3.6.5.1 Оператор break ... 89
3.6.5.2 Оператор continue ... 90
3.6.5.3 Оператор exit ... 91
3.6.6 О вложенных циклах ... 91
3.6.7 Задача табуляции функции ... 92
3.6.7.1 Использование цикла for ... 92
3.6.7.2 Использование цикла while ... 94
3.6.7.3 Использование цикла repeat ... 95
3.7 Для самостоятельного решения ... 97

Часть 4 Подпрограммы ... 99

4.1 Общие сведения ... 100
4.1.1 Параметры в подпрограммах ... 101
4.1.2 Размерные и ссылочные типы данных ... 101
4.1.3 Ссылка и значение ... 103

4.2 Процедуры ... 103
4.2.1 Передача параметра по значению ... 104
4.2.2 Передача параметра по ссылке ... 105
4.2.3 Значение параметра по умолчанию ... 106
4.2.4 Процедуры без параметров ... 107
4.2.5 Выход из вложенных циклов ... 107
4.3 Функции ... 108
4.4 Рекурсия ... 111
4.5 Опережающее объявление подпрограмм ... 111
4.6 Перегрузка имен подпрограмм ... 112
4.7 Подпрограмма в качестве параметра ... 113
4.8 Процедурная переменная ... 114
4.9 Лямбда-выражения ... 116
4.9.1 Лямбда-функции и лямбда-процедуры ... 118
4.9.2 Захват переменной ... 119
4.10 Первое понятие класса ... 120
4.11 Для самостоятельного решения ... 122

Часть 5 Последовательности ... 123

5.1 Последовательности sequence ... 125
5.1.1 Создание последовательностей ... 126
5.1.1.1 Генераторы ... 127
5.1.1.2 Генераторы, использующие лямбда-выражения ... 128
5.1.1.3 Генераторы бесконечных последовательностей ... 130
5.1.1.4 Ввод элементов последовательности с клавиатуры ... 130
5.1.1.5 Самостоятельное создание последовательностей (yield) ... 133
5.1.2 Операции с последовательностью в целом ... 136
5.1.2.1 Пустые последовательности и метод Count ... 136
5.1.2.2 Операции с последовательностью + и * ... 136
5.1.2.3 Перебор элементов последовательности (цикл foreach) ... 137
5.1.2.4 Перебор элементов последовательности (.ForEach) ... 137
5.1.2.5 Проецирование (метод Select) ... 137
5.1.2.6 Проецирование (метод SelectMany) ... 138
5.1.2.7 Максимум, минимум, сумма, произведение и среднее ... 139
5.1.2.8 Агрегирование элементов (метод Aggregate) ... 141
5.1.2.9 Перестановка элементов в обратном порядке (метод Reverse) ... 142

5.1.3 Выборка подпоследовательности ... 142
5.1.3.1 Выборка элементов от начала последовательности ... 142
5.1.3.2 Пропуск части первых элементов ... 142
5.1.3.3 Пропуск части последних элементов ... 143
5.1.3.4 Выборка элементов от конца последовательности ... 143
5.1.4 Кортежи ... 143
5.1.5 Выборка на основе условия ... 146
5.1.5.1 Выборка неповторяющихся элементов (.Distinct) ... 146
5.1.5.2 Фильтрация (метод Where) ... 146
5.1.5.3 Последовательность на основе пар элементов ... 147
5.1.6 Нумерация элементов последовательности ... 147
5.1.7 Разбиение последовательности ... 148
5.1.7.1 Разбиение на части указанной длины (.Batch) ... 148
5.1.7.2 Разбиение на две части по условию (.Partition) ... 148
5.1.7.3 Разбиение на две части по длине (.SplitAt) ... 149
5.1.7.4 Срез последовательности (.Slice) ... 149
5.1.8 Операции с несколькими последовательностями ... 150
5.1.8.1 Объединение, пересечение и разность ... 150
5.1.8.2 Декартово произведение (.Carthesian) ... 151
5.1.8.3 Чередование элементов (.Interleave) ... 152
5.1.8.4 Соединение элементов (метод Zip) ... 152
5.1.8.5 Создание последовательности кортежей (метод ZipTuple) ... 153
5.1.8.6 Распаковка последовательности кортежей (метод UnZipTuple) ... 153
5.1.9 Сортировка последовательности ... 154
5.1.9.1 Сортировка по возрастанию (.Sorted) ... 154
5.1.9.2 Сортировка по убыванию (.SortedDescending) ... 154
5.1.9.3 Сортировка по возрастанию (.Order) ... 154
5.1.9.4 Сортировка по убыванию (.OrderDescending) ... 155
5.1.9.5 Сортировка по возрастанию ключа (метод OrderBy) ... 155
5.1.9.6 Сортировка по убыванию ключа (метод OrderByDescending) ... 155
5.1.9.7 Вторичная сортировка по возрастанию ключа (метод ThenBy) ... 155
5.1.9.8 Вторичная сортировка по убыванию ключа (метод ThenByDescending) ... 156
5.1.10 Поиск и проверка выполнения условий ... 156
5.1.10.1 Элемент с указанным номером (метод ElementAt) ... 156
5.1.10.2 Элемент с указанным номером (метод ElementAtOrDefault) ... 156
5.1.10.3 Если последовательность пустая… (метод DefaultIfEmpty) ... 157

5.1.10.4 Наличие элемента в последовательности (метод Contains) ... 158
5.1.10.5 Наличие элемента в последовательности (операция in) ... 158
5.1.10.6 Есть ли в последовательности элементы? (метод Any) ... 158
5.1.10.7 Все ли элементы удовлетворяют условию? (метод All) ... 158
5.1.10.8 Сравнение последовательностей (метод SequenceEqual) ... 159
5.1.11 Табуляция функции с помощью последовательности ... 159
5.1.11.1 Использование цикла foreach ... 159
5.1.11.2 Табуляция последовательностью (.Tabulate) ... 160
5.2 Множества set of T ... 160
5.2.1 Создание множества ... 161
5.2.2 Конструктор множества ... 162
5.2.3 Операции над множествами ... 162
5.2.3.1 Перебор элементов множества в цикле foreach ... 162
5.2.3.2 Добавление элемента ко множеству (Include и +=) ... 163
5.2.3.3 Удаление элемента из множества (Exclude и -=) ... 163
5.2.3.4 Проверка наличия элемента во множестве (in) ... 164
5.2.3.5 Объединение множеств (+) ... 164
5.2.3.6 Разность множеств (-) ... 164
5.2.3.7 Пересечение множеств (*) ... 164
5.2.3.8 Равенство множеств (=) ... 165
5.2.3.9 Неравенство множеств (<>) ... 165
5.2.3.10 Строгое вложение (<) ... 165
5.2.3.11 Нестрогое вложение (<=) ... 165
5.2.3.12 Строго содержит (>) ... 165
5.2.3.13 Нестрого содержит (>=) ... 165
5.3 Динамические массивы array of T ... 166
5.3.1 Создание динамических массивов ... 167
5.3.2 Генераторы динамических массивов ... 169
5.3.2.1 Генератор на основе лямбда-выражения (ArrGen) ... 170
5.3.2.2 Ввод элементов массива с клавиатуры ... 171
5.3.3 Операции со всеми элементами массива ... 173
5.3.3.1 Перебор элементов массива в цикле ... 174
5.3.3.2 Перебор элементов массива (.ForEach) ... 175
5.3.3.3 Арифметические операции + и * с массивами ... 175
5.3.3.4 Проецирование (метод Select) ... 175
5.3.3.5 Проецирование (метод SelectMany) ... 176

5.3.3.6 Максимум, минимум, сумма, произведение и среднее ... 177
5.3.3.7 Агрегирование элементов (метод Aggregate) ... 178
5.3.3.8 Перестановка элементов в обратном порядке (метод Reverse) ... 178
5.3.4 Выборки элементов из массива ... 179
5.3.5 Срезы ... 180
5.3.6 О «мягких» срезах ... 183
5.3.7 Операции над массивом ... 183
5.3.7.1 Преобразование элементов массива (.ConvertAll) ... 183
5.3.7.2 Заполнение массива значениями (.Fill) ... 184
5.3.7.3 Замена значения элемента по всему массиву (.Replace) ... 184
5.3.7.4 Перестановка элементов в обратном порядке (Reverse) ... 185
5.3.7.5 Перемешивание элементов случайным образом (Shuffle) ... 185
5.3.7.6 Сортировка элементов массива (Sort) ... 185
5.3.7.7 Преобразование элементов массива (.Transform) ... 186
5.3.7.8 Поиск элементов массива по условию ... 186
5.3.8 Операции с индексами массива ... 187
5.3.8.1 Индекс максимального элемента в массиве ... 187
5.3.8.2 Индекс минимального элемента в массиве ... 188
5.3.8.3 Бинарный поиск индекса элемента (.BinarySearch) ... 188
5.3.8.4 Поиск индекса элемента, удовлетворяющего условию ... 188
5.3.8.5 Поиск индексов элементов, удовлетворяющих условию ... 189
5.3.8.6 Поиск индекса заданного элемента (.IndexOf) ... 190
5.3.9 Продвинутые операции с массивами ... 190
5.3.10 Ключевое слово params в подпрограммах ... 192
5.4 Статические массивы array [] of T ... 193
5.4.1 Создание статических массивов ... 193
5.4.2 Работа со статическими массивами ... 194
5.5 Для самостоятельного решения ... 196

Часть 6 Символы и строки ... 199

6.1 Символьный тип данных ... 201
6.1.1 Анализ символов на принадлежность к группе ... 204
6.1.1.1 Является ли символ буквой? ... 205
6.1.1.2 Является ли символ цифрой? ... 205
6.1.1.3 Является ли символ буквой или цифрой? ... 205
6.1.1.4 Является ли символ пробельным? ... 205
6.1.1.5 Является ли символ знаком препинания? ... 206

6.1.1.6 Принадлежит ли буква к верхнему регистру? ... 206
6.1.1.7 Принадлежит ли буква к нижнему регистру? ... 206
6.1.1.8 Располагается ли символ в указанном интервале? ... 207
6.1.2 Операции преобразования символов ... 207
6.1.2.1 Приведение буквенного символа к верхнему регистру ... 207
6.1.2.2 Приведение буквенного символа к нижнему регистру ... 207
6.1.2.3 Преобразование символа в число ... 207
6.1.2.4 Символ, предшествующий указанному ... 208
6.1.2.5 Символ, следующий за указанным ... 208
6.1.2.6 Смещение по кодовой таблице на указанное число символов ... 208
6.1.3 Ввод символов ... 208
6.1.4 Пример: нахождение суммы в символьном виде ... 211
6.2 Строковый тип данных ... 212
6.2.1 Ввод строк ... 214
6.2.2 Вывод строк ... 217
6.2.3 Длина строки ... 217
6.2.4 Арифметические операции со строками (+ и *) ... 219
6.2.5 Сравнение строк ... 220
6.2.6 Копирование строк ... 221
6.2.7 Выделение подстроки ... 222
6.2.8 Срезы строк ... 222
6.2.9 Смена регистра символов ... 223
6.2.10 Удаление символов в начале и в конце строки ... 223
6.2.11 Удаление подстрок ... 224
6.2.12 Инверсия ... 225
6.2.13 Вставка подстроки ... 226
6.2.14 Проверки в строке ... 226
6.2.15 Разбиение строки на слова ... 227
6.2.16 Замена подстроки в строке ... 228
6.2.17 Сцепление (слияние) строк ... 228
6.2.18 Поиск в строке ... 231
6.2.19 Использование регулярных выражений ... 233
6.2.19.1 Некоторые метасимволы ... 234
6.2.19.2 IsMatch – наличие подстроки в строке ... 236
6.2.19.3 RegexReplace – замена подстрок в строке ... 237
6.2.19.4 MatchValue – поиск первого вхождения подстроки ... 238

6.2.19.5 MatchValues – поиск всех вхождений подстроки ... 238
6.2.19.6 Matches – результаты поиска в элементах типа Match ... 238
6.2.19.7 Опции регулярного выражения ... 239
6.2.19.8 Элементы Matches ... 239
6.2.20 Преобразование строки в динамический массив ... 240
6.2.21 Преобразование целых чисел к строке ... 240
6.2.22 Преобразование вещественных чисел к строке ... 241
6.2.23 Преобразование строки к числу ... 241
6.2.23.1 Преобразование строки к целому числу ... 241
6.2.23.2 Преобразование строки к вещественному числу ... 242
6.2.23.3 Преобразование строки к массиву чисел ... 242
6.2.24 Пример с преобразованиями строк и чисел ... 243
6.2.25 Чтение данных из строки ... 244
6.2.25.1 Чтение из строки данных типа integer ... 244
6.2.25.2 Чтение из строки данных типа real ... 245
6.2.25.3 Чтение из строки символьных данных ... 245
6.2.26 Форматирование данных для вывода ... 246
6.2.26.1 Выравнивание строки пробелами ... 247
6.2.26.2 Составное форматирование ... 247
6.2.26.3 Функция и статический метод Format ... 249
6.2.26.4 Вывод с использованием составного форматирования ... 249
6.2.26.5 Интерполированные строки ... 249
6.3 Для самостоятельного решения ... 250

Часть 7 Типы данных ... 253

7.1 Записи ... 255
7.1.1 Обращение к полям записи ... 256
7.1.2 Конструктор записи ... 257
7.1.3 Инициализаторы полей ... 259
7.1.4 Инициализация записи ... 259
7.1.5 Вывод переменной типа запись ... 261
7.1.6 Операция присваивания для записей ... 262
7.1.7 Сравнение записей ... 262
7.1.8 Передача записей в качестве параметров ... 263
7.2 Перечислимый тип ... 263
7.3 Диапазонный тип ... 265

7.4 Эквивалентность и совместимость типов ... 265
7.4.1 Совпадение типов ... 266
7.4.2 Эквивалентность типов ... 267
7.4.3 Совместимость типов ... 267
7.5 Пример: работа с таблицей ... 268
7.6 Обобщенный тип ... 273
7.7 Указатели ... 275
7.8 Тип данных BigInteger ... 278
7.8.1 Инициализация данных типа BigInteger ... 279
7.8.2 Приведение BigInteger к другому типу ... 280
7.9 Тип данных decimal ... 280
7.10 Тип данных DateTime ... 281
7.11 Для самостоятельного решения ... 283

Часть 8 Многомерные массивы ... 285

8.1 Динамические двухмерные массивы (матрицы) ... 286
8.1.1 Описание и создание матриц ... 287
8.1.2 Генерация матриц ... 288
8.1.2.1 Заполнение случайными значениями ... 288
8.1.2.2 Заполнение фиксированным значением ... 289
8.1.2.3 Заполнение значениями, зависящими от индексов ... 290
8.1.2.4 Заполнение на основе одномерного массива ... 290
8.1.3 Клавиатурный ввод значений элементов матриц ... 291
8.1.4 Вывод матриц (.Print) ... 291
8.1.5 Переопределение размеров матрицы ... 292
8.1.6 Получение сведений о текущих размерах матрицы ... 293
8.1.7 Выборка элементов матрицы ... 293
8.1.8 Перебор всех элементов матрицы (.foreach) ... 295
8.1.9 Модификация строк и столбцов ... 295
8.1.10 Транспонирование матрицы ... 297
8.1.11 О матричных операциях ... 298
8.2 Статические двухмерные массивы ... 298
8.3 Массивы размерности выше двух ... 299
8.4 Примеры решения задач с матрицами ... 299
8.5 Для самостоятельного решения ... 302

Часть 9 Файлы 305

9.1 Файловый тип данных 307
9.1.1 Описание данных файлового типа 309
9.1.2 Некоторые термины 310
9.2 Связь программы с файлом 311
9.3 Открытие и закрытие файлов 312
9.3.1 Открытие текстового файла 313
9.3.1.1 Кодировка текстового файла 313
9.3.1.2 Открытие текстового файла для чтения 314
9.3.1.3 Открытие текстового файла для записи 315
9.3.1.4 Открытие текстового файла для дозаписи 315
9.3.2 Открытие типизированных файлов 316
9.3.2.1 Открытие типизированного файла для чтения/записи 317
9.3.2.2 Открытие типизированного файла для создания/перезаписи 318
9.3.3 Открытие бестиповых файлов 319
9.3.3.1 Открытие бестипового файла на чтение/запись 319
9.3.3.2 Открытие бестипового файла на создание/перезапись 320
9.3.4 Закрытие файлов 320
9.4 Запись данных в файл 321
9.4.1 Запись данных в текстовый файл 321
9.4.1.1 Write – и это все? 323
9.4.1.2 Отзвуки прошлого (файлы input/output) 324
9.4.2 Запись данных в типизированный файл 325
9.4.3 Запись данных в бестиповый файл 328
9.5 Чтение данных из файла 331
9.5.1 Чтение данных из текстового файла 331
9.5.1.1 Средства чтения данных из текстового файла 332
9.5.1.2 Обработка ситуации «конец файла» для текстовых файлов 336
9.5.1.3 О пробельных символах в конце записи и файла 337
9.5.2 Чтение данных из типизированного файла 337
9.5.2.1 Средства чтения данных из типизированного файла 337
9.5.2.2 Ситуация «конец файла» для типизированных файлов 340
9.5.3 Чтение данных из бестипового файла 340
9.6 Операции с файловым указателем 343
9.7 Прочие операции с файлами 345
9.8 Рекомендации по работе с файлами 346
9.9 Для самостоятельного решения 346

Часть 10 Стандартные коллекции ... 349

10.1 Линейные списки ... 351
10.1.1 Стек (Stack) ... 351
10.1.1.1 Операции для работы со стеком ... 354
10.1.1.2 Примеры использования стека ... 354
10.1.2 Очередь (Queue) ... 358
10.1.2.1 Операции для работы с очередью ... 359
10.1.2.2 Примеры использования очереди ... 360
10.1.3 Линейные списки: заключение ... 361
10.2 Двусвязный список LinkedList ... 362
10.2.1 Операции для работы с двусвязным списком ... 363
10.2.2 Некоторые приемы работы со списком LinkedList ... 365
10.2.3 Пример работы со списком LinkedList ... 367
10.2.4 Моделирование дека на основе LinkedList ... 368
10.3 Список List ... 370
10.3.1 Создание списка List ... 371
10.3.2 Операции для работы со списком List ... 372
10.3.3 Примеры использования списков List ... 374
10.4 Множество HashSet ... 376
10.4.1 Понятие о хэш-таблицах ... 377
10.4.2 Создание множества HashSet ... 380
10.4.3 Операции для работы с множеством HashSet ... 382
10.4.4 Примеры работы со множествами HashSet ... 383
10.5 Словарь Dictionary ... 385
10.5.1 Структура KeyValuePair<ключ, значение> ... 385
10.5.2 Создание словаря ... 386
10.5.3 Операции для работы со словарем ... 387
10.5.4 Примеры работы со словарем ... 388
10.6 Коллекции с упорядоченностью элементов ... 389
10.6.1 Упорядоченное множество SortedSet ... 390
10.6.1.1 И снова компараторы, снова интерфейс ... 391
10.6.1.2 Дополнительные операции SortedSet ... 393
10.6.1.3 Пример использования множества SortedSet ... 393
10.6.2 Упорядоченный список SortedList ... 394
10.6.3 Упорядоченный словарь SortedDictionary ... 395
10.7 Для самостоятельного решения ... 396

Часть 11 Модули и библиотеки ... 397
11.1 Модули ... 399
11.1.1 Структура модуля ... 399
11.1.1.1 Заголовок модуля ... 399
11.1.1.2 Раздел интерфейса ... 399
11.1.1.3 Раздел реализации ... 400
11.1.1.4 Раздел инициализации ... 401
11.1.1.5 Раздел финализации ... 402
11.1.1.6 Раздел uses ... 402
11.1.1.7 Локальные и глобальные имена ... 403
11.1.1.8 Пространство имен ... 403
11.1.1.9 Область видимости имен ... 404
11.1.1.10 Упрощенный синтаксис модуля ... 407
11.1.2 Циклические ссылки между модулями ... 408
11.2 Библиотеки dll ... 408
11.2.1 Структура библиотеки ... 409
11.2.2 Подключение библиотек ... 409
11.3 Документирующие комментарии ... 410
11.4 Пример программы ... 411
11.4.1 Реализация на основе модулей ... 412
11.4.2 Реализация на основе библиотеки ... 415
11.4.3 Реализация на основе модуля и библиотеки ... 418
11.5 Стандартные модули в PascalABC.NET ... 421
11.5.1 Модули для работы с графикой ... 421
11.5.1.1 Модуль GraphABC ... 421
11.5.1.2 Модуль GraphWPF ... 422
11.5.1.3 Модуль ABCObjects ... 422
11.5.1.4 Модуль WPFObjects ... 422
11.5.1.5 Модуль Graph3D ... 422
11.5.1.6 Модуль ABCSprites ... 423
11.5.2 Учебные модули ... 423
11.5.2.1 Модуль Робот ... 423
11.5.2.2 Модуль Чертежник ... 423
11.5.2.3 Модуль NumLibABC ... 423
11.6 Для самостоятельного решения ... 424

Часть 12 Обработка ошибок в программе ... 427

12.1 Предотвращение исключений ... 429
12.1.1 Подмена стандартных функций ... 430
12.1.2 Системная процедура Assert ... 431
12.1.3 Использование IDE-отладчика ... 433
12.2 Обработка исключений ... 434
12.2.1 Виды исключений ... 435
12.2.2 Оператор try … except ... 435
12.2.3 Поиск причины и места возникновения ошибки ... 437
12.2.3.1 Оператор try … finally ... 440
12.2.3.2 Оператор raise ... 441

Часть 13 Объектно-ориентированное программирование ... 445

13.1 Базовые понятия ... 446
13.1.1 Классы и объекты ... 447
13.1.1.1 Описание класса ... 449
13.1.1.2 Конструкторы и создание объекта ... 450
13.1.1.3 Три кита объектно-ориентированного программирования ... 452
13.1.1.4 Опережающие и циклические ссылки в классах ... 454
13.1.2 Поля класса ... 456
13.1.3 Методы класса ... 457
13.1.4 Свойства класса ... 458
13.1.5 Индексные свойства ... 463
13.1.6 Автоклассы и автосвойства ... 465
13.2 Наследование ... 466
13.2.1 Принцип подстановки Лисков ... 468
13.2.2 Перегрузка операций ... 468
13.3 Полиморфизм ... 472
13.3.1 Виртуальные методы ... 472
13.3.2 Абстрактные методы и классы ... 476
13.4 Операции as и is ... 482
13.4.1 Операция is ... 486
13.4.2 Операция as ... 487
13.5 Сопоставление с образцом ... 488
13.6 Деконструкторы ... 490
13.7 Анонимные классы ... 492

13.8 Методы расширения ... 494
13.8.1 Переменная Self ... 495
13.8.2 Перегрузка операций посредством расширения ... 497
13.9 Интерфейсы ... 498
13.9.1 Описание интерфейса ... 499
13.9.2 Реализация интерфейса в классе ... 500
13.9.3 Наследование интерфейсов ... 502
13.9.4 Некоторые стандартные интерфейсы .NET ... 503
13.9.4.1 Сравнение объектов. Интерфейс IComparable<T> ... 503
13.9.4.2 Сравнение объектов. Интерфейс IComparer<T> ... 505
13.9.4.3 Клонирование объекта. Интерфейс ICloneable ... 508
13.10 Статический класс System.Console ... 511
13.10.1 Некоторые свойства класса Console ... 512
13.10.2 Некоторые методы класса Console ... 512
13.11 Класс – обработчик исключений ... 513
13.12 Для самостоятельного решения ... 518

Часть 14 Ответы и решения ... 521

14.1 К части 1 ... 522
14.2 К части 2 ... 523
14.3 К части 3 ... 523
14.4 К части 4 ... 527
14.5 К части 5 ... 531
14.6 К части 6 ... 533
14.7 К части 7 ... 534
14.8 К части 8 ... 538
14.9 К части 9 ... 540
14.10 К части 10 ... 544
14.11 К части 11 ... 547
14.12 К части 13 ... 549

Приложения ... 557

Приложение 1 ... 558
Приложение 2 ... 558
Приложение 3 ... 559
Приложение 4 ... 561

Предметный указатель ... 565

Введение

Самая важная вещь в языке программирования — его имя. Язык не будет иметь успеха без хорошего имени. Я недавно придумал очень хорошее имя, теперь осталось изобрести подходящий язык.
Дональд Кнут, американский ученый

Язык программирования PascalABC.NET – мощный современный язык, включающий в себя классический Паскаль, большинство возможностей языка Object Pascal среды Delphi и многочисленные собственные расширения. Компилятор и библиотеки языка PascalABC.NET свободно распространяются по лицензии LGPL v3.

На базе языка PascalABC.NET создана одноименная интегрированная среда разработки и отладки программ, поддерживающая технологию IntelliSence. Реализация выполнена на платформе Microsoft .NET Framework и опирается на использование ее библиотек. В операционных системах Linux и MacOS требуется среда проекта Mono, под которой запускается консольный компилятор.

PascalABC.NET – мультипарадигменный язык. Он позволяет программировать в структурном, объектно-программированном и функциональном стиле, а также смешивать эти стили в одной программе. Насколько это было возможно, PascalABC.NET впитал лучшие идеи, реализованные в языках C#, Python и Haskell. Большое количество «синтаксического сахара» делает программирование на PascalABC.NET простым и комфортным как для начинающих, так и для имеющих опыт в создании программ.

Немного истории

Первый высококачественный компилятор языка Паскаль для персональных компьютеров с операционной системой DOS появился в 1983 году под названием Turbo Pascal. В дальнейшем он был переименован в Borland Pascal и под таким именем существует до сих пор.

Популярность Turbo/Borland Pascal привела к выпуску множества литературы, посвященной программированию в этой, замечательной для своего времени среде. Язык массово изучался в школах и вузах. В 1994 году компания Borland выпустила последнюю версию 7.1 и затем отказалась от дальнейшей поддержки проекта в связи переходом к Borland Delphi. Компания не открыла исходные тексты, оставив лицензию проприетарной. Это привело к тому, что что любой экземпляр Turbo/Borland Pascal, установленный на компьютер в последние два десятка лет, оказывается нелицензионной («пиратской») копией. Об этом не следует забывать учителям информатики школ и других учебных заведений, до сих пор предлагающим учащимся «скачать себе где-нибудь Турбо Паскаль».

Появление Borland Delphi несколько снизило интерес к Borland Pascal. Визуальная среда быстрой разработки, поддержка объектно-ориентированных технологий – все это вызывало большой интерес, сдерживаемый лишь ценой на продукт. Использовать Delphi для целей обучения могли себе позволить лишь немногие вузы.

Шли годы, система совершенствовалась, но цена на нее особенно не снижалась. Параллельно развивалась операционная система Windows, для которой Borland Pascal был чужеродным – он работал в DOS-режиме. Windows XP оказалась последней операционной системой, в которой еще можно было беспроблемно использовать Borland Pascal.

В 1997 стартовал проект по свободно распространяемой мультиплатформенной версии Паскаля, получивший название Free Pascal (FPC) - реализация на базе языка Object Pascal. Существует также открытый проект визуальной разработки программ Lazarus, основанный на FPC.

Языки программирования развиваются непрерывно. Здесь Паскалю не повезло. Заложенное еще его создателем, Н. Виртом, стремление сделать язык простым, породило множество ограничений, сдерживающих развитие Паскаля. Язык оказался неспособным поддерживать современные идеи, методы и технологии программирования. Как следствие, в коммерческой сфере интерес к Паскалю со временем пропал, и язык остался востребованным лишь в сфере образования.

В 2003 году энтузиастами Южного федерального университета (ЮФУ, г. Ростов-на-Дону) была создана учебная среда программирования Pascal ABC (ABC в английском языке имеет значение «азбука»), как альтернатива платным средам на базе языка Паскаль. Pascal ABC очень быстро набрал популярность в учебных заведениях России и стран бывшего СССР. Впоследствии разработчики решили дать Паскалю вторую жизнь. В процессе выработки спецификаций «нового Паскаля» была учтена потребность обеспечить быстрый последующий переход к изучению языка C#, а также шуточные пожелания преподавателей ЮФУ «писать в одну-две строчки» любые программы, которые даются в школах. В 2009 году появилась первая стабильная версия открытого проекта PascalABC.NET 1.2.

За прошедшие годы произошли существенные изменения в коллективе разработчиков, язык также претерпел множество изменений. На время написания этой книги актуальной является версия PascalABC.NET 3.5.

В книге все реализации языка Паскаль, кроме PascalABC.NET, будут называться общим термином «***базовый Паскаль***». В случае необходимости может указываться конкретная реализация.

Зачем изучать PascalABC.NET?

1. Базовый Паскаль за пределами сферы образования практически не используется. Можно возразить, что и PascalABC.NET также не используется за пределами сферы образования. Но если изучать Паскаль, то такой, на котором удобно и быстро писать и отлаживать программы.

2. PascalABC.NET позволяет в очень короткий срок перейти к изучению современных коммерческих языков программирования, например, C#.

3. PascalABC.NET позволяет изучить современные технологии программирования, научиться оперировать последовательностями и кортежами, писать программы с элементами функционального стиля, применять стандартные коллекции.

4. PascalABC.NET дает возможность отказаться от концепции статических массивов в пользу динамических, существенно упрощая работу с массивами, в том числе, при обмене с процедурами и функциями.

5. Возможность решать в несколько строчек большинство задач, даваемых в учебниках по информатике, позволяет преподавателю сосредоточиться на алгоритмах решения, а не переписывать каждый раз одни и те же фрагменты реализации. Быстрое написание и отладка позволяют за одно занятие рассмотреть намного больше задач, повышая эффективность обучения.

6. Алгоритм, реализованный на PascalABC.NET нагляден и легко читается, делая лишним рисование блок-схем. Вносить изменения в готовую программу также быстро и легко. Дополнительным удобством является возможность использовать в программах идентификаторы, содержащие буквенные символы, отличные от латиницы (кириллические, греческие и т.д.).

7. Скорость выполнения готовой программы в большинстве случаев такая же, как у программы, написанной на C#.

8. В PascalABC.NET всегда остается возможность рассмотреть подробную реализацию нужного алгоритма на «низкоуровневом» базовом Паскале.

9. В PascalABC.NET можно обращаться к любым библиотекам платформы .NET Framework. Можно создавать свои библиотеки, которые затем использовать, в том числе, и в программах. написанных на других .NET-языках.

О требовании эффективности программ

При знакомстве с заданиями по информатике, предлагаемыми в процессе обучения школьников, учащихся и студентов, встречается пожелание, а то требование написать программу, «эффективную по памяти и/или эффективную по времени». Вот, например, подобные критерии ФИПИ, предлагаемые для оценивания результатов

сдачи ЕГЭ-2019: «Программа считается эффективной по времени, если при увеличении количества исходных чисел N в k раз время работы программы увеличивается не более чем в k раз. Программа считается эффективной по памяти, если память, необходимая для хранения всех переменных программы, не превышает 1 килобайта и не увеличивается с ростом N».

Смысл такого требования для экзаменационной или иной проверочной работы ясен: оценка навыков алгоритмического мышления. Но к программированию на конкретном языке это имеет очень отдаленное отношение.

ЭВМ трех первых поколений занимали огромные площади, требовали больших расходов на обслуживание, содержание и ремонт, а доступ к ним был весьма ограничен. В производительности и объеме оперативной памяти основная масса ЭВМ уступала современному смартфону. Все это делало каждый час работы машинного времени очень дорогим, а уровень программистов оценивался их способностью писать те самые «эффективные» программы.

Появление персональных компьютеров изменило ситуацию коренным образом. Вычислительная техника стала надежной и общедоступной, а ее цена, расходы на обслуживание и ремонт – небольшими. Стоимость машинного часа «персоналки» сейчас оказывается ничтожной в сравнении со стоимостью часа работающего с ней человека. И программиста, в том числе. Программирование из элитарной профессии превратилось в обыденное дело и в обязательном порядке изучается каждым школьником. Изменились технологии программирования, появились новые алгоритмические языки.

В этих условиях на первое место выходит уже не критерий эффективности программного продукта, а трудозатраты на его разработку и последующее сопровождение – именно они определяют конечную стоимость программного обеспечения. Фирмы-разработчики с мировым именем бесцеремонно заявляют: «Программа медленно работает? Купите более современный компьютер или попробуйте нарастить оперативную память».

А что же с эффективностью? Она по-прежнему важна при разработке системного программного обеспечения, задач, работающих в реальном времени, программ для микропроцессоров и суперкомпьютеров. Многие ли программисты этим работают? Возможно, 1% или даже 0.1% от общего их количества. Остальные спокойно создают программы, требующие для работы десятков мегабайт памяти, в то время как такие же программы на ЭВМ третьего поколения успешно работали в 64 Кбайтах.

Но все же, – мы для компьютеров или компьютеры для нас? Чье время и силы дороже – человека или машины? Если программа «съест» лишний килобайт памяти, а процессор затратит на вычисление лишнюю десятую долю секунды, но при этом человек высвободит полчаса при создании программы – неужели это так плохо и неправильно?

Требование «эффективности» сейчас – это в первую очередь требование писать программу так, чтобы ее было легко читать, понимать, отлаживать и сопровождать. Чтобы программа, написанная одним человеком, могла быть в короткие сроки понята и модифицирована другим. А разговоры об экономии памяти и времени исполнения – они удел отдельных авторов, написавших школьные (и не только) учебники еще в эпоху ЭВМ третьего поколения и регулярно переиздающих свои творения без существенных изменений. Вызывает лишь горькую иронию обучение работе в «системе программирования Турбо Паскаль», благополучно скончавшейся четверть века назад.

Мы не будем в дальнейшем заниматься анализом «эффективности по памяти и по времени» - разработка алгоритмов выходит за пределы данной книги, а направим усилия в сторону написания за минимальное время коротких и понятных другим программ. Таковы требования сегодняшнего времени.

Чего вы здесь не найдете

Эта книга не научит вас работать с классическим языком Паскаль, представленным Н.Виртом, и прочими реализациями базового Паскаля, – она с самого начала предполагает изучение языка PascalABC.NET. Вы можете продолжать работать с базовым Паскалем – он достаточно полно поддерживается средой, но его синтаксис и семантика упоминаются лишь там, где это сочтено необходимым.

Вы не научитесь составлять базовые алгоритмы и рисовать блок-схемы – для этой цели существует множество других превосходных книг.

Здесь нет описания архитектуры и схемотехники компьютеров, не рассматриваются основы двоичной арифметики: автор не ставил задачи попытаться заменить школьные учебники информатики.

Как пользоваться этой книгой

Книга рассчитана не только на новичков, начинающих изучать программирование на Паскале, но также и на читателей, обладающих опытом работы с этим языком. Первые научатся писать программы на PasacalABC.NET, вторые получат возможность повысить свой уровень владения языком, освоить современные технологии и приемы программирования.

Главное условие успешного усвоения материала – занятия перед компьютером с загруженной средой программирования PascalABC.NET версии не ниже 3.5. Настоятельно рекомендуется выполнять все примеры программ, заключенные в рамку. Тексты этих программ, за исключением примитивных, прилагаются.

Предполагается, что пользователь умеет работать с компьютером, способен установить на нем необходимые программные продукты и компоненты, обладает базовыми знаниями информатики на уровне требований учебной программы 6-7 класса средней школы. Для понимания многих задач может понадобиться знание математики.


Материал предназначен для начинающих.


Материал рассчитан на читателя, имеющего опыт в программировании.


Материал требует определенных знаний математики.

В тексте будут встречаться слова и словосочетания, выделенные ***полужирным курсивом***. Это указывает на первое появление нового термина, необходимого для понимания дальнейшего материала. Если термин описан в другом месте, за ним следует ссылка на описание, заключенная в круглые скобки. Отсутствие такой ссылки подразумевает, что термин или должен быть читателю знаком, или разъясняется на месте.

Все приведенные в книге тексты программ, даже если они рассматриваются как варианты для базового Паскаля, работоспособны в PascalABC.NET, если прямо не сказано иное.

Если вы не можете сформулировать на естественном языке алгоритм нахождения минимума из трех чисел и никогда не видели блок-схем – эта книга не для вас!

Парадигмы программирования

PascalABC.NET – современный мультипарадигменный язык программирования. Это означает, что на нем можно писать программы, сочетая различные ***парадигмы***. Не вдаваясь в подробности, будем считать, что под парадигмой понимается некий набор понятий, образующих стиль написания программы.

Ранние универсальные алгоритмические языки базировались на ***императивной*** парадигме, предполагающей запись программы в виде набора инструкций, которые нужно последовательно выполнить для получения результата. Императивная парадигма отвечает на вопрос о том, ***как*** следует решать проблему. Противоположностью является ***декларативная*** парадигма, в которой указывается, что задано и каким должен быть результат. Декларативная парадигма отвечает на вопрос о том, ***что*** нужно сделать для решения проблемы. Возможность сочетать в программе обе эти парадигмы делает программирование комфортным, позволяя не отвлекаться на мелочи.

Одна из первых предложенных парадигм – ***процедурная***. В процессе императивного программирования в последовательных участках кода выделяются некоторые

блоки, к которым происходит обращение более одного раза. Эти блоки выделялись в языковые конструкции, получившие название процедур. В процедурной парадигме программа представляет собой набор процедур, одна из которых является главной и из нее производятся обращения к прочим.

Развитие процедурного программирования привело к возникновению парадигмы ***структурного программирования***. В ее основе лежит представление программы, как совокупности процедурных блоков, которая имеет четкую иерархию. Это позволяет лучше видеть всю структуру связей между отдельными блоками, одновременно предполагая, что внесение изменений в один из них не влияет на работу остальных.

Парадигма ***объектно-ориентированного*** программирования предполагает взгляд на программу, как на совокупность отдельных объектов, определенным образом взаимодействующих друг с другом. При этом, каждый объект создается на основе своеобразного «чертежа» - класса.

Все упомянутые парадигмы применяются в императивном программировании. Этим подходам противопоставляется парадигма ***функционального программирования***, трактующая реализацию алгоритма, как процесса нахождения значений некоторых математических функций. Подробнее этот вопрос будет рассмотрен далее.

Возвращаясь к PascalABC.NET можно сказать, что у программиста есть возможность выбрать одну любую парадигму и придерживаться ее, либо в достаточно произвольной пропорции использовать в программе часть или даже все из вышеперечисленных парадигм, получая при этом качественный и наглядный программный код с минимальными затратами труда.

Наша первая программа

В языке Паскаль в простейшем случае программа начинается с английского слова **begin** (начало) и заканчивается словом **end**. (конец) – именно так, с точкой на конце. Конструкция begin … end называется ***операторными скобками***, поэтому вполне допустимо сказать, что простейшая программа – это операторные скобки, за которыми следует точка. Можно набрать такую программу и даже запустить ее. Конечно, она ничего не выведет (и ничего не сделает), но и сообщений об ошибке выдано не будет.

```
begin
end.
```

Формат записи текста программы свободный, и это означает, что его размещение может быть произвольным, например, даже таким: **begin end**. Полезно знать, что у программистов сложился определенный стиль записи текстов программ, который облегчает восприятие программы и ее отладку. Выделенные в тексте жирным

шрифтом слова программы называются ***ключевыми*** (служебными словами, зарезервированными в языке для нужд программы).

Конструкции языка Паскаль называются ***операторами***. Это нестрогое, но вполне функциональное определение, а разбор подробностей может завести очень далеко. Операторы разделяются символом «точка с запятой». Перед **end** указывать точку с запятой не требуется, но если вы до этого программировали на языках семейства C (произносится «си»), – можете упорно продолжать ее ставить. Каждый оператор обычно располагают с новой строки, если нет серьезных причин разместить в одной строке несколько операторов.

Если в операторные скобки заключить некоторый набор операторов, получится ***составной оператор*** или ***блок***. Блок в программе может использоваться везде, где может использоваться одиночный оператор языка. Получается, что программой может быть блок, завершающийся точкой.

Наша первая программа будет выводить на монитор фразу «Привет! Я - PascalABC.NET».

```
begin // p00001
  Println('Привет! Я - PascalABC.NET')
end.
```

Слово **begin** и соответствующее ему **end** принято располагать на отдельных строках строго друг под другом, если нет какой-то причины отходить от такого расположения. Это операторные скобки и они, как любые скобки, должны быть парными, т.е. каждой «открывающей скобке» **begin** соответствует своя «закрывающая скобка» **end**, что подчеркивается таким ступенчатым расположением. Все, что находится в операторных скобках, записывается с отступом. Такое расположение удобно делать при помощи клавиши табуляции «Tab».

Оператор Println('текст') организует вывод на монитор текста, записанного в одинарных кавычках, после чего осуществляется переход к следующей строке. Чтобы не делать такого перехода, вместо Println используется Print.

В любом месте программы можно располагать пояснения, которые называются ***комментариями***. Существуют несколько способов записи комментариев:

- комментарий можно начать символами //, например

```
// комментарий – все до конца строки;
```

- комментарий можно начать символом: {, например

```
{ комментарий – все,
        пока не встретится }
```

Фигурные скобки { … } можно заменить на (* … *).

А теперь введите текст программы и запустите ее нажатием клавиши F9.

Если нужно вывести несколько строк, пока что для каждой строки будем записывать отдельный оператор Println.

```
begin // p00002
  Println('Привет! Я - PascalABC.NET'); // первая строка вывода
  Println('------------------------')  // вторая строка вывода
end.
```

Символы подчеркивания были специально перенесены на другую строчку и выравнены по левой одинарной кавычке. Это позволяет увидеть, как будет оформлен вывод и одновременно избавляет от необходимости подсчитывать количество выводимых символов.

Отметим, что если оператору Print передать не один элемент, а несколько, разделенных запятыми, то после вывода каждого элемента будет делаться пробел.

```
Print(Элемент1, Элемент2, … ЭлементN);
Print('Маша', 'ела', 'кашу');
```

В случае, если пробелы между выводимыми элементами не нужны, вместо оператора Print (Println) используется оператор Write (Writeln). Можно в одном операторе Print (Write) организовать вывод в несколько строк. Для этого в месте, где нужна смена строки, надо записать элемент с именем Newline.

```
Println('Маша', NewLine, 'ела', Newline, 'кашу');
```

Здесь все три слова будут выведены в столбик. Любители стиля «ретро» могут вместо Newline (англ. – новая строка) писать известную в базовом Паскале комбинацию символов #13#10.

Понятие о типах данных

В программе на языке Паскаль для всех переменных и констант к моменту их первого использования должен быть известен ***тип данных***. Он указывает на то, как данные должны быть организованы для хранения в памяти компьютера, а также определяет правила их обработки. Паскаль относится к языкам со строгой типизацией. Это означает, что тип данных указывается обязательно и впоследствии не может быть изменен. Указание типа выполняется явно при описании в программе переменной или именованной константы, но может быть также выполнено компилятором на основе автоматического выведения.

Если в одной переменной некоторого типа может содержаться множество значений, тип данных называется ***структурированным***. Таковы массивы, строки, файлы и т.д.

Особым типом данных является ***последовательность***, которая по существу хранит алгоритм последовательного получения данных.

Большинство типов в PascalABC.NET подразделяются на размерные, ссылочные и типы указателей. В то же время, статические массивы, множества, размерные строки и файлы, унаследованные от Object Pascal (Delphi), нельзя однозначно отнести к размерному или ссылочному типу. По представлению в памяти они относятся к ссылочному типу, а по своему поведению - к размерному.

Следует понимать, что приведенная классификация является достаточно условной. Несмотря на то, что PascalABC.NET по возможности сохраняет поведение типов, которые пришли из языка Object Pascal (Delphi), имеется ряд особенностей реализации, связанных как с платформой .NET, так и с самим PascalABC.NET. Об этих особенностях будет подробно сообщаться в соответствующих главах книги.

Каждый тип данных в PascalABC.NET имеет отображение на тип .NET Framework. Все типы, кроме типов указателей, являются производными от типа **Object**, а Тип указателя принадлежит к неуправляемому коду и моделируется типом **void***. Можно дополнительно уточнить, что размерные типы данных являются производными от System.ValueType.

Часть 1

Арифметика целых чисел

— Два… и еще один…
— А два и еще один будет?...
Детрит запаниковал. В дело пошла высшая математика.
Терри Пратчетт.
«К оружию! К оружию!»

Когда у человека возникла потребность в счете, появились первые числа. Такие числа в математике называют натуральными. Натуральный ряд чисел – это числа 1, 2, 3, … Он образует бесконечное множество натуральных чисел $\mathbb{N}$, поэтому наибольшего натурального числа не существует. А наименьшее натуральное число равно единице.

Если к множеству $\mathbb{N}$ добавить ноль, то получится множество целых неотрицательных чисел. Отрицательные числа и ноль расширяют $\mathbb{N}$ до множества целых чисел $\mathbb{Z}$.

Над множеством $\mathbb{Z}$ определен ряд арифметических операций: сложение, вычитание, умножение, целочисленное деление с остатком и изменение знака.

В математике множества $\mathbb{N}$ и $\mathbb{Z}$ бесконечны. Но в компьютерах (и в языках программирования) они ограничены размерами памяти, отводимыми на представление целых чисел. Натуральные числа и ноль представляются в прямом двоичном коде, поэтому диапазон их представления составляет $0 \le n \le 2^m-1$, где m – количество бит, отводимых под представление числа n. Отрицательные числа представляются в дополнительном коде, и диапазон их представления составляет $-2^m \le n < 0$.

В компьютерной арифметике преимущества работы с целыми числами заключаются в том, что результаты арифметических операций всегда точны, а время выполнения обычно меньше, чем при работе с числами других разновидностей. Главный недостаток - необходимость следить за тем, чтобы значения не выходили за пределы отведенного им диапазона.

1.1 Целые типы в PascalABC.NET

Перед тем, как начать работу, мы оцениваем отведенное нам рабочее пространство с тем, чтобы определить, разместится ли в нем все необходимое. Такую же оценку должен сделать компилятор перед созданием машинного кода. С этой целью программист указывает в своей программе тип каждого элемента данных. Тип определяет множество значений, которые может принимать элемент, а также множество производимых с ним операций. Зная тип, компилятор может определить размер памяти, который следует отвести под данные и особенности операций, которые с этими данными надо произвести. Типы целых чисел, приведены в таблице 1.1.

Тестирование, проведенное автором в среде 32-разрядной операционной системы Windows, показало что процессор Intel Core 2 Duo E8400 обеспечивает наибольшую скорость выполнения арифметических операций при работе с данными типа **integer** и **cardinal**. Использование типов данных **byte** (**shortint**) ухудшает производительность на 15-20%, тип данных **int64** (**uint64**) ухудшает производительность на 10-20%, а тип **word** (**smallint**) показал двукратное драматическое снижение производительности. Возможно, в других условиях будут получены другие данные, но это лишь доказывает, что всегда полезно знать особенности системы, в которой исполняется программа. Особую актуальность такое знание приобретает при выполнении встречающихся в различных конкурсных заданиях условий написать программу, оптимальную по времени исполнения и/или памяти.

Длина, байт	Без знака	Со знаком
1	**byte** 0 .. 255	**shortint** -128 .. 127
2	**word** 0 .. 65 535	**smallint** -32 768 .. 32 767
4	**longword**, **cardinal** 0 .. 4 294 967 295	**integer**, **longint** -2 147 483 648 .. 2 147 483 647
8	**uint64** 0 .. 18 446 744 073 709 551 615	**int64** -9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807
переменная		**BigInteger** Ограничено только памятью компьютера

Таблица 1.1. Целые типы PascalABC.NET

Помимо перечисленных в таблице 1.1, существует тип **BigInteger**. Он позволяет работать с данными, имеющими в записи практически неограниченное количество цифр. Для работы с числами в таком формате используется поразрядная арифметика (примерно, как мы считаем «в столбик»), поэтому скорость работы с данными типа **BigInteger** во много раз ниже, чем с прочими. Любые целочисленные данные могут быть без потерь преобразованы к типу **BigInteger**. Обратное преобразование к другим целым типам запрещено, поскольку оно может сопровождаться потерей данных. Подробнее о работе с этим типом данных будет рассказано в части 7. Пока отметим, что данные типа **BigInteger** могут включаться в выражения наряду с прочими типами.

1.2 Константы

В программе могут присутствовать величины, значения которых неизменяемы. Они могут быть никогда не изменяемыми (например, в 1 метре 100 см и поэтому коэффициент для перевода метров в сантиметры всегда равен 100), или неизменяемыми в пределах программы (требуется найти сумму 10 случайных чисел из диапазона от -5 до +20). Такие неизменяемые величины называют константами.

Константы описываются перед программным блоком **begin** … **end** в так называемом ***разделе описания констант***, начинающемся служебным словом **const**.

Каждая константа описывается в виде

```
имя константы = значение; // именованая константа с неявным типом
```

или

```
имя константы: тип = значение; // типизированная константа
```

В диалекте языка Turbo Pascal (Borland Pascal) значение типизированной константы разрешено изменять присваиванием. В Borland Delphi такое изменение по умолчанию запрещено, но его можно разрешить директивой компилятора {$J+}. Free Pascal, наоборот, по умолчанию разрешает изменять значения типизированных констант, но это изменение можно запретить директивой компилятора {$J-}. PascalABC.NET **категорически запрещает** менять значение констант, в том числе типизированных.

Основное назначение констант – задавать их значения в одном, общем для всей программы месте и пресекать любые попытки программиста изменить эти значения. Подумайте, хорошо ли будет, если где-то в программе случайно изменить ранее объявленное значение числа π?

В языке PascalABC.NET константы несколько теряют свою значимость по сравнению с базовым Паскалем (это будет понятно при знакомстве с динамическими массивами в части 5), но их польза несомненна.

Кроме констант, значение которых определяет программист, имеется некоторое количество предопределенных констант, т.е. констант, значения которых компилятору уже знакомо и описывать их не нужно. Для целых типов предопределены следующие константы: MaxShortInt, MaxByte, MaxSmallInt, MaxWord, MaxInt, MaxLongWord, MaxInt64 и MaxUInt64. Их имена получаются путем приписывания Max к имени соответствующего типа, а значения равны максимально допустимым для этого типа значениям в соответствии с таблицей 1.1. Все эти имена помнить совсем необязательно, они есть в Справке, а кроме того, для каждого целочисленного типа определен набор констант вида T.V, где T – тип, а V описывает, что представляет константа:
`T.MaxValue` – максимальное значение типа T;
`T.MinValue` – минимальное значение типа T.

Например, int64.MaxValue, как и MaxInt64, определяет максимальное значение для типа int64, равное 9 223 372 036 854 775 807.

Все константы, о которых говорилось выше, имеют свои имена. Именованные константы удобны тем, что имя запомнить обычно намного проще, чем числовое значение. Например, значительно удобнее использовать константу РадиусЗемли, чем величину 6356863, увидев которую, не сразу догадаешься, что это такое.

```
// p01001
const
  Длина = 400;
  Ширина = 730;
  Высота = 142;

begin
  Println('Объем параллелепипеда равен', Длина * Ширина * Высота)
end.
```

Наряду с именованными константами существуют константы неименованные – ***литералы***. Целочисленный литерал (в переводе с латыни – буквальный) изображает значение числовой величины. В приведенном выше примере 400, 730 и 142 – литералы. В программах литералы используют повсеместно.

Числовое значение представляется литералом в привычном виде: последовательностью цифр, которая может впереди иметь знак плюс или минус. Считается, что число записано в десятичной системе счисления. Можно также записать шестнадцатеричное число, для чего первым символом указывают знак денежной единицы $. В этом случае знак числа писать нельзя, поскольку шестнадцатеричная запись отражает внутреннее представление числа.

Вы ошибетесь, если решите, что целочисленная константа, тип которой не указан явно, получит тип, который достаточен, чтобы вместить значение, указанное литералом. На самом деле, тип будет совпадать с типом литерала, использованным в качестве значения константы, а этот тип определяется в соответствии с таблицей 1.2.

Минимальное значение	Максимальное значение	Тип
-2 147 483 648	2 147 483 647	**integer**
-9 223 372 036 854 775 808	9 223 372 036 854 775 807	**int64**
9 223 372 036 854 775 808	18 446 744 073 709 551 615	**uint64**
< -9 223 372 036 854 775 808	> 18 446 744 073 709 551 615	сообщение об ошибке

Таблица 1.2. Типы данных, назначаемые целочисленным литералам.

Кроме литерала, значение константы может определяться ***арифметическим выражением*** (см. 1.4), результатом вычисления которого должна быть целочисленная величина. В PascalABC.NET при определении значения константы посредством арифметического выражения запрещается использовать пользовательские функции, а значения переменных в выражении должны быть известны компилятору.

Встретив в программе описание константы, компилятор устанавливает ее тип, отводит в памяти место, достаточное для размещения значения константы, вычисляет значение константы и помещает его отведенную память. Вся работа совершается на стадии компиляции, поэтому к моменту начала выполнения программы значения констант уже находятся на отведенных им местах. Этим

объясняется причина запрета попыток изменить значение константы при выполнении программы.

1.3 Переменные

Кроме констант, в программе могут присутствовать величины другого рода, значение которых может изменяться в процессе выполнения программы. Такие величины называются ***переменными***. Каждой переменной программист назначает имя (***идентификатор***), с помощью которого затем оперирует с ней. Как и константа, переменная имеет тип. Типы у целочисленных переменных точно такие же, как у целочисленных констант, поэтому информация из таблицы 1.1. верна и для переменных. Как и константе, компилятор отводит переменной место в памяти компьютера, но содержимое этой области памяти разрешено менять.

Перед тем, как первый раз использовать переменную, ее необходимо описать, дав возможность компилятору установить тип этой переменной.

> PascalABC.NET рекомендует описывать переменные непосредственно перед их использованием, а не в отдельном разделе описания переменных, как это требовалось в базовом Паскале. Переменные, описанные в некотором блоке, за его пределами не существуют.

Описание переменной имеет вид

```
var ИмяПеременной: тип;
```

где тип мы пока будем выбирать из таблицы 1.1.

Если нужно описать несколько переменных одного типа, их имена перечисляются списком через запятую:

```
var ИмяПеременной1, ИмяПеременной2, … ИмяПеременнойN: тип;
```

Недопустимо смешивать в одном списке var описания переменных различных типов.

При описании переменной можно присвоить ей начальное значение (это называется ***инициализацией***), но лишь одной для каждого var:

```
var Имя переменной: тип := значение;
```

Конструкция := в языке Паскаль носит название знака ***операции присваивания***. Она понимается следующим образом: следует вычислить значение выражения, помещенного справа от знака операции присваивания и поместить его в переменную, имя которой указано слева от этого знака.

В случаях, когда тип переменной можно установить из указанного или вычисленного значения, PascalABC.NET позволяет делать автовыведение типа. Чтобы использовать автовыведение типа, в описании переменной тип не указывается. Как и

в случае с константами, тип которых явно не указан, он устанавливается компилятором в соответствии с таблицей 1.2.

```
var Имя переменной := значение;
```

Примеры описаний переменных.

```
var a, b, gamma, w15: integer; // описание переменных списком
var bt: byte; // описание одной переменной
var n: word := 18; // описание типа, совмещенное с инициализацией
var s := 0; // описание с автовыведением типа (integer)
var MyBytes := $C7; // шестнадцатиричное значение с типом integer
```

Имя переменной в PascalABC.NET может иметь практически любую длину и образуется из букв, цифр и знака подчеркивания, но с цифры оно начинаться не может. Прописные и строчные буквы не различаются, поэтому имена proba, PrObA и PROBA эквивалентны. Буквы не обязаны быть только латинскими – допускаются любые буквы Unicode – русские (кириллица), западноевропейская латиница, греческие и т.д. Поэтому появление переменной с именем Уголβ вопросов у компилятора не вызывает. Недопустимо использовать в качестве имен ключевые слова языка, но и тут есть выход: в таких случаях перед именем ставится ***экранирующий символ*** &, например &begin.

При описании переменной целого типа, объединенном с инициализацией, в качестве значения может указываться произвольное арифметическое выражение, результатом вычисления которого должна быть целочисленная величина. Если это условие нарушить, то при наличии описания типа компилятор зафиксирует ошибку, а при отсутствии описания переменная получит не тот тип, который программист ожидал.

Имеется еще одна, мощная разновидность описания переменных, объединенного с присваиванием начального значения и автовыведением типов. Она основана на так называемом ***кортежном присваивании***.

```
var (Имя1, Имя2, … ИмяN) := (Значение1, Значение2, … ЗначениеN);
```

Здесь переменная Имя1 получает Значение1 (и соответствующий тип), Имя2 получает Значение2 и т.д.

Конструкция, записанная в круглых скобках, называется ***кортежем*** (см. 5.1.4), отсюда и происходит название этого вида множественного присваивания.

```
var (a, b, c, i) := (132, -58, 0, 1);
```

Обратите внимание, что указывать тип переменных в кортежном списке нельзя.

Конструкция вида

```
var (a, b, c: int64, i: byte) := (132, -58, 0, 1);
```

будет забракована компилятором. При необходимости явно указать типы можно либо отказаться от кортежного присваивания, либо воспользоваться ***явным приведением типа*** (см. 1.7) в правой части и сделать описание примерно таким:

```
var (a, b, c, i) := (132, -58, int64(0), byte(1));
```

Использование неинициализированных переменных алгоритмически неверно и часто приводит к плохо обнаруживаемым ошибкам. Чтобы помочь начинающим пользователям, в PascalABC.NET любая числовая переменная, которая при описании не получила начального значения, инициализируется нулем. Тем не менее, не нужно на это полагаться Хороший стиль программирования предполагает, что программист сам проводит необходимую инициализацию переменных.

1.4 Арифметические выражения

В языках программирования под термином «выражение» понимают набор из имен переменных, констант, знаков операций, скобок и имен функций. Арифметическое выражение является аналогом математической формулы и результатом его вычисления будет число. Если это число будет целым, то говорят о целочисленном арифметическом выражении. Частными случаями выражения являются уже рассмотренные константы и переменные.

1.4.1 Арифметические операции

Константы и переменные в арифметическом выражении могут связываться между собой при помощи знаков арифметических операций. При этом образуются конструкции вида

```
A ЗнакОперации B
ЗнакОперации B
```

Здесь A и B называются ***операндами***, а знак операции принято называть ***операцией***. Если операция используется с двумя операндами, она называется ***бинарной***. Существуют также операции с одним операндом, называемые ***унарными***. Есть также операция с тремя операндами, она называется **тернарной**, но в PascalABC.NET используется термин ***условная операция*** (см. 3.4).

К арифметическим операциям относятся сложение (знак операции +), вычитание (-) и умножение (*). Привычный знак деления (/) мы пока использовать не будем, потому что результат этой операции в Паскале не является целочисленным. Деление нацело выполняет операция **div**. Еще одна операция, **mod**, дает (в программировании принято говорить «***возвращает***») остаток от целочисленного деления. Арифметическая унарная операция фактически одна – изменение знака

числа, для чего перед операндом указывается знак минус. Может быть также указан и плюс, но он не выполняет никаких действий.

> Операции возведения в степень, возвращающей целочисленный результат, в Паскале нет! Для целых чисел есть только ***функция*** возведения в квадрат Sqr(n). Отметим, что часто бывает удобнее и короче написать n*n, чем Sqr(n). В куб возводим, записывая n*n*n, в четвертую степень можно возвести, записав Sqr(Sqr(n)) и т.д.

Рассмотрим программу.

```
begin
   var (a, b) := (30, 8);
   var c := a + b;
   var d := c * a + 2 * b;
   var (e, f) := (a div b, a mod b);
   var g := -e + f;
   Println(a, b, c, d, e, f, g)
end.
```

Вначале a получает значение 30, b получает значение 8 (тип обоих **integer**). Затем вычисляется значение a + b. 30 + 8 = 38. Переменная c получает значение 38. А что даст автовыведение типа? Складываются значения типов **integer**. Ожидаемо, что тип результата тоже будет **integer**. Далее, 38 * 30 + 2 * 8 = 1156 и это значение запоминается в d. Здесь тоже понятно, какой будет тип – **integer**. Вычисляется выражение a **div** b. 30 делим нацело на 8. Фактически, выделяем целую часть простой дроби 30 / 8, получая число 3, которое помещается в переменную e. Тип будет снова **integer**. Вычисляем a **mod** b. Это остаток от деления 30 / 8. Частное мы уже нашли, оно равно 3, поэтому 30 - 3 * 8 = 6. Значение 6 отправляется в f, по-прежнему неся с собой тип **integer**. Переходим к строке с описанием переменной g. Вот он – унарный минус – стоит перед e. Меняем знак e, получая -3 и это значение складываем со значением f, равным 6, получая в результате 3. Оно отправляется в g, а тип его, конечно же, все тот же: **integer**.

Если написать кортежное присваивание

```
var (c, d) := (a + b, c * a + 2 * b);
```

результат окажется не тем, который ожидается. Здесь d зависит от c, но невозможно угадать, что будет вычисляться раньше, c или d. Поэтому компилятор в данном случае зафиксирует ошибку и выдаст сообщение «Неизвестное имя 'c'».

1.4.2 Приоритет арифметических операций

Когда мы вычисляли значение выражения c*a+2*b, то не задумываясь нашли значений произведений c*a и 2*b, а только затем их сложили между собой. Почему так? Разве нельзя сначала найти значение c*a, прибавить к нему 2, и лишь потом умножить результат на b? Дело в том, что мы еще в начальной школе усвоили

правило арифметики: умножение делается раньше сложения. Чтобы не путаться во множестве правил, в программировании введено понятие ***приоритета операций***.

Приоритет операции – это некоторое целое число. Чем оно меньше, тем приоритет выше, тем раньше выполняется операция. Операции, имеющие одинаковый приоритет, выполняются в естественном порядке следования, слева направо. Для изменения порядка выполнения операций служат круглые скобки, имеющие наивысший приоритет. Знание приоритета операций позволяет избежать нагромождений из скобок и обеспечить правильный порядок выполнения операций.

В описании каждого языка программирования имеется таблица приоритетов всех имеющихся в языке операций. Нам пока достаточно следующих знаний:

- унарные операции + и – , а также функция Sqr() имеют приоритет 1;
- операции *, **div**, **mod** имеют приоритет 2;
- бинарные операции + и – имеют приоритет 3.

Если приоритет нескольких подряд идущих операций одинаков, они выполняются в порядке слева направо. Об этом часто забывают.

1.4.3 Игры с унарным плюсом и минусом

Пусть дана следующая программа

```
begin
   var (a, b, c) := (3, -4, 8);
   var y := -2 * a + 5 * b + - - - + - - + + + - 2 * c;
   Println(y)
end.
```

Является ли такое «дикое» выражение, инициализирующее y, допустимым?

Для ответа на этот вопрос попытаемся разбить это выражение на отдельные операции при помощи расстановки скобок. Также подставим вместо переменных их значения. Не забываем, конечно, о приоритете операций.

```
-2*a+5*b+---+--+++-2*c
((-2)*3) + (5*(-4)) +---+--+++-2*8
(-6)+(-20) + (---+--+++-)2*8 выделяем унарные операции
-26 + (---+--+++-)2*8
Унарные плюсы опускаем – они ничего не меняют
-26 + (------)2*8
Осталось 6 унарных минусов; это три пары. Каждая пара унарных минусов дает
унарный плюс.
(-26) + (+++)8*2
```

```
Снова опускаем унарные плюсы
(-26) + 8*2
(-26) + 16
-26+16
-10
```

Можно утверждать, что переменная y будет инициализирована значением -10 с автовыведением типа **integer**. Кажется сложным? Может быть. Но если понять, как это делается, в дальнейшем не будет проблем, связанных с унарными операциями любого типа и над любыми данными.

Еще один пример: записать в PascalABC.NET выражение вида $\frac{a}{-b} - c + \frac{-d}{-e}$

Это можно записать как a div (-b) – c + (-d) div (-e), но можно и не мудрить со скобками, записав a div -b – c + -d div –e

А как лучше? Лучше со скобками. В большинстве языков недопустимо писать два знака операции подряд, а вы не всегда будете пользоваться только PasacalABC.NET.

1.4.4 Стандартные целочисленные функции

Арифметическое выражение может содержать вызовы ***функций*** (см. 4.1).

Под функцией в программировании понимают некоторый самостоятельный фрагмент программы, имеющий имя и возвращающий результат. К функции можно обращаться из других мест программы (***вызывать*** функцию) путем упоминания ее имени. Часть функций компилятор «знает» и их называют стандартными или встроенными. Другую часть пользователь при необходимости может ***подключить*** из имеющихся внешних файлов-библиотек («модулей»), либо запрограммировать самостоятельно (***пользовательские функции***).

Обращение к функции (ее вызов) состоит в записи имени функции, за которым в круглых скобках следует список передаваемых ей параметров (***аргументов функции***), на основе которых будет вычисляться значение. Найденное значение подставляется на место вызова функции. Параметры отделяются друг от друга запятыми. Если параметров нет, круглые скобки после имени функции тоже можно не указывать.

Рассмотрим некоторые из стандартных функций PascalABC.NET
`Abs(n)` – возвращает абсолютное значение аргумента n
`Max(m, n)` – возвращает максимальное из значений m, n
`Min(m, n)` – возвращает минимальное из значений m, n
`Random(n)` – возвращает целое случайное число из диапазона 0 .. n-1
`Random(m, n)` – возвращает целое случайное число из диапазона m .. n
`Random2(n)` – возвращает кортеж из двух целых случайных чисел в диапазоне 0 .. n;
Random2(m, n) – возвращает кортеж из двух целых случайных чисел в диапазоне m .. n;
Random3(n) – возвращает кортеж из трех целых случайных чисел в диапазоне 0 .. n;

Random3(m, n) – возвращает кортеж из трех целых случайных чисел в диапазоне m .. n;
`Sign(a)` – возвращает -1 при a<0, 0 при a=0 и 1 при a>0;
`Sqr(n)` – возвращает квадрат n (следите за типом, может вернуть **int64** !!!).

Теперь мы можем составлять достаточно сложные целочисленные выражения и проводить с их помощью инициализацию описываемых переменных . Этого вполне достаточно, чтобы начать писать несложные программы, производящие вычисления и выводящие полученные результаты.

```
begin // p01002
  var (Длина, Ширина) := Random2(100, 400);
  var (ПериметрПрямоугольника, ПлощадьПрямоугольника) :=
      (2 * (Длина + Ширина), Длина * Ширина);
  var (Катет1, Катет2) := Random2(10, 90);
  var КвадратГипотенузы := Sqr(Катет1) + Sqr(Катет2);
  var МаксимумТрех := Max(Random(30), Max(Random(25),
      Random(21)));
  Println(Длина, Ширина, ПериметрПрямоугольника,
      ПлощадьПрямоугольника);
  Println(Катет1, Катет2, КвадратГипотенузы, NewLine,
      МаксимумТрех)
end.
```

Вывод будет выглядеть так:

```
373 321 239466 119733
81 88 14305
 29
```

Представим себя на месте пользователя, для которого мы написали такую программу. Как ему понять, где что выведено? Текста программы он не видит, а даже если бы и увидел, текст ему мало что даст. Это приводит нас к пониманию следующего: мало составить алгоритм и написать программу - надо еще позаботиться о том, чтобы эта программа обеспечивала удобный для пользователя ввод и вывод информации.

Набор программных средств и приемов программирования, формирующих взаимодействие пользователя с программой, образует так называемый ***пользовательский интерфейс***. Создание эффективных пользовательских интерфейсов – отдельная область конструирования программ, которая выходит за рамки данной книги. Но какие-то, пусть минимальные, удобства для пользователя программа обеспечить обязана. Например, вывести кроме набора чисел необходимые пояснения. Сделать это совсем несложно, достаточно вставить в нужных местах списка выводимых значений поясняющий текст в одинарных кавычках.

```
Println('Длина=', Длина, 'ширина=', Ширина, 'периметр=',
        ПериметрПрямоугольника, 'площадь=', ПлощадьПрямоугольника);
```

Соответствующая строка примет следующий вид

```
Длина= 276 ширина= 313 периметр= 172776 площадь= 86388
```

Оператор Println чаще используют при выводе значений переменных без поясняющего текста, поскольку он разделяет выводимые элементы пробелом. Если разделители мы хотим указать сами, удобнее использовать оператор Writeln (или Write, если после вывода строку переводить не надо). Этот оператор не выводит ничего помимо указанного в списке.

```
Writeln('Длина=', Длина, ', ширина=', Ширина, ', периметр=',
        ПериметрПрямоугольника, ', площадь=', ПлощадьПрямоугольника);
```

Вывод станет еще нагляднее

```
Длина=352, ширина=299, периметр=210496, площадь=105248
```

Остановимся на том, как работает эта программа, опуская некоторые подробности ввиду достаточно информативных имен переменных.

Функция Random2(100, 400) возвращает кортеж из двух случайно выбранных из диапазона [100;400] целых чисел. Кортежное присваивание инициализирует переменную Длина первым из элементов кортежа, а переменную Ширина – вторым. В следующей строке описаны две переменные, которые также инициализируются кортежным присваиванием. Переменные Катет1 и Катет2 инициализируются кортежным присваиванием на основе функции Random2, а переменная МаксимумТрех при инициализации дважды обращается к стандартной функции Max.

Ниже приведены примеры записи целочисленных арифметических выражений.

Математическая запись	**Запись на PascalABC.NET**	**Ошибочная запись на PascalABC.NET**
$-\frac{a \cdot b}{c \cdot d}$	-(a*b **div** (c*d))	-a*b **div** c*d
$\frac{a+b}{c+d} \times \frac{1}{e}$	(a+b) **div** (c+d) **div** e	(a+b)/(c+d)*1/e
$a^5 + b^8$	a*Sqr(a*a)+Sqr(Sqr(b*b))	

Рассмотрим текст еще одной программы, а потом сравним его с программой на базовом Паскале. Чтобы не было «мучительно больно» за потраченное на изучение PascalABC.NET время.

```
begin
   var (a, b) := Random2(-5, 20);
   Println(a, b);
   Writeln('Минимум=', Min(a, b))
end.
```

При всей своей краткости, приведенная программа делает не так уж мало. Вначале в кортежном присваивании переменные a и b инициализируются с помощью функции получения двух случайных чисел в диапазоне [-5; 20]. Полученные значения выводятся на монитор. Затем вычисляется и выводится минимальное значение из a и b.

Вывод выглядит следующим образом

```
11 -4
Минимум=-4
```

Теперь рассмотрим аналогичную программу на базовом Паскале. Конечно, ее тоже можно набрать и выполнить в PascalABC.NET, поскольку его язык обеспечивает высокую совместимость с базовым Паскалем.

```
var
  a, b, min: integer;

begin
  Randomize;
  a := Random(26) - 5;
  b := Random(26) - 5;
  Writeln(a, ' ', b);
  if a < b then min := a
  else min := b;
  Writeln('Минимум=', min)
end.
```

1.5 Оператор присваивания

Говоря об описании переменных, объединенном с их инициализацией, удобно попутно познакомиться с оператором присваивания. Он получается, если убрать из описания переменной с автовыведением типа служебное слово **var**.

Оператор присваивания используется для ранее описанных переменных и позволяет присвоить им некоторое значение. Его структура очень проста

```
Имя переменной := выражение;
```

Есть и кортежное присваивание

```
(Имя1, Имя2, … ИмяN) := (Выражение1, Выражение 2, … Выражение N);
```

Работа оператора присваивания состоит в вычислении значения выражения в правой части и присваивания этого значения имени переменной в левой части. Тип значения, полученного в выражении, должен совпадать или быть ***автоматически***

приводимым к типу переменной. Это означает, что мы не обязаны указывать в программе, как именно производить преобразование типа.

> Для целочисленных значений все типы, кроме BigInteger, автоматически приводимы путем копирования нужного количества байт без какого-либо анализа и это может оказаться неисчерпаемым источником труднонаходимых ошибок.

Например, корректно написать

```
var a := 35*40+283; // значение 1683, тип integer, 4 байта
var b: int64 := a*a; // значение 2832489, тип int64, 8 байт
```

Но вот пример, если сказать мягко, «неожиданности»:

```
begin // p01003
  var a: int64 := 546 * 546 * 546; // должно быть 162 777 336
  a := a * a; // должно быть 26 494 507 823 224 896
  var b: integer := a * a; // то же, что и выше, но 4 байта мало
  Println(a, b)
end.
```

При выводе получаем

```
26494507823224896, -1259597824
```

Первое значение совершенно правильное – это $(546^3)^2 = 546^6$, а второе – результат попытки поместить это значение в переменную, для которой отведено количество памяти, недостаточное для размещения такого числа. В погоне за эффективностью компиляторы строят программу, бесконтрольно пересылающую нужное количество байт и в данном случае из восьми байт будут пересланы только четыре.

Ошибки, связанные с потерей разрядов в целочисленной арифметике, к сожалению, нередки и трудно находимы. Об этом начинающему программисту нужно помнить и быть особенно внимательным, используя «короткие» типы данных длиной один и два байта. В PascalABC.NET вероятность ошибок несколько снижена вследствие использования по умолчанию типа **integer** с длиной четыре байта, а вот в базовом Паскале тип **integer** был двухбайтным (из-за 16-битной архитектуры компьютеров прошлых лет), что нередко порождало ошибки вследствие выхода значений за величину 32767.

Но нет ли тут нарушения? Нет. В стандарте языка Паскаль (есть и такой, но строго им никто не пользуется) сказано, что представление типа **integer** должно обеспечить размещение целочисленных значений в диапазоне -32768 ... 32767. Но это вовсе не является запретом на размещение в типе **integer** значений большей величины.

1.6 «Сюрпризы» целочисленной арифметики

При вычислении выражений компилятор заводит внутренние переменные в соответствии с типом операндов. Рассмотрим пример с литералами.

```
begin
  var a: int64 := 125 * 125 * 125 * 125 * 125;
  Println(a)
end.
```

Полученное значение 125^5 = 30 517 578 125 вполне укладывается в диапазон **int64**, но при попытке компиляции этой программы мы получим сообщение «Переполнение в арифметической операции». Объяснение причины достаточно простое. Литерал 125 оценивается, как имеющий тип **integer**. Каждая операция умножения строится по схеме **integer** := **integer** * **integer**. На последнем (четвертом) умножении значение произведения выйдет за пределы MaxInt = 2 147 483 647, что обнаружится компилятором. Что же, в этот раз повезло! Но вот другой пример, с переменными.

```
begin
  var n := 125;
  var a: int64 := n * n * n * n * n;
  Println(a)
end.
```

В отличие от предыдущего примера, вместо литерала здесь используется переменная. И при компиляции переполнение не обнаруживается, поскольку компилятор «не знает» значения n – оно появится только после выполнения присваивания, что произойдет только на стадии выполнения программы. К сожалению, при выполнении программы переполнение в целочисленной арифметике не контролируется – так уж устроены процессоры персональных компьютеров. В результате получим неверный результат 452807053. Хорошо еще, если это и вся программа. А если нет и найденное значение a, не выводясь, используется в дальнейших вычислениях? Получаем труднонаходимую ошибку.

И еще один пример. Теперь объявим n не переменной, а константой.

```
const
  n = 125;

begin
  var a: int64 := n * n * n * n * n;
  Println(a)
end.
```

Компилятор подставляет вместо константы литерал 125. Результат мы уже обсудили раньше: сообщение о переполнении в арифметической операции на стадии компиляции. Снова компилятор нас уберег от ошибки. Убедились, что константы бывают полезными?

1.7 Явное приведение типа

Бывают случаи, когда программисту приходится принимать некоторые дополнительные меры к тому, чтобы результат вычисления выражения был верным.

Пусть требуется выполнить такой оператор присваивания

```
b := 152 * (integer.MaxValue + 2344322) * (23432 + 12312312);
// результат 4 030 998 575 382 838 272
```

Анализ выражения показывает, что будет получено значение, на порядки превышающее **integer**.MaxValue. Следовательно, для переменной b нам понадобится, как минимум, тип **int64**. Первая попытка начинающего программиста - решить задачу «в лоб».

```
begin
  var b: int64 := 152 * (integer.MaxValue + 2344322) * (23432 + 12312312);
  Println(b)
end.
```

При запуске такой программы будет получено уже известное нам сообщение компилятора «Переполнение в арифметической операции». Но непонятно, где такое переполнение возникает. К счастью, в программистском арсенале есть прием, называемый по известной с древних времен фразе «Разделяй и властвуй». Разобьем вычисление на части.

```
begin
  var p1: int64 := integer.MaxValue + 2344322;
  p1 := 152 * p1;
  var p2 := 23432 + 12312312;
  p1 := p1 * p2;
  Println(p1)
end.
```

И снова получаем то же самое сообщение компилятора, но теперь видно, что оно относится к оператору `var p1: int64 := integer.MaxValue + 2344322;`

Причина понятна. Константа **integer**.MaxValue имеет тип **integer**, литерал 2344322 также получает тип **integer** и это позволяет компилятору диагностировать переполнение в операции сложения.

Ситуацию можно попробовать спасти, объявив переменную (или константу) с типом **int64** и значением 2344322. Тогда компилятор будет иметь дело со сложением значений типов **integer** и **int64** и переполнения возникнуть не должно, поскольку в подобных операциях тип результата определяется типом операнда, занимающим большее количество памяти.

```
begin
  var c: int64 := 2344322;
  var p1 := integer.MaxValue + c;
  p1 := 152 * p1;
  var p2 := 23432 + 12312312;
  var b := p1 * p2;
  Println(b)
end.
```

Наша программа заработала. Более того, она выдала верный результат. Если цель была лишь в том, чтобы один раз произвести вычисление, можно на этом и закон-

чить. Но мы попробуем улучшить программу, сделав ее нагляднее, т.е. постараемся максимально приблизить запись текста программы к исходной формуле. Это одно из проявлений хорошего стиля в программировании. Если даже через много лет понадобится внести изменения в программу, связанные с изменением формулы, вы сразу найдете нужное место. Будем считать, что место, вызывавшее проблему, найдено. И вернемся к первому варианту программы, лишь немного его подправив.

```
begin
  var c: int64 := 2344322;
  var b := 152 * (integer.MaxValue + c) * (23432 + 12312312);
  Println(b)
end.
```

И это тоже работает! Работает потому, что мы внедрили в арифметическое выражение одну величину типа **int64**, которая потребовала от компилятора сделать автоматическое приведение типов к **int64** во всех операциях, в которых хотя бы один операнд имеет такой тип.

Чтобы по-настоящему развязать руки программисту, в PascalABC.NET имеется возможность в нужный момент явно приводить тип к необходимому. Для этого используется запись вида **тип** (значение). Это не единственный способ, но пока будем пользоваться им.

```
begin
  var b := 152 * (integer.MaxValue + int64(2344322)) * (23432 + 12312312);
  Println(b)
end.
```

Не правда ли, эта программа похожа на ту, которую пытался написать начинающий программист? Только в отличие от той программы, она правильно работает. Также, можно было написать **int64**(**integer**.MaxValue) + 2344322).

Явное приведение типа – мощный инструмент в руках опытного программиста. Вы избежите многих проблем, если будете в нужное время вспоминать об этой возможности помочь компилятору.

1.8 Ввод целочисленных данных

До сих пор мы писали программы, в которых все необходимые данные были известны заранее. Но так бывает очень редко. Типичная ситуация требует сделать нужные вычисления при заданных значениях некоторых данных. Например, мы хотим найти значение площади прямоугольника, длины сторон которого нужно указать в процессе работы программы.

Инициализация переменных или присваивание им некоторых значений в коде программы проблемы не решат. Нужна операция, позволяющая ввести нужные значения в процессе работы программы. Такой процесс называется ***вводом***.

PascalABC.NET предлагает совмещать описание переменной с вводом ее значения:

```
var ИмяПеременной := ReadInteger('Текст приглашения ко вводу');
```

Если приглашение не нужно, оператор выглядит еще проще

```
var ИмяПеременной := ReadInteger;
```

Имеется также вариант оператора с ReadlnInteger, но о нем мы вспомним в части 6, когда будем изучать ввод символьных данных. Пока лишь отметим, что поскольку символьные данные тут не вводятся, достаточно писать ReadInteger.

Если переменная была описана ранее, ключевое слово **var** указывать нельзя – повторное описание переменных язык запрещает.

Так вводится значение одной переменной. А если их больше? Вспомним кортежное присваивание: оно и тут имеется, только в ограниченном количестве – для двух и трех переменных.

```
var (имя1, имя2) := ReadInteger2('Текст приглашения ко вводу');
var (имя1, имя2, имя3) := ReadInteger3('Текст приглашения ко вводу');
```

В Паскале операция ввода с клавиатуры непременно должна завершиться нажатием клавиши Enter. Вводимые данные могут разделяться пробелами, но можно также в ходе ввода использовать Enter – это решает пользователь программы.

Универсальная программа для расчета площади прямоугольника могла бы выглядеть так

```
begin // p01004
  var (a, b) := ReadInteger2('Введите длину и ширину прямоугольника:');
  Println('Площадь прямоугольника равна', a * b)
end.
```

ReadInteger2 осуществляет прием с клавиатуры двух значений типа **integer** (это подсказывает цифра 2 в его названии), а кортежное присваивание помещает эти значения в переменные a и b соответственно. Обратите внимание, что в операторе вывода указано не имя переменной, а выражение. Это нормальная практика. Если значение выражения требуется в программе единственный раз, совершенно излишне заводить переменную и делать ей присваивание – такие действия лишь тратят память компьютера и увеличивают время работы программы.

В базовом Паскале оператор ввода имел вид

```
Read(список переменных, разделенных запятыми);
```

Такой оператор полноценно поддерживается и в PascalABC.NET. Более того, можно привести примеры, когда он необходим. Это и ввод значений в количестве, большем трех, ввод значений переменных с типом, отличным от **integer**, ввод разнотипных значений и некоторые другие случаи.

```
begin
  var a: byte;
  var b, c: int64;
  var d: integer;
  Print('Введите значения a,b,c,d:');
  Read(a, b, c, d);
  …
end.
```

1.9 Инкремент и декремент

В ряде алгоритмов требуется изменять значение переменной на определенную величину, чаще всего, на единицу. Если эта величина положительная, говорят об ***инкременте*** (по-английски increment – увеличение), если отрицательная – о ***декременте*** (decrement – уменьшение).

Инкремент и декремент можно реализовать операцией присваивания, например

```
i := i + 1;
j := j - 2;
```

С точки зрения математики написаны абсурдные вещи. Но знак := – не знак равенства, это знак операции присваивания. Вначале вычисляется значение правой части и только потом оно запоминается в переменной, указанной в левой части. К текущему значению i прибавляется единица, и новое значение запоминается в качестве i. А новое значение j окажется уменьшенным на два.

В языке Паскаль для таких операций есть более короткая запись:

```
Inc(i); // инкремент 1
Dec(j, 2); // декремент 2
```

Общий вид записи операций инкремента и декремента следующий

```
Inc(ИмяПеременной, ЗначениеИнкремента);
Dec(ИмяПеременной, ЗначениеДекремента);
```

Если значение инкремента или декремента равно единице, его можно не указывать.

Еще одну реализацию инкремента и декремента предоставляет разновидность оператора присваивания, позаимствованная из языков семейства C. Присваивание, соединенное с инкрементом или декрементом, можно записать в виде

```
i += 1;
j -= 2;
```

Операция += увеличивает значение переменной, указанной в левой части на величину, указанную в правой части. Операция -= уменьшает значение переменной, указанной в левой части на величину, указанную в правой части.

Поскольку инкремент и декремент реализуются посредством целочисленных операций сложения и вычитания, при их выполнении могут возникать ошибки, связанные с выходом значений за отведенные границы типа (***переполнение разрядной сетки***). Отметим, что также есть операция *=, работающая аналогично +=, но производящая не сложение, а умножение.

1.10 Для самостоятельного решения

T1.1. Имеется оператор

```
v := shortint.MaxValue.
```

Какой тип и значение получит переменная v, если к ее исходному значению 1154 раза прибавить единицу? Вопрос не предполагает использования компьютера.

T1.2. Определите тип и значение результата вычисления выражения

```
smallint.MinValue + shortint.MaxValue
```

Вопрос не предполагает использования компьютера.

T1.3. Пусть требуется покрасить стены помещения, имеющего размеры Длина, Ширина и Высота, заданные в сантиметрах. Помещение имеет дверной проем (ширина 92 см, высота 210 см) и оконный проем (ширина 207 см, высота 150 см). Предложите программу, вычисляющую количество банок краски и стоимость их приобретения, если расход краски составляет 120 г. на 10 000 см^2, а банка содержит 3800 г. краски. Размеры помещения и стоимость банки краски (в пределах 1800 .. 2400 руб) задаются путем ввода с клавиатуры.

Часть 2

Вещественные числа

Господь сотворил целые числа; остальное — дело рук человека.
Леопольд Кронекер,
немецкий математик

Математически, множество вещественных (или действительных) чисел $\mathbb{R}$ образуется из множества целых $\mathbb{Z}$ путем присоединения к нему нецелых значений. В вычислительной технике и программировании под вещественными числами понимают подмножество рациональных чисел $\mathbb{Q}$, представляющих собой значения, которые можно выразить обыкновенной дробью m/n. Речь идет именно о подмножестве, поскольку под хранение данных такого типа отводится определенное фиксированное количество машинной памяти. Из-за этого ограничения большая часть рациональных чисел представляется приближенными с некоторой точностью значениями (в этом случае мы говорим о ***машинной точности***). Конечно, если нужна абсолютная точность, можно создать собственный тип данных, в которых числитель и знаменатель дроби хранятся в виде пары целочисленных значений, и описать набор операций над ними, но такие типы будут рассматриваться позднее. Отметим лишь, что библиотека численных методов NumLibABC, входящая в состав PascalABC.NET, включает класс для работы с обыкновенными дробями, реализованный именно описанным выше способом.

Формат объявления констант и переменных вещественного типа не отличается от формата для целочисленных типов.

2.1 Типы вещественных чисел

PascalABC.NET предоставляет три типа вещественных чисел: **single**, **real** (и его синоним **double**) и **decimal** (все цифры, хранимые в этом типе, точные).

Длина, байт	Тип и количество значащих цифр	Диапазон представляемых значений
4	**single** 7-8	-1.8×10^{308} .. 1.8×10^{308}
8	**real (double)** 15-16	-3.4×10^{38} .. 3.4×10^{38}
16	**Decimal** 28-29	-79228162514264337593543950335 .. 79228162514264337593543950335

Таблица 2.1. Вещественные типы данных

Тип **decimal** достаточно специфичен, поэтому его пока мы рассматривать не будем. В частности, для него не определено большинство стандартных математических функций.

Для вещественных типов определены стандартные константы MaxReal (MaxDouble) и MaxSingle, представляющие собой максимальные значения, которые может хранить соответствующих тип данных. Эти значения указаны в таблице 2.1. Кроме того, определены константы вида T.V, где T принимает значения типа **single**, **real** или **double**, а V описывает, что собой представляет константа.

`R.MaxValue` – максимальное значение типа R;
`R.MinValue` – минимальное значение типа R;
`R.Epsilon` – самое маленькое положительное значение типа R;
`R.NaN` – значение «не число» (Not a Number), которое получается, например, при делении 0/0;
`R.NegativeInfinity` – «минус бесконечность», которое получается при делении отрицательного числа на ноль, например, -2/0;
`R.PositiveInfinity` – «плюс бесконечность», которое получается при делении положительного числа на ноль, например, 2/0.

Имеются и другие константы, например, число π: `Pi = 3.141592653589793` и основание натурального логарифма e: `E = 2.718281828459045`.

2.2 Литералы вещественного типа

Литерал изображает в программном коде некоторое значение при помощи символов. Вещественное число, может иметь одну из следующих форм записи:

- m.n – привычный формат записи числа, имеющего целую часть m и дробную n, например, 3.14, -1000.2643, 0.00002432234;
- m.nEk – «научный формат» записи числа, в котором латинская буква E (или e) отделяет мантиссу m.n от десятичного порядка k. Мантисса может быть записана в виде целого числа – тогда литерал имеет вид mEk. Примеры записи литералов в научном формате:

```
3E6 // 3×10^6
5.7e-11 // 5.7×10^-11
-0.000234e-7 // -0.000234×10^-7 = -0.234×10^-10
```

2.3 Арифметические выражения вещественного типа

Правила составления и записи арифметических выражений вещественного типа такие же, как для целочисленных выражений. Любые части целочисленных выражений могут входить в вещественное выражение. Разница имеется лишь в некоторых операциях.

2.3.1 Деление и возведение в степень

Операция деления бинарная и записывается при помощи символа «слэш» (/). Операция q/r возвращает результат деления q на r вещественного типа. Например, 5/8 возвращает вещественное значение 0.625, несмотря на то, что литералы 5 и 8 изображают целые числа. Здесь произойдет автоприведение типа, причем для операции деления оба целочисленных операнда приводятся к типу **real**.

Операция деления имеет приоритет 2, такой же, как операции *, div, mod.

Операция возведения в степень также бинарная; ее знак операции ** представляет собой пару знаков операции умножения, следующих друг за другом без пробела, а приоритет равен 1. Запись вида a**b обозначает операцию a^b. Реализация этой операции в PascalABC.NET базируется на стандартных средствах платформы Microsoft .NET Framework (System.Math.Pow), что накладывает некоторые ограничение: при отрицательном основании степени показатель может быть только целочисленным. Это легко объяснить. Когда a и b вещественные, возведение в степень производится в соответствии с известным тождеством

$$a^b = e^{b \cdot \ln a}$$

Понятно, что при неположительном значении переменной a невозможно вычислить логарифм. Тем не менее, на случай такого некорректного возведения в степень предусмотрено получение результата со значением NaN (Not a Number).

```
begin // p02001
  var x := 5 ** 8;
  var b := 3.5 ** 2.9;
  Println(x, b, x ** b, 3 ** 0, 4 ** (-3), (-4.2) ** (-1.12))
end.
```

Приведенная программа выдаст следующий результат

```
390625 37.826601883412 3.2906421758964E+211 1 0.015625 NaN
```

Здесь также имело место автоприведение целочисленных литералов к типу **real**.

> В бинарных вещественных операциях и при обращении к функциям, требующим вещественные параметры, значения всех целых типов, кроме BigInteger, будут автоматически приведены к типу **real**.

2.3.2 Приведение типа

Приведение целого типа к вещественному в худшем случае приводит к потере точности в представлении числа.

```
begin
  var i := 1234567891011213141;
  var a := i + 0.0;
  Writeln(i:21, a:21);
end.
```

Получим следующий результат (подчеркнутые цифры потеряны):

```
1234567891011213141 1.23456789101121E+18
```

Автоприведение значения вещественного типа к целым типам запрещено и попытка его организовать приведет к выдаче сообщения об ошибке при компиляции. Это понятно: можно потерять дробную часть полностью. Программист обязан указать, как именно производить такое приведение: отбросить дробную часть или округлить значение до целого по правилам арифметики. Отбрасывает дробную часть функция Trunc(a), округляет – функция Round(a).

```
begin
  var (a, b, c, d) := (-12.87, 12.87, -5.2, 5.2);
  Println(Trunc(a), Round(a));
  Println(Trunc(b), Round(b));
  Println(Trunc(c), Round(c));
  Println(Trunc(d), Round(d))
end.
```

```
-12 -13
12 13
-5 -5
5 5
```

Разберитесь, почему результаты получились именно такими.

> Функция Round(a) реализует так называемое «банковское» округление. Если вещественное значение находится точно посередине между двумя целыми, то округление осуществляется к ближайшему четному целому значению. Например, `Round(2.5)=2, Round(3.5)=4`.

Тип можно приводить явно подобно тому, как это делалось для целых типов. Например, **single**(1 / 3), **real**(6.18) и т.п. Это требуется редко, поскольку автоприведение производится к типу **real**, а тип **single** не дает ничего, кроме потери точности.

2.3.3 Об арифметическом переполнении

В отличии от ситуации с переполнением в целочисленной арифметике, аппаратная часть компьютера умеет сообщать о переполнении при выполнении операций с плавающей точкой (так именуются в архитектуре компьютеров форматы представления вещественных данных). Соответственно, программа также умеет реагировать на такие переполнения. Эта реакция состоит в возвращении в качестве результата переполнения констант NaN, NegativeInfinity или PositiveInfinity. Значения таких констант могут быть операндами выражений.

```
begin
  var a := 0 / 0;
  var b := 2 / a;
  var c := 1 / 0;
  var d := 1 / c;
  var e := a * c;
  Println(a, b, c, d, e)
end.
```

```
NaN NaN Infinity 0 NaN
```

Проанализируем полученные результаты. 0 / 0 дает NaN – это понятно, результат операции не является числом. 2 / NaN также дает NaN, что тоже понятно. 1 / 0 дает бесконечность (∞ – Infinity) и это не вызывает сомнений. 1 / ∞ – конечно же, ноль. NaN * ∞ – наверное, NaN тут лучшее, что можно придумать.

Существуют логические функции, позволяющие обнаруживать различные переполнения и в зависимости от результатов направлять дальнейший ход работы программы. Об этом – в следующих частях. Пока достаточно помнить, что переполнение при работе с вещественными числами само по себе не приводит к аварийному завершению программы.

2.3.4 Некоторые математические функции

Бибилиотека PascalABC.NET насчитывает большое количество математических функций, работающих с вещественным типом данных. Подробно об этом можно узнать в Справке по PascalABC.NET. Приведем часть из них. В описании функций a и b представляют вещественные значения, n – целочисленные значения.

`Abs(a)` – возвращает абсолютное значение a;
`ArcSin(a)` – возвращает арксинус a в радианах. Есть также `ArcCos` и `ArcTan`;
`Ceil(a)` – возвращает ближайшее большее a целочисленное значение;
`Cos(a)` – возвращает угла a, заданного в радианах. Есть также `Sin(a)`, `Tan(a)`;
`DegToRad(a)` – возвращает радианную меру угла a, заданного в градусах;
`Exp(a)` – возвращает e^a;
`Floor(a)` - возвращает ближайшее меньшее a целочисленное значение;
`Frac(a)` – возвращает дробную часть a;
`Int(a)` – возвращает целую часть a;
`Ln(a)` – возвращает натуральный логарифм a. Есть синоним `Log(a)`;
`Log10(a)` – возвращает десятичный логарифм a;
`Log2(a)` – возвращает логарифм a по основанию 2;
`LogN(a,n)` – возвращает логарифм a по основанию n;
`Max(a,b)` – возвращает максимальное из значений a и b;
`Min(a,b)` – возвращает минимальное из значений a и b;
`RadToDeg(a)` – возвращает значение угла a, переведенное из радиан в градусы;
`Random` – возвращает случайное число на интервале [0;1];
`Random(a,b)` – возвращает случайное число на интервале [a;b];
`Round(a)` – возвращает целое значение a;
`Round(a,n)` – возвращает значение a, округленное до n знаков после запятой;
`Sign(a)` – возвращает значение -1.0, 0.0, 1.0 в зависимости от знака числа;
`Sinh(a)` – возвращает гиперболический синус a. Есть также `Cosh(a)`, `Tanh(a)`;
`Sqr(a)` – возвращает a^2;
`Sqrt(a)` – возвращает $\sqrt{a}$;
`Trunc(a)` – возвращает целое значение a, отсекая дробную часть.

2.4 Точность машинной арифметики

Поскольку вещественные значения представляются в компьютере с некоторой конечной точностью, в ряде случаев приходится принимать определенные меры с тем, чтобы получить приемлемый результат. Наибольшее число проблем порождают операции, многократно увеличивающие результат вычитания близких по

значению чисел. Предположим, мы хотим вычислить значение следующего выражения, равное 10^{20}

$$\frac{1}{\left(\frac{1}{3}+10^{-20}\right)-\frac{1}{3}}$$

Если запрограммировать выражение «как есть», получим следующий код:

```
begin
  Println(1 / ((1 / 3 + 1e-20) - 1 / 3))
end.
```

При выполнении этой программы выводится значение Infinity. Значение ⅓ будет вычисляться в представлении типа real, т.е. иметь до 16 верных цифр. В этих условиях сложение с величиной 1e-20, представляющей собой единицу в двадцатом после запятой разряде, не изменит вычисленного значения. Последующее вычитание ⅓ даст в результате ноль. Деление единицы на ноль, конечно же, даст бесконечность.

Если раскрыть скобки в знаменателе, это ничего не даст: приоритет операций сложения и вычитания одинаков, так что сначала будет выполнено сложение, а затем вычитание. Но если принудительно поменять местами члены выражения, можно получить правильный результат 1e+20.

```
begin
  Println(1 / (1 / 3 - 1 / 3 + 1e-20))
end.
```

Приведенный пример показывает, что при программировании даже внешне простых выражений можно получить неожиданные проблемы, если не учитывать особенностей работы с вещественными числами.

2.5 Вывод вещественных значений

Оператор Print(a) выводит результат с 15 значащими цифрами после запятой, либо с меньшим числом знаков, если число имеет точное машинное представление.

```
begin
  Println(1 / 8, 0.000001, 352 / 1231234, 2 * 124534 + 4563325351.8)
end.
```

```
0.125 1E-06 0.000285892040018388 4563574419.8
```

Если нужно управлять представлением выводимых значений, используется оператор базового Паскаля Write(a:i:j), где a – выводимое значение вещественного типа, i – общее количество позиций, отводимых под вывод, j – количество позиций, отводимых под знаки в дробной части числа. При i=0 количество позиций под вывод определяется автоматически, при j=0 дробная часть не выводится. Отметим, что при выводе число округляется до j знаков после запятой.

```
begin
  Writeln(1 / 8:0:5, 0.000001:10:7, 352 / 1231234:10:5,
      2 * 124534 + 4563325351.8:16:3)
end.
```

```
0.12500 0.0000010   0.00029  4563574419.800
```

2.6 Ввод вещественных значений

PascalABC.NET при вводе значений типа **real** предлагает совмещать описание переменной с вводом ее значения:

```
var ИмяПеременной := ReadReal('Текст приглашения ко вводу');
```

Если приглашение не нужно, оператор выглядит еще проще

```
var ИмяПеременной := ReadReal;
```

Имеется также вариант оператора с ReadlnReal, но поскольку символьные данные тут не вводятся (о них рассказывается в части 6), достаточно писать ReadReal.

Если переменная была описана ранее, ключевое слово **var** указывать не нужно.

Как и при вводе данных типа **integer**, можно вводить значения двух и трех переменных.

```
var (имя1,имя2) := ReadReal2('Текст приглашения ко вводу');
var (имя1,имя2,имя3) := ReadReal3('Текст приглашения ко вводу');
```

Вещественные значения также можно вводить при помощи оператора базового Паскаля Read, указывая в списке ввода переменные вещественного типа. Формат оператора ввода не отличается от рассмотренного в части 1 для целочисленных типов.

2.7 Обращение к системным библиотекам .NET

В состав функций для работы с вещественными значениями PascalABC.NET не вошла часть функций, доступных в среде Microsoft .NET Framework. В частности, может быть полезной функция Atan2(y, x), возвращающая значение угла в радианах, тангенс которого равен отношению y / x. Другая интересная функция, IEEEReminder(x, y), возвращает значение x – y × Q, где Q – округленный до ближайшего целого результат деления x / y (если x / y находится на равном расстоянии от двух целых чисел, выбирается четное число). Вызов этих функций производится в виде System.Math.Atan2(…) и System.Math.IEEEReminder(…) соответственно.

2.8 Для самостоятельного решения

T2.1. Решите задачу T1.3, задавая размеры в метрах и вес в килограммах. Нехваткой краски до 0,1 кг пренебречь (1.1 кг считать за 1 кг, 2.1 кг считать за 2 кг, …).

T2.2. К бассейну подходят две трубы. Первая наполняет его за t1 часов, а вторая – за t2 часов. За сколько часов наполнят бассейн обе трубы, наполняя бассейн одновременно? Отметим, что если время наполнения задать отрицательным, то это будет означать опорожнение бассейна. Вы можете воспользоваться готовой формулой:

$$\frac{1}{t}=\frac{1}{t_1}+\frac{1}{t_2}$$

Эта же формула (следовательно, и программа) пригодна для расчета времени, за которое два землекопа выкопают канаву, а также для расчета общего электрического сопротивления при параллельном соединении проводников или общей электрической емкости при последовательном соединении конденсаторов.

Часть 3

Логика в программе

Машинная программа выполняет то, что вы ей приказали делать, а не то, что бы вы хотели, чтобы она делала (третий закон Грида).

Артур Блох. Законы Мерфи

Программа – это разновидность записи алгоритма на некотором языке, которую компилятор умеет понимать с целью превращения программы в машинный код. Алгоритм строится на основе модели, с определенной степенью достоверности отображающей реальный мир. А мир неоднозначен, каждое мгновение нам приходится делать какой-то выбор. Поэтому язык программирования должен включать средства для организации логических действий.

Необходимая гибкость в реализации логики достигается за счет введения в Паскаль данных логического (булевского) типа - **boolean**.

3.1 Логический тип данных

Вещественные и целые типы относятся к числовым. Для логического типа точно так же можно описывать логические переменные, инициализировать их логическими значениями и использовать автовыведение типа. Логических констант в языке Паскаль две: True (истина) и False (ложь). Ими обычно и выполняют инициализацию. Но можно также инициализировать переменную результатом вычисления целого логического выражения. Если при описании переменная не получила начального значения, ей будет присвоено значение False, но писать программы, полагаясь на такую инициализацию – это дурной вкус.

```
begin // p03001
  var a, b: boolean;
  var c := True;
  var (d, e, f) := (False, True, False);
  Print(a, b, c, d, e, f)
end.
```

В результате работы этой программы будет выведена строка

```
False False True False True False
```

3.2 Логические выражения

Логическое выражение, подобно арифметическому выражению, является неким аналогом математической формулы и результатом его вычисления будет одна из двух логических констант – истина или ложь.

Логические выражения могут строиться:
- на основе логического выражения с предшествующей ему унарной операцией логического отрицания;
- на основе двух арифметических выражений, связанных операцией отношения;
- на основе двух логических выражений, связанных логической операцией.

3.2.1 Операции отношения

Операции отношения хорошо известны из математики. Их шесть: равно « = », не равно « <> », меньше « < », меньше или равно « <= », больше « > » и больше или равно « >= ». Если с помощью одной из этих операций связать два арифметических выражения, мы получим ***утверждение***, которое будет истинным, либо ложным. Еще раз вспомним, что числовые константы – это тоже частный случай арифметического выражения и рассмотрим несколько примеров.

6 < 9.2 – это истинное утверждение, поэтому значением логического выражения будет True.
3 = 5 – это ложное утверждение и значением логического выражения будет False.
2 * 2 = 5 – здесь также получим False
5 * 8 > (10 - 7) * 8 – здесь получаем True (5 * 8 = 40, 10 – 7 = 3, 8 * 3 = 24, 40 > 24 – истина)

Приоритет операций отношения ниже, чем у арифметических операций, что позволяет записывать утверждения без дополнительных скобок. Всегда сначала будут вычислены значения арифметических операций, а потом выполнится операция отношения.

3.2.2 Логические операции

В булевой алгебре определены 16 логических операций, но в языке Паскаль их только четыре. Остальные операции при необходимости легко моделируются.

Унарная логическая операция **not**, называемая логическим отрицанием («НЕ», инверсия) и указываемая перед логическим выражением, позволяет получить его противоположное значение. Эта операция превращает True в False и False в True. Приоритет операции **not** максимально высокий и равен 1 (такой же, как для унарных плюса и минуса), поэтому следующее за операцией **not** логическое выражение не забываем заключать в скобки, если оно содержит еще какие-либо операции.

not ((3 + 8) >= 5) вернет False, поскольку 3 + 8 = 11, значение отношения 11 >= 5 истинно и его инверсия дает значение ложно.

Бинарная логическая операция **or** называется логическим сложением («ИЛИ», дизъюнкция). Ее приоритет равен 3 (как у арифметического сложения или вычитания), поэтому и тут помним о скобках. Результатом операции **or** будет False лишь тогда, когда оба логических выражения имеют значения False, а прочих случаях результатом будет True.

(3 + 6 > 12) **or** (2 * 2 = 4) – результатом будет True. 3 + 6 = 9, отношение 9 > 12 ложно, поэтому первым операндом (слева от **or**) будет False. 2 * 2 = 4, отношение 4 = 4 истинно, вторым операндом будет True. Поскольку один из операндов равен True, результатом будет True.

Бинарная логическая операция **and** называется логическим умножением («И», конъюнкция). Ее приоритет равен 2 (он такой же, как у арифметического умножения или деления), так что по-прежнему помним о скобках. Результатом операции **and** будет True лишь тогда, когда оба логических выражения будут иметь значения True, а прочих случаях результатом будет False.

(2 > 5) **and** (3 * 3 = 9) – результатом будет False. Отношение 2 > 5 ложно и первым операндом будет False. 3 * 3 = 9, отношение 9 = 9 истинно, вторым операндом будет True. Поскольку один операндов равен False, результатом будет False.

Все это нетрудно запомнить, если понять, что логическое сложение и умножение работают почти по тем же правилам, что и их арифметические собратья, да и приоритет выполнения операций у них такой же. Полагая, что False это ноль, а True – единица, можно рассуждать следующим образом:
- сложение целых чисел, принадлежащих множеству {0,1}, дает 0 только тогда, когда оба числа равны нулю. Точно так же, логическое сложение дает результат False только когда оба операнда равны False;
- произведение двух целых чисел, принадлежащих множеству {0,1}, дает 1 только тогда, когда оба операнда равны 1. И логическое умножение дает результат True только когда оба операнда равны True.

Последняя из логических операций – это бинарная **xor**, «сложение по модулю два», она же – «исключающее ИЛИ». У нее такой же приоритет 3, как у **or**. Посмотрим, что «исключает» эта операция, сравнив ее с операцией **or** в таблице истинности, содержащей результаты для всех четырех возможных комбинаций операндов.

A	B	A **or** B	A **xor** B
False	False	False	False
False	True	True	True
True	False	True	True
True	True	True	False

Результат операции **xor** отличается от результата **or** лишь в одном случае – при истинности обоих операндов. Логично предположить, основываясь на имени операции **or** (ИЛИ), что (A **or** B) = True при истинности ИЛИ одного операнда, ИЛИ другого. Но **or** истинна и при истинности обоих операндов, поэтому **xor** исключает именно этот случай.

Пополним наши сведения о приоритетах операций в выражениях:
- унарные операции +, –, **not**, **, а также функция Sqr() имеют приоритет 1;
- операции *, /, **div**, **mod**, **and** имеют приоритет 2;
- бинарные операции + и –, а также **or** и **xor**, имеют приоритет 3;
- операции отношения =, <>, >, >=, <, <= имеют приоритет 4.

Временами может возникать необходимость запрограммировать заданное логическое выражение, содержащее логические операции, отсутствующие в языке Паскаль. В этом случае нужно предварительно сделать эквивалентное преобразование.

Запись в формуле	Наименование операции	Запись в PascalABC.NET
A∨B, A+B, A|B	Логическое ИЛИ, дизъюнкция	A **or** B
A∧B, A&B, A·B, AB	Логическое И, конъюнкция	A **and** B
¬A, !A, Ā	Логическое НЕ, отрицание, инверсия	**not** A
A⊕B, A+B(mod2)	Исключающее ИЛИ, сложение по mod 2	A **xor** B
A≡B, A↔B	Эквивалентность	**not** (A **xor** B)
A→B, A⊃B	Импликация	**not** A **or** B

Теперь у нас есть все необходимые знания, позволяющие строить и записывать произвольные логические выражения.

Пример 1. Пусть требуется написать условие принадлежности точки с координатами (x, y) II или IV координатной четверти, исключая координатные оси.

Во II четверти абсциссы (x) точек отрицательны, а ординаты (y) положительны. В IV четверти наоборот, абсциссы положительны, а ординаты отрицательны. Абсциссы и ординаты объединяем по И.

Условие для II-й четверти: `(x < 0) and (y > 0).`
Для IV-й четверти: `(x > 0) and (y < 0).`
Объединяем эти условия по ИЛИ, получая
`(x < 0) and (y > 0) or (x > 0) and (y < 0).`
Поскольку приоритет **or** ниже, чем **and**, удалось обойтись без лишних двух пар скобок.

Пример 2. Записать в языке Паскаль выражение $(x \vee y)\,\&\, z \equiv x \to \neg y + z \cdot x$

Здесь просто пользуемся таблицей, приведенной выше.

```
not (((x or y) and z) xor (not x or not y or z and x))
```

Пример 3. Вычислить значение выражения ¬(a & b)→(¬a | c) при a=True, b=False, c=True.

```
begin
  var (a, b, c) := (True, False, True);
  Println(a and b or (not a or c))
end.
```

3.2.3 О «короткой схеме»

Выражение P **or** Q истинно, если хотя бы P или Q истинно. Резонно задать вопрос: если P истинно, зачем вычислять Q? Короткая схема вычисления значения логического выражения дает ответ: не надо его вычислять! А надо ли вычислять P, если истинно Q? Конечно, тоже нет. Итак, у нас есть P и есть Q. Если значение одного из них истинно, второе можно не вычислять. Беда в том, что мы ничего не знаем об истинности P и Q, так что придется хотя бы один из операндов вычислить. Для

определенности решили, что вычисляем всегда первый операнд, P, а второй, Q – только если значение P окажется ложным.

«Короткая схема» распространяется также на вычисление P **and** Q. Если значение P = False, результатом операции будет False и Q не вычисляется.

Знание того, что Q может не вычисляться, полезно в двух случаях. Первый случай связан с возможностью не выполнять операцию, которая может привести к возникновению ошибки в программе.

```
(p = 0) or (12 mod p = 3) – обход ошибки деления на 0 при p=0
```

Второй случай позволяет получить гарантированный вызов функции. Например, при a=5 вычисление выражения F(a) and (a > 3.4) обеспечит вызов F(a), а вычисление выражения (a > 3.4) and F(a) обойдет этот вызов. Это важный момент, потому что функция может попутно сделать какие-то действия, например, вывести значение a.

3.3 Условный оператор

Условный оператор реализует выбор одной возможности из двух на основе анализа некоторого условия. Схематически он реализуется так:

```
ЕСЛИ НекотороеУсловиеВыполняется ТО СделатьДействие1
ИНАЧЕ СделатьДействие2
```

В языке Паскаль вместо русских слова пишутся английские:

```
if P then A
else B;
```

Здесь P – это логическое выражение, значение которого проверяется. Если оно истинно (True), выполняется оператор A, если ложно (False) – выполняется оператор B. На месте оператора можно указать любой оператор языка, в том числе, другой условный оператор или блок. Чтобы было легче читать (и понимать) логику программы, в записи условного оператора применяются отступы: **else** (а иногда и **then**) пишется под **if**.

Рассмотрим пример. Пусть даны натуральные числа a и b. Если их сумма четная, то вычислить произведение a×b, иначе вычислить утроенный остаток от целочисленного деления a на b.

```
begin
  var (a, b) := ReadInteger2('Введите два числа:');
  if (a + b) mod 2 = 0 then Println(a * b)
  else Println(3 * (a mod b))
end.
```

Вернемся еще раз к схематической записи условного оператора

```
if P then A
else B;
```

Если для P=False выполнять ничего не нужно, условный оператор можно сократить до записи вида

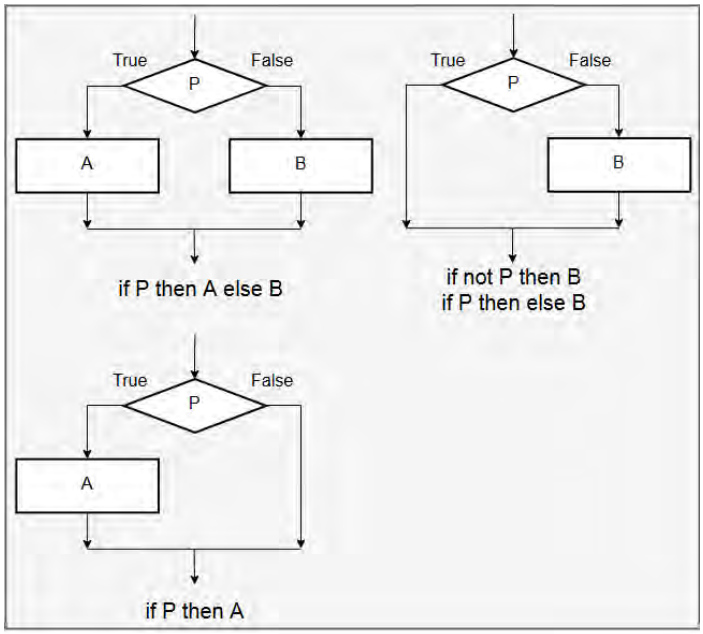
```
if P then A;
```

Если ничего не нужно выполнять для P=True, есть два пути. Первый – инвертировать P и свести форму записи условного оператора к предыдущему формату, второй - не писать оператор A, поскольку его нет:

```
if not P then B;
If A then else B;
```

Какой формат записи выбрать – дело личных предпочтений.

На рис. 3.1 представлены все три возможные блок-схемы и соответствующие им форматы записи условного оператора.



Возможность записывать условные операторы в неполном формате (без **else**) может породить коллизию. Рассмотрим такую запись:

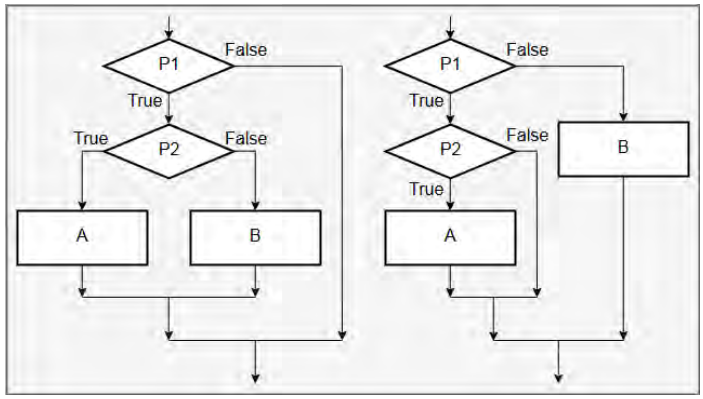
```
if P1 then
if P2 then A
else B;
```

Облегчить понимание реализованного ветвления алгоритма по условиям P1 и P2 помогают отступы. Но отступы сами по себе не определяют, как будет работать программа, а лишь иллюстрируют, как она должна по нашему пониманию работать. Возможны два варианта, показанные на блок-схемах.

```
if P1 then
   if P2 then A
   else B;
```

```
if P1 then
   if P2 then A
else B;
```

В языке Паскаль существует простое правило: каждое **else** связывается с ближайшим предшествующим ему **then**, который не имеет собственного **else**. Согласно этому правилу верна левая из приведенных схем.



Чтобы реализовать правую схему, надо использовать операторные скобки.

```
if P1 then
  begin
    if P2 then A
  end
else B;
```

3.4 Условная операция

Достаточно часто встречаются случаи, когда некоторое значение вычисляется в зависимости от выполнения определенного условия.

Пусть дана формула, определяющая кусочно-заданную функцию

$$y = \begin{cases} 4x+7, \text{ при } x < -5 \vee x > 8 \\ 3x^2 - 11x + 6, \text{ при } -5 \le x \le 8 \end{cases}$$

Запишем программу с использованием условного оператора

```
begin // p03002
  var x := ReadInteger('x=');
  var y: integer;
  if (x >= -5) and (x <= 8) then y := 3 * x * x – 11 * x + 6
  else y := 4 * x + 7;
  Writeln('y=', y)
end.
```

Что в этой программе не очень хорошо? Первое - объявляется переменная y и на это уходит целая строка. Надо заранее решить, какой тип будет иметь эта переменная. И второе, более неприятное – исходная формула распалась на два оператора.

Улучшить ситуацию поможет использование условной операции. Она пришла в PascalABC.NET из языков семейства C, где называлась ***тернарным выражением***. Условная операция по логике вычисления очень похожа на условный оператор и имеет формат

```
P ? A : B
```

Здесь P – некоторое логическое выражение, A и B – выражения (не операторы!), вычисление которых зависит от значения P. Если P=True, вычисляется значение выражения A, если P=False, вычисляется значение выражения B. Вычисленное значение становится значением условной операции.

Условная операция может существенно сократить запись алгоритма. Приведенную выше программу с использованием условной операции можно записать следующим образом

```
begin
  var x := ReadInteger('x=');
  var y := (x >= -5) and (x <= 8) ? 3 * x * x – 11 * x + 6 : 4 * x + 7;
  Writeln('y=',y)
end.
```

Если переменная y нужна только для вычисления и вывода результата, можно сэкономить еще одну строку программы и одну переменную.

```
begin // p03003
  var x := ReadInteger('x=');
  Writeln('y=', (x >= -5) and (x <= 8) ? 3 * x * x – 11 * x + 6 : 4 * x + 7)
end.
```

Сравните эту программу с записью на базовом Паскале:

```
var
  x,y : integer;

begin
  Write('x=');
  Readln(x);
  if (x >= -5) and (x <= 8) then y := 3 * x * x – 11 * x + 6
  else y := 4 * x + 7;
  Writeln('y=', y)
end.
```

С виду не такой уж и большой выигрыш. Но и задача была из простейших. В более сложных задачах подобная «экономия» накапливается и в результате программа на PascalABC.NET получается короче на десятки и даже сотни строк.

Условная операция может внутри себя (в выражениях) содержать другие условные операции и это позволяет реализовывать достаточно сложные вычисления. Кроме того, она сама может быть частью выражения.

Найдем с помощью условных операций максимум из трех чисел (одна из задач, предлагаемая школьникам при изучении программ с ветвлением по условию). Пусть a, b, c – заданные значения. Их максимум можно найти в одну строчку. Заодно вспомните, что в строке можно писать и больше одного оператора. Но делать это следует лишь тогда, когда они некоторым образом взаимосвязаны, а не ради экономии места в коде программы.

```
var max := a > b ? a : b; max := c > max ? c : max;
```

Хороший стиль программирования предполагает расположение этого кода в двух строках:

```
var max := a > b ? a : b;
max := c > max ? c : max;
```

Всем условная операция хороша, но … «На то и щука, чтобы карась не дремал». Вернемся к формальной записи условной операции

```
P ? A : B
```

> Если тип результатов вычисления A и B не совпадает, но оба типа могут быть автоматически приведены один к другому, делается попытка выбрать типом результата некоторый общий, в котором корректно отображаются оба типа. При невозможности корректного отображения будет выбран беззнаковый тип результата. Если один из типов невозможно автоматически преобразовать к другому, типом результата выбирается именно этот тип.

Если автоматическое приведение ни одного из типов к другому невозможно, компилятор выдаст сообщение об ошибке.

Рассмотрим пример.

```
begin
  var (a, b) := (1, 2);
  var c := a > b ? 1234567 : 152.408;
  Println(c)
end.
```

У константы, записанной после разделителя ?, тип **integer**, а у следующей за ней тип **real**. Тип **integer** приводится к **real**, но тип **real** не приводится к **integer**, поэтому для переменной с компилятор назначит тип **real**.

3.5 Оператор выбора

Условный оператор и условная операция позволяют выбрать один вариант из двух. Когда вариантов больше, приходится вкладывать анализ одного условия в другое. При большом количестве вариантов получаются достаточно громоздкие и запутанные конструкции. Решение проблемы предлагает оператор выбора.

```
case Переключатель of
  СписокВыбора1: Оператор1;
  СписокВыбора2: Оператор2;
  …
  СписокВыбораN: ОператорN;
  Else
    ГруппаОператоров
end;
```

Переключатель – это выражение ***порядкового*** или ***строкового*** типа. К порядковому типу относятся целочисленные типы, символьный, логический, а также ***диапазонный*** и ***перечислимый***.

Диапазонный тип получается путем выделения из порядкового типа данных, лежащих в некотором диапазоне, поэтому он совместим с порядковым типом. Например, если переменная порядкового типа **integer** в PascalABC.NET имеет диапазон изменения -2 147 483 648 .. 2 147 483 647, переменная типа **smallint**, в которой может находиться значение из диапазона -32 768 .. 32 767, фактически является диапазонным типом, производным от **integer**.

Переменные диапазонного типа описываются с указанием диапазона, в котором может изменяться значение переменной. Для описания самого типа служит раздел программы **type,** описанный в части 7, но пока будем использовать в переключателе целочисленные типы.

> Константа диапазонного типа указывается путем записи нижней границы диапазона, за которой следуют две точки и верхняя граница диапазона. Например, 3..5 или -25..18.

Списки выбора строятся путем перечисления через запятую констант типа, совместимого с типом переключателя. В случае использования диапазонного типа значения в списках выбора не должны пересекаться.

Работа оператора выбора происходит следующим образом. Значение переключателя отыскивается в списках выбора, начиная от первого по порядку. Если поиск удачен, выполняется соответствующий оператор и на этом работа оператора выбора заканчивается. Если поиск неуспешен, выполняется группа операторов, указанная после **else**. Ветка **else** может отсутствовать и тогда при неуспешном поиске оператор вывода не делает ничего.

Рассмотрим программу, выводящую прописью значение однозначного числа. В ней генерируется случайное целое число n из диапазона 0..9, затем оно отображается на мониторе и при помощи оператора выбора выводится значение числа прописью.

```
begin // p03004
  var n := Random(0,9);
  Print(n);
  case n of
    0: Writeln('ноль');
    1: Writeln('один');
    2: Writeln('два');
    3: Writeln('три');
    4: Writeln('четыре');
    5: Writeln('пять');
    6: Writeln('шесть');
    7: Writeln('семь');
    8: Writeln('восемь');
    9: Writeln('девять');
  end
end.
```

Еще один пример, не требующий особых пояснений:

```
begin
  var n:=Random(-999,999);
  Print(n);
  case n of
    0..9: Writeln('однозначное');
    10..99: Writeln('двухзначное');
    100..999: Writeln('трехзначное');
    else Writeln('отрицательное');
  end
end.
```

Покажем, как могла бы выглядеть эта программа, написанная без применения оператора выбора.

```
begin
  var n := Random(-999, 999);
  Print(n);
  if (n >= 0) and (n <= 9) then Writeln('однозначное')
  else if (n >= 10) and (n <= 99) then Writeln('двухзначное')
  else if (n >= 100) and (n <= 999) then Writeln('трехзначное')
  else Writeln('отрицательное')
end.
```

Не так уж и чудовищно, но точно менее наглядно. Язык программирования предоставляет различные возможности для записи алгоритма, а выбор остается за программистом. Умение сделать его правильно и рационально в конечном счете и отличает профессионала от новичка.

И третий пример. Достаточно часто встречается задача на сравнение двух целочисленных значений. Исходов у такого сравнения три: первое значение больше, значения равны, второе значение больше. Начинающие программировать обычно используют связку из пары условных операторов, но есть и другое решение – с помощью оператора выбора. Оно не совсем очевидно: ведь этот оператор использует константы порядкового типа, а сравнение – логическая операция. Но ведь ничто не мешает результат сравнения «вычислить», например, найти разность первого и второго значения. А переводить знак этой разности в число умеет функция Sign(x).

```
begin
  var (a, b) := ReadInteger2('Введите два
числа:');
  if a > b then Print('Первое число больше')
  else
    if a = b then Println('Числа равны')
    else Println('Второе число больше')
end.
```

```
begin
  var (a, b) := ReadInteger2('Введите
два числа:');
  case Sign(a - b) of
    1: Print('Первое число больше');
    0: Println('Числа равны');
    -1: ('Второе число больше')
  end
end.
```

3.6 Циклы

С помощью алгоритмов, в которых шаги следуют строго друг за другом, возможно, временами разветвляя эти шаги, можно решить крайне ограниченный круг задач. Алгоритмы решения остальных требуют некоторого количества повторений своих отдельных частей. Такие повторяющиеся участки называют циклическими, а операторы языка Паскаль, реализующие соответствующие повторения – ***операторами цикла***. Цикл состоит из ***заголовка*** цикла и ***тела*** цикла. Заголовок определяет условие прекращения (или выполнения) цикла, а тело цикла содержит операторы, которые нужно повторять.

Язык PascalABC.NET предлагает пять разновидностей операторов цикла. Много это или нет – вы решите самостоятельно, ознакомившись со всеми. Здесь описаны четыре разновидности цикла. Пятая (**foreach**) описана в (5.1.2.3).

3.6.1 Цикл с заданным числом повторений (loop)

Это самый простой вид цикла. Его используют, когда число повторений тела цикла заранее известно, а значение номера прохода по циклу не используется. С помощью таких циклов часто оформляют вывод или строят элементы рисунков.

```
loop ЦелочисленноеВыражение do
    ТелоЦикла;
```

ЦелочисленноеВыражение вычисляется и его значение определяет количество повторений тела цикла. ТелоЦикла – любой оператор языка PascalABC.NET, либо блок.

Рассмотрим пример. Выведем в строке 40 звездочек.

```
begin
  loop 40 do
    Write('*');
  Println
end.
```

На практике, конечно, поступают иначе, но пока мы этого еще не умеем:

```
begin
  Println(40 * '*')
end.
```

Поскольку после вывода каждого элемента оператор Print размещает пробел, был использован оператор Write. Оператор Println по завершении вывода сорока звездочек осуществляет переход к новой строке.

Рассмотрим пример **нецелесообразного** нахождения суммы арифметической прогрессии. Нецелесообразного потому, что нет смысла писать программу с циклами там, где результат можно получить по готовой формуле. Но подобные задания почему-то любят давать в школьной (и даже вузовской!) информатике, когда изучают тему с циклами. Итак, требуется найти сумму первых тридцати чисел последовательности, которая начинается числами 5, 8, 11, ...

Попытаемся найти закономерность. 8 - 5 = 3, 11 – 8 = 3. Последовательность образует арифметическую прогрессию с разностью d=3 и первым членом $a_1 = 5$. Требуется найти сумму первых n = 30 членов.

Формула суммы первых n членов арифметической прогрессии известна: $S = \frac{2a_1 + d(n-1)}{2} \times n$

Программа, выполняющая расчет по этой формуле, может быть записана следующим образом:

```
begin // p03005
  var (a1, d, n) := (5, 3, 30);
  Println((2 * a1 + d * (n - 1)) * n div 2)
end.
```

Результатом будет число 1455 и это легко проверить с помощью обычного калькулятора (именно поэтому в реальности никто и не программирует такие задачи – быстрее на бумажке найти ответ). Обратите внимание, что операцию деления мы поставили последней. Причина проста: деление целых выполняется нацело. Числитель может оказаться нечетным числом и ответ окажется неверным. А после умножения на n в числителе всегда будет четное число (это легко доказывается – попробуйте).

Решим эту же задачу при помощи цикла loop.

```
begin // p03006
  var (a1, d, n, s) := (5, 3, 30, 0);
  loop n do
  begin
    s += a1;
    a1 += d
  end;
  Println(s)
end.
```

Здесь пришлось ввести дополнительную переменную s для накопления суммы. Обнуляем s, а затем в теле цикла увеличиваем значение s значение на величину очередного слагаемого. Первое слагаемое равно a_1, а далее в этом же цикле оно наращивается на d. Операция присваивания с инкрементом += оказывается очень к месту.

При достаточно больших значениях n эта программа будет выполняться примерно в n раз дольше предыдущей. Но быстродействие современных компьютеров столь велико, что задержка во времени получения результата будет ощутима, если n превысит десятки миллионов. К сожалению, при этом программа начнет неверно считать из-за превышения суммой значения, отводимого для типа **integer**. А как поправить ситуацию? Поможет явное приведение типа.

```
var (a1, d, n, s) := (int64(5), 3, 1000000000, int64(0));
```

Тут тип **int64** был использован дважды потому, что в теле цикла два присваивания с инкрементом и величина a_1, как и s, превысит значение n. Не забывайте: программирование целочисленных алгоритмов требует особого внимания!

3.6.2 Цикл с параметром (for)

Другое название этого цикла – ***цикл со счетчиком***. Он также организует повторение цикла заданное количество раз, но при этом автоматически увеличивает (или уменьшает) на единицу значение некоторой переменной, указанной в заголовке цикла (***параметра цикла***). Структура цикла может иметь один из следующих видов

```
for var ИмяПараметра : тип := Выражение1 to Выражение2 do // №1
    ТелоЦикла;
```

```
for var ИмяПараметра := Выражение1 to Выражение2 do // №2
    ТелоЦикла;
for var ИмяПараметра : тип := Выражение1 downto Выражение2 do // №3
    ТелоЦикла;
for var ИмяПараметра := Выражение1 downto Выражение2 do // №4
    ТелоЦикла;
```

Тип параметра цикла описывают (№1 и №3), если не устраивает результат автовыведения типа. Этот тип должен быть порядковым. *Выражение1* и *Выражение2* должны быть одного типа, совпадающего с типом параметра цикла или автоматически приводиться к нему. В противном случае компилятор зафиксирует ошибку.

Рассмотрим работу операторов цикла с условными номерами 1 и 2. Перед первым выполнением тела цикла, параметр цикла получает значение *Выражения1*. Вычисляется значение *Выражения2*. Если оно не меньше значения *Выражения1*, выполняется тело цикла. Затем параметр цикла увеличивается на единицу (делается так называемый ***шаг***). Полученное значение параметра цикла вновь сравнивается со значением *Выражения2*. Как только значение параметра цикла станет меньше значения *Выражения2*, цикл завершит свое выполнение.

Выражения, определяющие начальное и конечное значение параметра цикла, вычисляются один раз - перед первым входом в цикл. Значение параметра цикла в теле цикла программисту менять запрещено – это ошибка, фиксируемая компилятором.

Циклы с условными номерами 3 и 4 отличаются тем, что параметр цикла не увеличивается, а уменьшается на единицу. Поэтому тело цикла выполняется лишь пока значение параметра цикла больше или равно значению *Выражения2*.

На рисунке 2.3 приведены блок-схемы цикла **for** с инкрементом параметра цикла (шагом +1), а на рисунке 2.4 – блок-схемы цикла **for** с декрементом параметра цикла (шагом -1). Другая величина шага в этом операторе не предусмотрена.

Название «цикл со счетчиком» вероятнее всего, появилось вследствие того, что параметр цикла ведет себя как счетчик, изменяясь на единицу после каждого выполнения тела цикла. Ведь когда мы накапливаем сумму, то значение переменной увеличиваем на величину очередного слагаемого, а когда что-то подсчитываем – то увеличиваем на единицу.

Цикл **for** пользуется большой популярностью. Особенно часто он используется при работе с такими типами данных, как массивы, рассмотренные в частях 5 и 8.

Количество выполнений тела цикла k можно получить по несложной формуле:
$k = |b - a| + 1$

Составим программу, вычисляющую в цикле **for** сумму первых 30 чисел последовательности 5, 8, 11, ..., которую мы реализовывали с помощью цикла **loop**.

```
begin
  var (a1, d, n, s) := (5, 3, 30, 0);
  for var i := 1 to n do
    s += a1 + d * (i - 1);
  Println(s)
end.
```

Стала ли программа короче? Да, на три строки. Какую цену мы за это заплатили? Добавили еще одну переменную, но это пустяк. Намного важнее, что стала видна формула суммы арифметической прогрессии и теперь понятно, что именно программа вычисляет. А это дает дополнительную уверенность в правильности решения поставленной задачи.

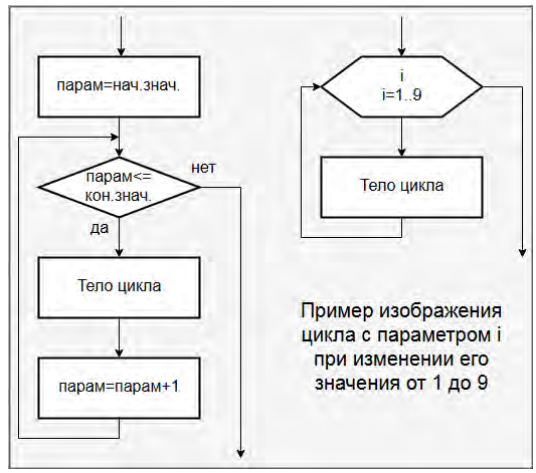


Рисунок 2.3. Блок-схемы цикла **for** с шагом параметра +1

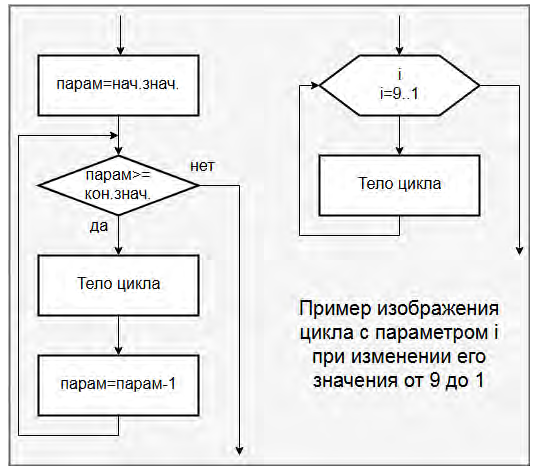


Рисунок 2.4. Блок-схемы цикла **for** с шагом параметра -1

Рассмотрим задачу о вычислении произведения n чисел, введенных с клавиатуры.

```
begin // p03007
  var p := int64(1); // для вычисления произведения
  for var i := 1 to ReadInteger('n=') do
    p *= ReadInteger('Введите число:');
  Println(p)
end.
```

Если у вас еще недостаточно опыта работы с PascalABC.NET, эта программа с первого взгляда может показаться непростой. На самом деле в ней использованы только рассмотренные конструкции и убрано все лишнее.

Первая строка описывает переменную p, которая инициализируется единицей и получает тип **int64** (произведение может оказаться большим и лучше подстраховаться). Инициализация единицей обязательно делается при вычислении произведений так же, как инициализация нулем делается при вычислении сумм. Вторая строка – заголовок цикла с параметром. Начальное значение параметра равно единице, конечное – n. Значение n нужно единственный раз (для вычисления верхней границы параметра цикла) и нет смысла вводить для него отдельную переменную. Используем возможность указывать после **to** выражение, ведь оно вычисляется лишь один раз до входа в цикл. В качестве выражения используем ReadInteger, организуя ввод значения n с приглашением ко вводу. В теле цикла оператор *= домножает значение p на очередное число, вводимое при помощи еще одного ReadInteger. Оператор Println выводит результат.

Что в этой программе не очень хорошо? В заголовке цикла имеется совершенно ненужная переменная *i* – она требуется лишь самому заголовку цикла. Следовательно, в данном случае выбор цикла **for** был неудачным и лучше использовать цикл **loop**, записав оператор **loop** ReadInteger('n=') **do**

3.6.3 Цикл с предусловием (while)

Рассмотренные циклы предполагают известное или вычисляемое число повторений. В то же время, существенное количество алгоритмов основано на неизвестном количестве повторений своих циклических частей (количестве итераций). Такие алгоритмы предполагают, что цикл выполняется (или завершается) при достижении каких-либо условий.

Одна из разновидностей итеративного алгоритма - цикл с предусловием. В заголовке цикла находится некоторое логическое выражение и пока оно истинно, выполняется тело цикла. Цикл завершится, когда условие станет ложным. Приведем структуру соответствующего оператора цикла:

```
while ЛогическоеВыражение do
    ТелоЦикла;
```

В качестве *ТелаЦикла* может быть записан любой оператор языка или блок.

На рисунке 2.5 показаны блок-схемы, изображающие оператор **while**.

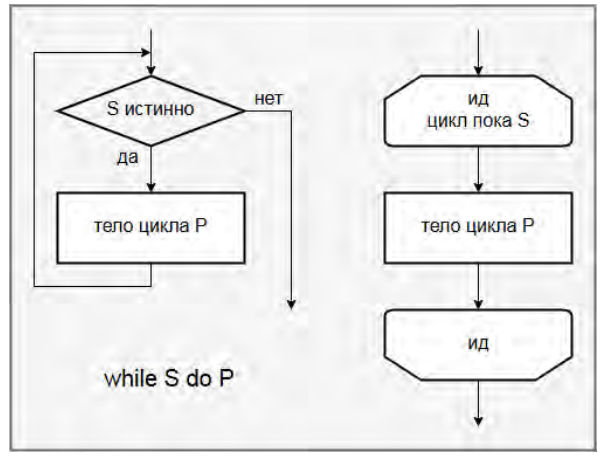


Рис 2.5. Блок-схемы цикла с предусловием (**while**)

Цикл с предусловием можно использовать в математических итерационных алгоритмах для проведения вычислений с заданной точностью, при вводе данных, когда их количество заранее неизвестно, а условие завершения ввода определено некоторым введенным значением, при поиске среди каких-то данных нужного элемента и во многих других случаях.

Рассмотрим пример. Пусть нам требуется найти максимальное значение среди заранее неизвестного количества вводимых натуральных чисел. При появлении нуля или отрицательного числа ввод завершается. Известно лишь, что для диапазона вводимых данных достаточно использовать тип **integer**.

```
begin // p03008
  var (max, n) := (0, 1);
  while n > 0 do
  begin
    n := ReadInteger;
    if n > max then max := n
  end;
  Println('max =', max)
end.
```

Какие решения были приняты при написании программы? Переменная max, отведенная для хранения максимального значения, инициализируется нулем. Это понятно: алгоритм поиска максимума в ряду чисел предполагает инициализацию или первым значением, или значением, меньшим минимального. Первого значения нет, пока не будет осуществлен ввод, но известно, что числа натуральные, поэтому ноль – неплохой выбор. Переменная n назначена для хранения очередного введенного числа. По ее значению проверяется условие завершения цикла, поэтому ее следует инициализировать любым положительным числом, например, единицей. Заголовок цикла **while** с логическим выражением n>0. Берется из условия задачи. Цикл выполняется, пока введенное значение положительно, т.е. является

натуральным числом. В теле цикла осуществляется ввод очередного значения n. Если оно превышает текущее значение max, то оно запоминается в качестве нового максимума. Когда цикл завершится, выводится значение max.

Иногда эту задачу решают с использованием двух операторов ввода. Выбор за вами.

```
begin
  var n := ReadInteger;
  var max := n;
  while n > 0 do
  begin
    if n > max then max := n;
    n := ReadInteger
  end;
  Println('max =', max)
end.
```

3.6.4 Цикл с постусловием (repeat)

Этот цикл похож на цикл с предусловием, но в отличие от него, здесь сначала выполняется тело цикла, а потом проверяется, не следует ли этот цикл завершить.

```
repeat
  ТелоЦикла
until ЛогическоеВыражение;
```

Это единственный из операторов цикла, в котором *ТелоЦикла* может содержать несколько операторов и при этом не быть заключено в операторные скобки.

Цикл с постусловием при истинности *ЛогическогоВыражения* завершается.

Рассмотренную выше программу для нахождения максимального среди введенных чисел можно записать и с помощью цикла с постусловием.

```
begin
  var max := 0;
  var n: integer;
  repeat
    n := ReadInteger;
    if n > max then max := n;
  until n <= 0;
  Println('max =', max)
end.
```

Было девять строк – девять и осталось, так что и тут выбирайте, что больше нравится.

На рисунке 2.6 показаны блок-схемы, изображающие оператор **repeat**

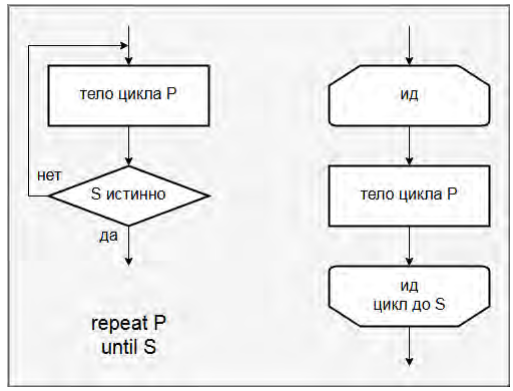


Рисунок 2.6. Блок-схемы цикла с постусловием (**repeat**)

Истинность логического выражения в заголовке оператора цикла **while** определяет условие *выполнения* цикла, а в **repeat** – условие его *завершения*. Наличие предусловия в цикле **while** может привести к тому, что тело цикла не выполнится ни разу. Постусловие в **repeat** – залог того, что тело цикла обязательно выполнится хотя бы один раз. Для бесконечного цикла **while** указываем **while** `True` **do**, для бесконечного **repeat** указываем **until** `False`.

3.6.5 Изменение нормального хода выполнения цикла

В процессе выполнения тела цикла могут возникать обстоятельства, требующие изменить ход его обычного выполнения, например, досрочно завершить выполнение цикла или прекратить выполнять текущий проход по телу цикла и начать следующий.

3.6.5.1 Оператор break

Оператор вызывает немедленный выход из цикла. Используется, если обнаружена нецелесообразность дальнейшего выполнения тела цикла. Помним, что после выхода из цикла все переменные, определенные в теле цикла с помощью **var**, а также параметр цикла **for**, становятся недоступны.

Рассмотрим программу, находящую сумму последних цифр введенного натурального числа, не превышающую 21.

```
begin
  var (s, n, final) := (0, ReadInteger, False);
  while (not final) and (n > 0) do
  begin
    var d := n mod 10; // очередная цифра числа
    if s + d <= 21 then s += d
    else final := True;
    n := n div 10
  end;
  Println(s)
end.
```

Это пример реализации программы с «флажком». Описывается переменная-флажок (*final*), которая перед входом в цикл инициализируется значением False. В заголовок цикла включается проверка флажка на True – признак, что цикл нужно завершить. Программы с флажками были весьма популярны в то время, когда после выхода работы Э. Дейкстры о структурном программировании повсеместно начались гонения на сторонников оператора перехода **goto**. К сожалению, наличие флажков затрудняет понимание реализованного алгоритма. Оператор break помогает несколько улучшить ситуацию.

```
begin
  var (s, n) := (0, ReadInteger);
  while n > 0 do
  begin
    var d := n mod 10; // очередная цифра числа
    if s + d <= 21 then s += d
    else break;
    n := n div 10
  end;
  Println(s)
end.
```

3.6.5.2 Оператор continue

Оператор немедленно передает управление заголовку цикла. Оставшаяся часть тела цикла не выполняется. Если условие завершения цикла, определенное заголовком, не выполняется, начинается очередное выполнение тела цикла.

На практике оператор **continue** нужен редко, поскольку легко и без ущерба для наглядности заменяется условным оператором, пропускающим при необходимости оставшиеся операторы в теле цикла путем «упрятывания» их под **else**.

Пример: Вывести натуральные числа, не превышающие 20, которые не кратны 3.

```
begin // p03009 самый простой и понятный вариант
  for var i := 1 to 20 do
    if i mod 3 <> 0 then Print(i);
  Println
end.
```

```
begin // надуманный вариант с continue
  for var i := 1 to 20 do
  begin
    if i mod 3 = 0 then continue;
    Print(i)
  end;
  Println
end.
```

```
begin // он же, без continue
  for var i := 1 to 20 do
    if i mod 3 = 0 then
    else Print(i);
  Println
end.
```

3.6.5.3 Оператор exit

Единственное назначение – немедленно завершить работу той программной единицы, в которой он встретился. Бывает очень полезен.

3.6.6 О вложенных циклах

Поскольку в теле цикла может находиться любой оператор, ничто не мешает разместить там оператор цикла. Так появляются вложенные циклы. Они вкладываются друг в друга, как матрешки, ровно столько раз, сколько этого требует алгоритм. Количество вложений циклов друг в друга в программировании называют ***глубиной вложения***.

В качестве примера рассмотрим программу, печатающую таблицу умножения. В ней имеется вложенный цикл с глубиной вложения, равной двум. В программе использована процедура вывода Write(i * j : 3), дополняющая выводимое значение слева пробелами до трех позиций.

```
begin // p03010
  Write('   ');
  for var j := 1 to 9 do
    Write('  ', j);
  Println;
  for var i := 1 to 9 do
  begin
    Write('  ', i);
    for var j := 1 to 9 do
      Write(i * j: 3);
    Println;
  end
end.
```

Операторы **break** и **continue** действуют только в пределах того цикла, в теле которого они записаны. Если требуется прервать выполнение не только внутреннего цикла, но и внешнего по отношению к нему, приходится идти на различные программистские уловки. Решение «в лоб» - использовать **goto** – ***оператор перехода***. При этом дополнительно потребуется описывать метки в разделе меток. Современный подход к программированию дает достаточное количество средств, позволяющих обойтись без этого архаизма, поэтому оператор **goto** в книге не рассматривается. Если он вам нужен – открывайте любую книгу по базовому Паскалю. В следующей части книги будет рассмотрен один из современных способов досрочного прерывания выполнения группы вложенных циклов из самого внутреннего – помещение «матрешки» из вложенных циклов в подпрограмму.

3.6.7 Задача табуляции функции

Весьма популярна задача получения таблицы значений некоторой функции для значений из заданного интервала. Рассмотрим ее реализацию с помощью операторов цикла.

Пусть требуется составить таблицу значений функции P(x,y) для x, меняющегося от -2π до 3π с шагом π/6 и y, меняющегося от -1.2 до 5.9 с шагом 0.4. Часто для указания подобных диапазонов используют запись вида -2π(π/6)3π и -1.2(0.4)5.9. Реализация состоит в использовании вложенных циклов. Будем для каждого значения y задавать перебор всех значений x, тогда внешний (первый) цикл будет менять значение y, а внутренний (второй, вложенный) будет менять значение x. Можно и наоборот, сделать внешний цикл по x, а внутренний – по y.

$$P = \frac{1.4e^{-1.6y} \cdot \cos^2\left(\frac{\pi x}{2}\right)}{1 + 0.17\sqrt[3]{x - 1.185}}$$

3.6.7.1 Использование цикла for

Цикл for обеспечивает изменение своего, только целочисленного параметра, да и то на 1 или -1. Для использования такого цикла потребуется 1) найти количество повторений цикла, обеспечивающее получение всех значений аргумента x (y) и 2) найти формулу, связывающую очередное значение параметра цикла с нужным значением аргумента x (y). Не очень-то приятное занятие.

Найдем количество повторений цикла n_x, обеспечивающее получение интервала изменения x = -2π(π/6)3π и выведем формулу, дающую необходимые значения аргумента x при i = 0, 1, … n_x-1:

$$n_x = \left\lfloor \frac{3\pi - (-2\pi)}{\frac{\pi}{6}} \right\rfloor + 1, \quad x = \frac{\pi i}{6} - 2\pi$$

Сделаем то же для аргумента y:

$$n_y = \left\lfloor \frac{5.9 - (-1.2)}{0.4} \right\rfloor + 1, \quad y = 0.4j - 1.2$$

Угловые скобки здесь обозначают операцию взятия целой части выражения, получаемую путем отбрасывания дробной части, т.е. стандартную функцию Trunc(a) для приведения типа **real** к **integer**. Упростим формулы и напишем программу.

$$n_x = 31, \quad x = \pi\left(\frac{i}{6} - 2\right), \quad n_y = \left\lfloor \frac{5.9 + 1.2}{0.4} \right\rfloor + 1, \quad n_y = 0.4j - 1.2$$

```
begin // p03011
  for var j := 0 to Trunc((5.9 + 1.2) / 0.4) do
  begin
    var y := 0.4 * j - 1.2;
    for var i := 0 to 30 do
    begin
      var x := Pi * (i / 6 - 2);
      var p := 1.4 * Exp(-1.6 * y) * Sqr(Cos(Pi * x / 6));
      var p1 := x - 1.185;
      p1 := 1 + 0.17 * Sign(p1) * Abs(p1) ** (1 / 3);
      p := p / p1;
      Println(y, x, p)
    end
  end
end.
```

Остановимся на программировании выражения для функции P(x,y). Сначала вычислен числитель p, в следующих двух строках – знаменатель p1 и лишь затем требуемое значение как p/p1. «Виновник» - кубический корень. В математике он определен для отрицательного аргумента, а операция возведения отрицательного значения в нецелую степень в PascalANC.NET дает NaN. Приходится вычислять кубический корень из абсолютной величины аргумента, а результат умножать на «знак» аргумента, возвращаемый функцией Sign(x). Зато теперь вы вспомнили (а кто-то узнал), как нужно поступать в подобных ситуациях.

Для тех, у кого получение формул для вычисления x и y на основе i и j вызвало затруднения. Если нужно сформировать значения аргумента для заданных a(h)b, где a – начальное значение, h - шаг изменения, b – конечное значение, то для пара-

метра цикла i = 0, 1, ... n-1 получаем x = a+i*h, а значение n-1 находим как Trunc((b-a)/h). Немного хлопотно, зато надежно, как в танке.

3.6.7.2 Использование цикла while

Никаких предварительных расчетов, никаких формул для числа повторений цикла и получения значений x и y ! С первого взгляда – превосходный вариант для задачи табуляции. Посмотрим, так ли это на самом деле.

```
begin // p03012
  var (ax, hx, bx) := (-2 * Pi, Pi / 6, 3 * Pi);
  var (ay, hy, by) := (-1.2, 0.4, 5.9);
  var y := ay;
  while y <= by do
  begin
    var x := ax;
    while x <= bx do
    begin
      var p := 1.4 * Exp(-1.6 * y) * Sqr(Cos(Pi * x / 6));
      var p1 := x - 1.185;
      p1 := 1 + 0.17 * Sign(p1) * Abs(p1) ** (1 / 3);
      p := p / p1;
      Println(y, x, p);
      x += hx
    end;
    y += hy
  end
end.
```

Длина программы увеличилась, но не сильно – на четыре строки – и это мелочь. Но давайте сравним, например, несколько первых строк результатов.

-1.2 -6.28318530717959 13.9895517325122	-1.2 -6.28318530717959 13.9895517325122
-1.2 -5.75958653158129 13.9105366758217	-1.2 -5.75958653158129 13.9105366758217
-1.2 -5.23598775598299 11.8430510155381	-1.2 -5.23598775598299 11.8430510155381
-1.2 -4.71238898038469 8.41134136228866	-1.2 -4.71238898038469 8.41134136228865
-1.2 -4.18879020478639 4.62241308051433	-1.2 -4.18879020478639 4.62241308051432
-1.2 -3.66519142918809 1.56173308815685	-1.2 -3.66519142918809 1.56173308815684
-1.2 -3.14159265358979 0.0724644209982689	-1.2 -3.14159265358979 0.0724644209982676
-1.2 -2.61799387799149 0.5131369555664	-1.2 -2.61799387799149 0.513136955566401
-1.2 -2.0943951023932 2.66367139092203	-1.2 -2.094395102393 19 2.66367139092203
-1.2 -1.5707963267949 5.80444695277673	-1.2 -1.5707963267949 5.80444695277674
-1.2 -1.0471975511966 8.94133315565904	-1.2 -1.0471975511966 8.94133315565904
-1.2 -0.523598775598299 11.1066480938966	-1.2 -0.523598775598299 11.1066480938966
-1.2 0 11.6440632062945	-1.2 2.22044604925031E-16 11.6440632062945

Отличия выделены жирным курсивом. С виду они ничтожны, но все же они есть. И это дает право предполагать, что могут существовать случаи, когда такие отличия окажутся существенными. Разберемся, из-за чего эти различия возникают.

Причина проста: в одном случае значения аргументов получается путем умножения, в другом – путем сложения. В сложении участвует шаг – значение π/6, пред-

ставленное в компьютере с некоторой погрешностью ε. После n шагов для цикла **for** получаем $b_1 = a+(n-1)(h+\varepsilon) = a+h(n-1)+\varepsilon(n-1)$. Для цикла **while** после n шагов получаем $b_2 = a+(h+\varepsilon)_1+ (h+\varepsilon)_2+ \ldots +(h+\varepsilon)_{n-1}$. Математически можно утверждать, что $b_1 \equiv b_2$, но в машинной арифметике это не так. При накоплении суммы очередное значение b_i округляется, что вносит дополнительные ошибки. По этой причине нужно предпочитать алгоритмы, вычисляющие значения путем умножения, а не путем последовательных сложений.

> При накоплении суммы вещественных чисел одновременно накапливаются ошибки машинного округления!

3.6.7.3 Использование цикла repeat

Для задачи табуляции мало чем отличается от программирования с циклом **while**. Единственная неприятность может возникнуть, если для какой-то комбинации параметров вычисление выполняться не должно: цикл repeat всегда выполняется хотя бы один раз. Но это не наш конкретный случай, поэтому пишем программу.

```
begin // p03013
  var (ax, hx, bx) := (-2 * Pi, Pi / 6, 3 * Pi);
  var (ay, hy, by) := (-1.2, 0.4, 5.9);
  var y := ay;
  repeat
    var x := ax;
    repeat
      var p := 1.4 * Exp(-1.6 * y) * Sqr(Cos(Pi * x / 6));
      var p1 := x - 1.185;
      p1 := 1 + 0.17 * Sign(p1) * Abs(p1) ** (1 / 3);
      p := p / p1;
      Println(y, x, p);
      x += hx
    until x > bx;
    y += hy
  until y > by
end.
```

Код на две строки короче, чем с циклом **while** и на две строки длиннее, чем с циклом **for**. Результат в точности совпадает с результатом для цикла **while**.

Ошибки округления, изначально представляющиеся ничтожно малыми, могут преподнести неприятный сюрприз. По замыслу неумелого программиста программа должна была вычислить квадраты для значений от 8/3 до 11/3 с шагом 1/3.

```
begin
  var (ax, hx, bx) := (8 / 3, 1 / 3, 11 / 3);
  var x := ax;
  repeat
    Println(x, Sqr(x));
    x += hx
  until x > bx
end.
```

Эта программа выводит следующие строки:

```
2.66666666666667 7.11111111111111
3 9
3.33333333333333 11.1111111111111
```

Первая строка – это для x = 8/3, вторая для 9/3 = 3, третья для 10/3. А где 11/3? Виновата ошибка округления. Значение x сумело превысить 11/3 и выход из цикла произошел досрочно. Рецепт простой: в проверке условия завершения цикла добавить к bx еще что-то. Обычно добавляют долю шага. Например, половину.

```
begin
  var (ax, hx, bx) := (8 / 3, 1 / 3, 11 / 3);
  var x := ax;
  repeat
    Println(x, x*x);
    x += hx;
  until x > (bx+hx/2)
end.
```

Теперь все работает нормально.

> Программу табуляции функции предпочтительно писать на основе цикла **for**. Если программист испытывает затруднения с предварительными расчетами, можно использовать циклы **repeat** и **while**.

В части 5 вы познакомитесь с еще одним циклом – **foreach**, позволяющим записать приведенную программу компактнее, нагляднее и без шаманства обеспечить выполнение нужное количество проходов по циклу. А пока посмотрите код такой программы:

```
begin
  foreach var x in PartitionPoints(8 / 3, 11 / 3, 3) do
    Println(x, x * x)
end.
```

И это все. Вы ведь стремитесь научиться современному программированию на PascalABC.NET, а оно именно таково: программа пишется быстро, коротко, понятно и позволяет компилятору порождать эффективный код.

Более того, можно сделать программу еще короче:

```
begin
  PartitionPoints(8 / 3, 11 / 3, 3).ForEach(x -> Println(x, x * x))
end.
```

Всегда существует множество решений одной и той же задачи. Умение мысленно увидеть эти решения, а затем выбрать из них оптимальное, отличает опытного программиста от новичка.

Может быть, подобные ошибки появляются при табуляции функции на интервале с экзотическими границами и экзотическим шагом? В качестве домашнего задания (и для получения ответа на заданный вопрос) попробуйте табулировать функцию

y = x + 1 на внешне вполне «безобидном» интервале (0 (0.2) 4) с помощью оператора цикла **while** или **repeat**, не прибавляя доли шага в проверке условия завершения цикла.

3.7 Для самостоятельного решения

Т3.1. Юлианская дата. Время от времени приходится решать непростую задачу: найти продолжительность некоторого достаточно протяженного интервала времени, зная даты его начала и конца. В науке с этой целью используют юлианские даты. В XVI веке французский историк Жозеф Скалигер предложил отсчитывать время в юлианских периодах, основанных на цикле продолжительностью 7980 лет. В дальнейшем, английский астроном Уильям Гершель развил идею Ж. Скалигера, предложив все даты выражать через число дней, прошедших от некоторого Юлианского дня, за который принимается начало цикла Скалигера. Сами даты при этом получили называния юлианских. С помощью юлианского календаря задача определения продолжительности исторических событий решается крайне просто. Например, зная что Великая Отечественная война началась 22 июня 1941 года, а закончилась 9 мая 1945 года и переведя эти даны в юлианские, можно вычислить продолжительность войны в днях: 2431585 – 2430168 +1 = 1418.

Для перевода даты григорианского календаря (d-m-y – день, месяц и год) в юлианскую предложен следующий алгоритм:

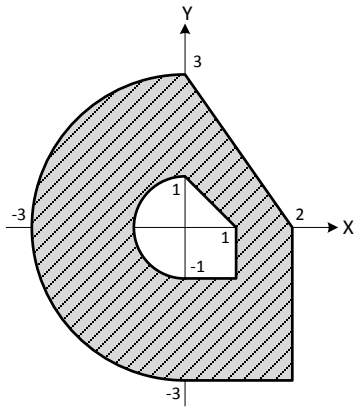
- Если m > 2, уменьшить m на 3, иначе уменьшить y на 1 и увеличить m на 9;
- Вычислить c, как результат целочисленного деления y на 100;
- Вычислить a, как остаток от деления y на 100;
- Получить номер юлианского дня по формуле (все деления – целочисленные)

$$j = \frac{146097 \times c}{4} + \frac{1461 \times a}{4} + \frac{153 \times m + 2}{5} + d + 1721119$$

Т3.2. Задача о сдаче. В классической формулировке рассматривается возможность выдачи сдачи S с некоторой полученной суммы минимальным количеством монет при условии, что имеются монеты лишь определенного достоинства, например, 1, 2, 5 и 10 рублей. Для упрощения задачи считается, что запас монет каждого достоинства не ограничен.

Одно из наиболее простых решений получается при помощи так называемого «жадного» алгоритма. Вначале пытаемся выдать сумму максимально возможным количеством монет наибольшего достоинства, т.е. десятирублевыми. Полученный остаток выдаем монетами оставшихся номиналов, рассматривая сначала пятирублевую монету, затем, если понадобится, двухрублевую и однорублевую. Составьте программу, которая запрашивает сумму сдачи и выводит нужное количество монет каждого достоинства, а также общее количество монет.

Т3.3. Попадание точки в заштрихованную область, включая ее границы. Дана точка с вещественными координатами (x,y). Определить, попадает ли она в заштрихованную область, показанную на рисунке.



В помощь тем, кто еще не изучал, либо уже забыл уравнения элементарных линий:

- прямая, проходящая через точки (3;3) и (2;0) имеет уравнение y = 3 - 1.5x;
- прямая, проходящая через точки (0;1) и (1;0) имеет уравнение y = 1 - x;
- прямая, проходящая через точки (0;-3) и (2;-3) имеет уравнение y = -3;
- прямая, проходящая через точки (0;-1) и (1;-1) имеет уравнение y = -1;
- прямая, проходящая через точки (2;0) и (2;-3) имеет уравнение x = 2;
- прямая, проходящая через точки (1;0) и (1;-1) имеет уравнение x = 1;
- внешняя окружность радиуса 3 имеет уравнение $x^2 + y^2 = 9$;
- внутренняя окружность радиуса 1 имеет уравнение $x^2 + y^2 = 1$.

Т3.4. Составить программу, которая запрашивает натуральное число n и выводит все его простые множители (т.е. произведение выведенных простых чисел должно равняться n). Предлагается реализовать следующий алгоритм:

1. Положить текущее значение делителя d равным 2;
2. Если n = 1, перейти к 5;
3. Если n делится на d без остатка, за новое значение n принять частное от деления n на d и вывести значение d (d – делитель n). Перейти к 2;
4. Если d^2 превышает n, принять значение n в качестве d, в противном случае увеличить значение d на единицу. Перейти к 3;
5. Завершить работу алгоритма.

Т3.5. Найти сумму цифр в целом числе типа **integer**. Учесть возможность ввода отрицательного числа.

Т3.6. Сгенерировать 15 случайных целых чисел в диапазоне от -999999 до 999999. Вывести для каждого числа само число и сумму его цифр. Найти и вывести число с наибольшей суммой цифр

Т3.7. Найти цифровой корень числа для типа данных **int64**. Цифровым корнем называется однозначное число, которое получается путем нахождения суммы цифра числа, затем суммы цифр этой суммы и так далее, пока значение не станет однозначным. Например, для числа 1582094673 получаем последовательно 45 и 9.

Часть 4

Подпрограммы

Каждая программа является частью другой программы и редко соответствует ей.

Алан Перлис, американский ученый

Функции используются для наведения порядка в хаосе алгоритмов.

Бьёрн Страуструп, создатель языка C++

4.1 Общие сведения

Парадигма процедурного программирования предполагает, что вся программа разбивается на самостоятельные программные единицы (подпрограммы), которые определенным образом связываются между собой посредством обращений друг к другу (***вызовов***). При этом различают вызывающие подпрограммы, из которых осуществляется вызов и вызываемые, которые подлежат вызову. Вызываемая подпрограмма может, в свою очередь, оказаться вызывающей и тогда говорят об уровне вложенности (***глубине***) вызовов относительно конкретной программной единицы. Отношение «вызывающий – вызываемый» достаточно условно. В частности, вызывающая подпрограмма непосредственно или через другие подпрограммы может вызывать сама себя - это называется ***рекурсивным вызовом*** или просто ***рекурсией*** (см. 4.4).

Среди подпрограмм принято различать ***процедуры*** и ***функции***. Процедура вызывается как самостоятельный компонент программы и ее вызов часто неотличим от оператора языка. Например, как понять что такое Print в записи Print(2 + 2) – оператор или процедура? Функция – это разновидность процедуры, которая в результате вызова возвращает вычисленное ей значение, поэтому функция является полноправным членом выражения. В то же время, если возвращаемое функцией значение не используется, к ней можно обратиться, как к процедуре.

В PascalABC.NET программист может обращаться к уже имеющимся подпрограммам, а также создавать собственные (пользовательские) подпрограммы. Для удобства работы подпрограммы могут компоноваться в самостоятельные программные единицы – ***модули*** (часть 11). Такие модули требуют подключения в разделе **uses**. Имеется возможность и непосредственного обращения к подпрограммам из состава платформы Microsoft .NET Framework.

Рассмотрим пример кода, подобный тем, какие мы писали не раз. Пришло время посмотреть на него с точки зрения процедурной парадигмы.

```
begin
  var (a, b) := ReadInteger2('Введите два целых числа');
  Println('Минимальное из них равно', Min(a, b))
end.
```

ReadInteger2 – это функция, а текст приглашения ко вводу является ее ***параметром***. Функция возвращает кортеж из двух целочисленных значений. Println – про-

цедура, осуществляющая вывод списка переданных ей параметров, один из которых является результатом вызова функции Min, в свою очередь, имеющую два параметра a и b.
Для иллюстрации приведен пример обращения к библиотеке платформы Microsoft .NET Framework

```
begin
  Println(DateTime.Now)
end.
```

Подобное обращение всегда содержит в записи одну или несколько точек. В данном случае говорят, что производится обращение к свойству Now класса DateTime, возвращающему текущие дату и время. Если терминология непонятна, пока пропустите ее: в свое время все будет разъяснено.

4.1.1 Параметры в подпрограммах

Параметры – это некоторые переменные, позволяющие подпрограммам обмениваться данными между собой. Параметры могут передаваться ***по значению*** и ***по ссылке***. Передача по значению означает пересылку копии данных. Передача по ссылке – пересылку лишь информации о том, где находятся данные. Чем больше объем данных, тем выше затраты на их копирование, тем выгоднее передавать такие данные по ссылке.

… У Маши есть младший брат Вова. Он написал домашнее сочинение, и просит Машу проверить, нет ли там ошибок. Как поступит Вова: сделает копию своего сочинения и отдаст Маше на проверку или предложит Маше взять его тетрадку со стола? Первый вариант действий – передача по значению, второй – передача по ссылке.

Потом Вова сел делать задание по физике. Он решил задачу, но при вычислении результата потребовалось вычислить квадратный корень из 13,5. А Маша как раз что-то считала на калькуляторе. Как поступит Вова: спросит, сколько будет, если извлечь квадратный корень из 13,5 или попросит Машу подойти, посмотреть на выражение и вычислить корень? И здесь первый вариант действий – передача по значению, второй – передача по ссылке.

Приведенные примеры показывают, что есть случаи, когда выгоднее передавать параметры по значению, и есть случаи, когда выгоднее передавать их по ссылке. Как именно должен передаваться параметр определяет описание в заголовке вызываемой подпрограммы. Это позволяет компилятору находить ошибки в случае неправильного использования параметров.

4.1.2 Размерные и ссылочные типы данных

Программа, написанная на языке PascalABC.NET, выполняется в среде Microsoft .NET Framework. Для повышения эффективности управления оперативная память в этой среде разделена на два типа – ***стек*** (stack) и ***кучу*** (heap). Мы не будем сейчас

касаться устройства этих структур, но отметим, что компилятор PascalABC.NET строит программный код с учетом такой организации памяти.

Во введении было сказано, что среди типов данных можно выделить два больших типа: ***размерные*** и ***ссылочные***. Когда мы используем в программе имя переменной, в памяти отыскивается некоторая информация, связанная с этим именем. Если эта информация является значением указанной переменной, тип данных считается размерным, а если она является адресом в памяти, начиная с которого располагается область хранения данных, тип данных считается ссылочным. Сама же информация о переменной в последнем случае называется ***корневой ссылкой***. У ссылочных типов имя переменной указывает место расположения корневой ссылки, которая указывает, в свою очередь, на область хранения данных или, если структура данных сложная, на другие ссылки. Компилятор располагает данные ссылочного типа в куче, а корневые ссылки всегда располагаются на стеке, что увеличивает скорость доступа к данным.

Данные размерного (***value***) типа являются ***статическими*** – их размер определяется на стадии компиляции программы. Компилятор старается размещать такие данные в стеке. Работа со стеком происходит быстрее, чем с кучей, поскольку данные в нем упорядочены. Ссылочные (***reference***) данные – ***динамические***, их размер может быть заранее неизвестен (например, может быть определен после ввода и/или обработки некоторых данных) и под такие данные память выделяется из кучи. В куче данные не обязательно занимают смежные участки памяти и это усложняет (и замедляет) доступ к ним.

Для англоязычных программистов все логично: параметры с типом данных value удобно передавать «by value» (по значению), а с типом reference - соответственно «by reference» (по ссылке). Устоявшаяся терминология перевода на русский язык заставляет говорить, что размерный тип удобно передавать по значению, а ссылочный – по ссылке. Но это удобства с точки зрения накладных расходов на передачу данных. А еще есть необходимость, которая временами вынуждает забыть об экономии. И она такова:

> Если после вызова подпрограммы мы желаем получить измененное ею значение некоторого параметра, такой параметр необходимо передать по ссылке. В противном случае параметр можно (иногда и необходимо) передать по значению.

4.1.3 Ссылка и значение

Когда мы записываем оператор присваивания `x := x + 1`, возникает смысловая путаница между местом в памяти компьютера, где находится некоторое значение, и самим этим значением. В правой части x обозначает само значение, а в левой – имя, соотнесенное с местом расположения значения. Чтобы избежать путаницы, говорят, что слева находится ссылка, а справа – значение. Ссылка не хранит значение данных, а лишь указывает место, где находится значение и обеспечивает доступ к этому месту через свое внутреннее имя, которое создает компилятор.

Ссылка – создаваемый компилятором посредник для связи имени, которое программист дал переменной, со значением этой переменной.

Вычисление значения выражения заключается в замене всех имен переменных их значениями и последующим выполнением над этими значениями указанных операций. Собственно, то же самое мы делаем без компьютера, подставляя в формулу конкретные значения. Результатом вычисления будет значение, которое размещается в месте, указанном ссылкой.

4.2 Процедуры

Процедура представляет собой самостоятельный блок программного кода, к которому можно обращаться для выполнения из различных мест программы. Процедура состоит из ***заголовка процедуры*** и ***тела процедуры***. Описание пользовательской процедуры имеет следующий вид:

```
procedure ИмяПроцедуры(Параметр1; … ПараметрN); // заголовок процедуры
begin
    ТелоПроцедуры
end;
```

Заголовок процедуры представляет собой имя процедуры, за которым может следовать список описаний параметров процедуры, разделенных точками с запятой и заключенный в круглые скобки. Если параметры отсутствуют, скобки тоже не ставятся. ИмяПроцедуры – обычный идентификатор и его написание подчиняется общепринятым правилам. Для каждого параметра указываются его имя и тип. Чтобы передать параметр размерного типа по ссылке, перед его именем указывают ключевое слово **var** или **const**. Для ссылочного типа в большинстве случаев указывать var или const не требуется. Описание переменных одного типа можно объединять, разделяя запятыми.

В случае, когда тело процедуры состоит из единственного оператора, можно использовать ***упрощенный синтаксис***:

```
procedure ИмяПроцедуры(Параметр1; … ПараметрN) : = Оператор;
```

При наличии описания процедуры, к ней можно обратиться (***вызвать*** процедуру). Вызов имеет вид

```
ИмяПроцедуры(Параметр1, … ПараметрN); // процедура с параметрами
ИмяПроцедуры; // процедура без параметров
```

В Паскале нельзя обратиться к процедуре (и к функции) до того, как она описана, поэтому в коде программы описания вызываемых процедур всегда предшествуют описанию процедур, вызывающих их. По этой же причине основная программа всегда следует последней и точка в ее конце – это общий конец кода программы.

Параметры, используемые в описании процедуры, называются ***формальными***. При вызове процедуры формальным параметрам в порядке их записи сопоставляются ***фактические*** параметры. Некоторые авторы предпочитают иную терминологию, называя формальные параметры просто ***параметрами***, а фактические – ***аргументами***.

Если формальный параметр принимается по значению, фактическим параметром должно быть выражение, тип которого совпадает с типом аргумента или автоматически к нему приводится. Если формальный параметр требует ссылки (перед ним указано **var** или **const**), то фактический параметр должен быть именем переменной, имеющей точно такой же тип.

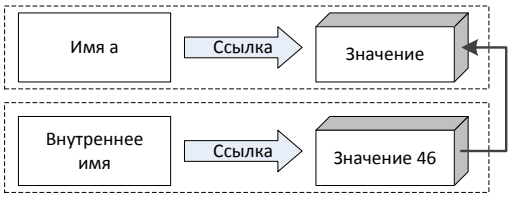
4.2.1 Передача параметра по значению

По значению передаются параметры, величину которых процедура не должна менять. Строго говоря, менять она их может, но только внутри себя, чтобы вызывающая программа об этом не знала. Параметр принимается по значению, если он размерного типа и в описании процедуры перед его именем не указано ключевое слово **var** или **const**.

Реализация передачи параметра по значению очень проста. Пусть имеется некая процедура P, имеющая формальный параметр a размерного типа **integer**, который принимается по значению. Заголовок такой процедуры можно записать как

```
procedure P(a: integer);
```

Обратимся к процедуре P, передав ей в качестве фактического параметра значение 46. Такой вызов будет иметь вид P(46).



Значения данных размерного типа хранятся на стеке. Произойдет прямое копирование значения 46 в другую область на стеке, на которую ссылается формальный параметр a. Вызывающая программа выполнит копирование, а потом забудет об этом. Что вызванная процедура P будет делать со значением, которое она «знает» под именем a, вызывающую программу не интересует. Более

того, как будет показано в дальнейшем, подобное любопытство большинство парадигм программирования пресекает.

4.2.2 Передача параметра по ссылке

По ссылке в процедуру передаются параметры размерного типа, для которых нужно разрешить изменение значений, а также параметры ссылочного типа. Сложные структуры данных могут занимать много места в памяти. Если эти структуры имеют размерный тип, их следует стараться передавать по ссылке, даже если изменение значений не планируется: это позволит избежать расходов на копирование, снижающее эффективность программного кода.

Для приема фактического параметра размерного типа по ссылке, перед именем соответствующего формального параметра в заголовке описания процедуры указывается ключевое слово **var** или **const**.

Ключевое слово **var** означает, что копироваться будет не сами данные, а ссылка на них. Точнее, для размерного типа данных копируется ссылка, а для ссылочного – ссылка на эту ссылку. Ключевое слово **const** также означает передачу ссылки, но при этом передается не ссылка, а ее копия. В этом случае компилятор пресечет попытку процедуры изменить значение, выдавая сообщение об ошибке на стадии компиляции программы. Кроме того, для данных размерного типа с помощью **const** мы избежим ненужного копирования.

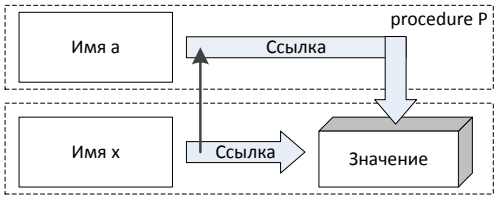
Фактический параметр ссылочного типа передается по ссылке даже при отсутствии указания ключевого слова **var** или **const**. Тем не менее, если в подпрограмме объект, переданный в качестве параметра, создается или полностью обновляется (SetLength для динамического массива, появление имени параметра в левой части оператора присваивания и т.п.), указывать ключевое слово **var** обязательно. Мы уже понимаем, почему: объект будет пересоздан, поэтому ссылка укажет на новое место в памяти. Не поставим **var** – вызванная подпрограмма не сможет изменить ссылку, и в вызывающей программе мы будем иметь либо старые данные, либо получим привет от сборщика мусора – программы из состава Microsoft .NET Framework, очищающей память от данных, которые считаются в дальнейшем ненужными.

Пусть рассмотренная выше процедура P имеет формальный параметр a, который принимается по ссылке. Заголовок такой процедуры можно записать как

```
procedure P(var a: integer);
```

Обратимся к процедуре P, передав ей в качестве фактического параметра значение переменной x: `P(x);`

При передаче параметра по ссылке копируется сама ссылка, поэтому и вызывающая программа, и процедура будут адресоваться к одному и тому же месту в памяти компьютера. Сами данные останутся на прежнем месте. Именно поэтому большие структуры данных передаются именно так: копируется лишь ссылка.



Процедура может как угодно менять значение «x», считая что это её собственное «a». Все происходит так, будто «x» становится еще одним именем данных, названных «a». Ниже приведены примеры записи процедур в обычном и упрощенном синтаксисе.

```
procedure Privet; // процедура без параметров
begin
    Println('PascalABC.NET приветствует Вас!')
end;

procedure Privet := Println('PascalABC.NET приветствует Вас!');

procedure ПлощадьПрямоугольника(Длина, Ширина : real;
        var Площадь : real);
begin
    Площадь := Длина * Ширина
end;

procedure ПлощадьПрямоугольника(Длина, Ширина : real;
        var Площадь : real) := Площадь := Длина * Ширина;

procedure ReadInt64(var a : int64);
begin
    Readln(a)
end;

procedure ReadInt64(var a : int64) := Readln(a);
```

Рассмотрите примеры правильного и неправильного вызова приведенных выше программ.

```
Privet;
var S: real;
ПлощадьПрямоугольника(120, 3*18, S); // Длина 120, Ширина 3*18
var r: integer;
ReadInt64(r); // неправильный тип r
var r: int64;
ReadInt64(r);
var S: real;
ПлощадьПрямоугольника(120, 3.1*18, 2+2); // нельзя передать 2+2
var S: integer;
ПлощадьПрямоугольника(120, 3*18, S); // неправильный тип S
```

4.2.3 Значение параметра по умолчанию

Процедура может иметь параметры, которые инициализируется значением путем его указания в заголовке. В этом случае говорят о значении параметра по умолчанию. Для таких параметров при вызове процедуры фактический аргумент можно не указывать. Отсюда следует правило, что формальные параметры, которые

имеют значения по умолчанию, должны указываться в списке параметров последними.

```
procedure Площадь(a: real; b: real := 0); // p04001
begin
  if b = 0 then Println('Площадь квадрата равна', a*a)
  else Println('Площадь прямоугольника равна', a*b)
end;

begin
  Площадь(46);  // второй параметр по умолчанию 0
  Площадь(30, 40)
end.
```

Имеется также возможность определять процедуры (и функции) с переменным числом параметров с указанием служебного слова **params** (см. 5.3.10).

Рассмотрим пример реализации такой функции. Возможно, вы обратили внимание, что стандартная функция Min(a,b) возвращает минимум лишь для двух аргументов. А если нужно получить минимум для большего их количества? Давайте напишем собственную функцию Min.

```
function Min<T>(params a: array of T) := a.Min;

begin
  Min(3.0, 4.9, 2.7, -5.1, 9.4, -2.0, 0.0, -8.2, 16.5).Println // -8.2
end.
```

Ваших знаний пока может оказаться недостаточно, чтобы понять эту функцию, но такой задачи в этом примере и не ставилось. «Все сразу, здесь и сейчас» - это плохой девиз для освоения языка программирования.

4.2.4 Процедуры без параметров

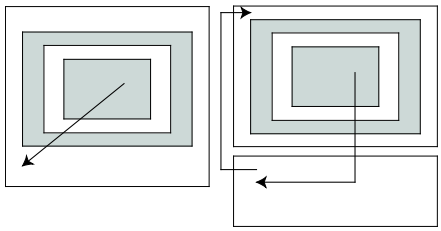
Параметры в процедурах служат для того, чтобы процедура могла обмениваться данными со своим окружением. Если такой обмен не нужен, то и параметры не нужны. Приведем пример процедуры, выводящей текущую дату и время.

```
procedure Сейчас := Println(DateTime.Now); // p04002

begin
  Сейчас
end.
```

4.2.5 Выход из вложенных циклов

В книге уже упоминалась проблема, связанная с досрочным выходом из цикла (см. 3.6.6). Когда циклы вложены один в другой, как можно прекратить работу всех циклов, находясь в самом внутреннем из них? Традиционно эта задача эффективно решалась применением оператора безусловного перехода **goto** к соответствующей ***метке***. Но мы изучаем современное программирование, в котором полезность

использования оператора **goto** признается достаточно спорной. В любом случае, полезно знать, как можно обойтись без этого оператора, а будете вы его использовать или нет – дело ваше.



Слева - выход по оператору goto на метку. Справа - вынесение циклов в процедуру с досрочным ее завершением по оператору Exit.

Вложенные циклы выносятся в отдельную процедуру. В этом случае выход из всех циклов с любого уровня вложенности равнозначен выходу из процедуры. Такой выход осуществляет оператор **exit**. Выполнение процедуры прерывается и осуществляется возврат в вызвавшую программную единицу.

4.3 Функции

Хорошая вещь процедура! Вот только, не всегда с ней удобно работать. Например, требуется найти частное от деления (15!-(a+2)!) на (a-2)!. Для тех, кто забыл, запись n! (произносится «эн факториал») обозначает произведение натуральных чисел от 1 до n включительно. Посмотрим, как в этом может помочь процедура.

```
procedure Fact(n: integer; var f: int64);  // Помещает n! в f
begin
  f := 1;
  for var i := 1 to n do
    f *= i
end;

begin
  var p, r: int64;
  var a := ReadInteger('a='); // ввод a
  Fact(15, p); // p := 15!
  Fact(a+2, r); // r := (a+2)!
  r := p-r;
  Fact(a-2, p); // p := (a-2)!
  Println(r / p)
end.
```

Привычный уже вопрос: что тут нехорошо? Нехорошо то, что формула растворилась в строках кода и надо ее «собирать», чтобы увидеть.

А если бы Fact была не процедурой, а функцией? Как квадрат, или корень, синус? Чтобы она не меняла значение переданного ей параметра, а при вызове возвращала результат вместо своего имени, и чтобы это имя можно было указывать прямо в выражении.

Подобную функцию можно записать так

```
function Fact(n: integer): int64;
begin
  Result := 1;
  for var i := 1 to n do
    Result *= i
end;

begin
  var a := ReadInteger('a='); // ввод a
  Println((Fact(15) - Fact(a + 2)) / Fact(a - 2));
end.
```

Не правда ли, код стал намного лучше?

Говоря о факториале, нельзя еще раз не отметить опасности, подстерегающей нас при выходе целочисленных значений за границу. Кардинально разрешает проблему использование значений типа **BigInteger**. Перепишем нашу функцию так, чтобы она могла вычислять факториалы больших чисел. Запустите эту программу и получите удовольствие от созерцания числа 500! Когда-то подобные задачи давали на олимпиадах…

```
function Fact(n: integer): BigInteger; // p04003
begin
  Result := 1;
  for var i := 1 to n do
    Result *= i
end;

begin
  Println(Fact(500));
end.
```

Как и процедура, функция состоит из заголовка и тела. Заголовок начинается служебным словом **function**, а заканчивается описанием типа значения, возвращаемого функцией в качестве результата. Список формальных параметров такой же, как у процедуры, но указание на вызов по ссылке (**var**) встречается нечасто. О переменной Result написано немного ниже.

Классическая функция принимает фактические параметры (в математике их называют аргументами) и возвращает свое значение, которое затем подставляется в месте вызова вместо имени функции. Менять значения фактических параметров классическая функция не должна. Но может. Такое поведение функции называется ***побочным эффектом***. Он и в самом деле какой-то «побочный»: только представьте, возводили x в квадрат посредством вызова Sqr(x), а значение x после вызова почему-то изменилось. Прямо как в афоризме у Козьмы Пруткова: «Щёлкни кобылу в нос — она махнет хвостом». Понятно, что писать функции с побочным эффектом нужно с большой осторожностью и только при серьезной необходимости. Злые языки говорят, что побочный эффект – это чтобы шпионов запутывать. А

если серьезно, иногда побочный эффект может оказаться полезным. Но увлекаться этим не следует, может оказаться себе дороже.

Как и процедуры, функции допускают при записи использовать упрощенный синтаксис, если вычисление функции сводится к выполнению единственного оператора присваивания. Разрешается также использовать автовыведение типа результата, возвращаемого функцией. Если результат, возвращаемый функцией, не нужен, ее можно вызвать, как процедуру.

В базовом Паскале при описании функции ее значение отождествлялось с именем.

```
function Cube(a: integer): integer;
begin
  Cube := a*a*a
end;
```

В целях совместимости такая запись разрешается, но считается устаревшей и не рекомендуется. В PascalABC.NET имя функции связывается со специальной переменной Result, тип которой определяется типом значения, возвращаемого функцией, поэтому объявлять переменную Result не требуется.

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

Конечно, можно записать эту функцию и в упрощенном синтаксисе:
function Cube(a: **real**) := a*a*a;

Рассмотрим еще одну программу, право на существование которой могут отрицать почитатели базового Паскаля. С непривычки может удивить, но ничего особенного в ней нет. Несколько нетипично использована функция ReadReal, позволяющая ввести значение переменной с приглашением ко вводу - но и только. Начинаете ощущать, что шуточные пожелания разработчикам создать язык, позволяющий писать программы в пару строк, о которых говорилось в Предисловии, оказываются на деле не такой уж и шуткой? А когда познакомитесь с лямбда-функциями, запись таких программ станет еще интереснее.

```
function Площадь(a, b: real) := a*b; // p04004

begin
  Println(Площадь(ReadReal('a='), ReadReal('b=')))
end.
```

Для любопытных «торопыжек» - один из возможных вариантов с лямбдой.

```
begin
  var Площадь: (real, real) -> real := (a, b) -> a*b;
  Println(Площадь(ReadReal('a='), ReadReal('b=')))
end.
```

4.4 Рекурсия

Имеется большой класс алгоритмов, в которых некое действие совершается путем повторного обращения к этому же алгоритму. Такие алгоритмы носят название рекурсивных, а подпрограммы, реализующие подобные алгоритмы – ***рекурсивными***.

Вот пример рекурсивного определения натурального числа: «*Число* - это одна из цифр от 0 до 9, за которой может следовать *число*». Здесь термин «число» определяется через свое собственное определение.

Рекурсия часто порождается рекуррентными математическими формулами. Вот популярный пример рекурсивного определения факториала

$$n! = \begin{cases} 1, \text{ при } n \le 1 \\ n(n-1)!, \text{ при } n > 1 \end{cases}$$

Реализация рекурсивных функций и процедур в PascalABC.NET не вызывает никаких проблем.

```
function Fact(n: integer): BigInteger := n <= 1 ? 1 : n*Fact(n-1);

begin
  var n := ReadInteger('n=');
  Writeln(n, '! = ', Fact(n))
end.
```

Не забывайте в записи рекурсивных функций с упрощенным синтаксисом явно объявлять тип результата, иначе компилятор выдаст ошибку.

4.5 Опережающее объявление подпрограмм

Если подпрограмма A вызывает подпрограмму B, а подпрограмма B вызывает подпрограмму A, то одна из подпрограмм входит в противоречие с требованием быть описанной до первого обращения. Рассмотрим пример из Справки PascalABC.NET

```
procedure A(i: integer);
begin
  ...
  B(i-1); // к этому моменту B должна быть описана
end;

procedure B(i: integer);
begin
  ...
  A(i div 2); // а этому моменту A должна быть описана
end;
```

При попытке откомпилировать программу с таким описанием процедур получим в строке с кодом `B(i-1);` сообщение «Неизвестное имя 'B'». Это ожидаемо.

Выходом из ситуации служит ***опережающее объявление*** подпрограммы, состоящее из ее заголовка, за которым следует ключевое слово **forward**.

```
procedure B(i: integer); forward; // опережающее объявление
procedure A(i: integer);
begin
  ...
  B(i-1);
end;
procedure B(i: integer);
begin
  ...
  A(i div 2);
end;
```

4.6 Перегрузка имен подпрограмм

Начнем с обсуждения конкретной проблемы. Пусть нам требуется функция, назовем ее Мин, которая возвращает минимум переданных ей значений. При попытке реализовать эту функцию возникают две проблемы.

Первая проблема – количество и тип параметров. Мы будем находить минимум скольких значений – двух, трех, десяти? Какой тип будут иметь эти значения? Могут ли значения иметь разные типы? Вторая проблема – тип результата. Конечно, он зависит от решения первой проблемы. Но все же, если параметры разнотипные, каким должен быть тип результата?

Казалось бы, простейшая задача – найти минимум. Но сколько проблем надо решить, прежде чем приступать к программированию! Допустим, мы решили скромно ограничиться следующими требованиями (программисты часто называют это ***спецификацией***): функция может иметь два, три или четыре параметра типа real и возвращает результат такого же типа. Напишем три функции с именами Мин2, Мин3 и Мин4 ? Так можно поступить и именно так поступали в системе программирования Turbo Pascal. Современный подход позволяет обойтись одним общим именем Мин для всех функций, сколько бы их не было. И позволяет это сделать ***перегрузка имени***. Здесь термин «перегрузка» не имеет ничего общего с физическими силами, а обозначает переопределение имени функции в зависимости от параметров в ее заголовке. Скорее это не «перегрузка», а «перезагрузка» одноименной программной единицы с другим кодом.

При программировании в PascalABC.NET для организации перегрузки специально ничего не нужно делать. В диалектах Free Pascal и Delphi существовало ключевое слово **overload**; его использование поддерживается исключительно в целях совместимости.

Посмотрим, как может выглядеть программа с перегрузкой имен функций.

```
function Мин(a, b: real) := a < b ? a : b; // p04005

function Мин(a, b, c: real) := Мин(Мин(a, b), c);

function Мин(a, b, c, d: real) := Мин(Мин(a, b), Мин(c, d));

begin
  Println(Мин(15, -7.04), Мин(4, 0, 9), Мин(3 - 5.1, 17 / 4, 5, 3 – 2 * 2))
end.
```

Здесь использовано многое из материала текущей части, поэтому рассмотрим приведенный код. Описаны три одноименные функции Мин, отличающиеся набором параметров. Следовательно, будет использоваться перегрузка имени. Все три функции записаны в упрощенном однострочном синтаксисе. Функции от трех и четырех параметров содержат обращение к одноименной функции от двух параметров, но это не рекурсия, потому что в теле функции нет обращения к ней самой. Тип значения **real**, возвращаемого функцией, автоматически выводится из значения выражения в теле функции. Передача параметров производится по значению (в описании формальных параметров отсутствует **var**), поэтому в качестве фактических параметров разрешено указывать произвольные выражения, автоматически приводящиеся к типу **real**, например, целочисленные.

Имена процедур перегружаются аналогичным образом.

4.7 Подпрограмма в качестве параметра

До сих пор мы рассматривали числовые параметры. На самом деле они могут быть любого известного в программе типа. В том числе – процедурного типа (и такие тоже существуют). Иными словами, с помощью параметров мы можем передавать в подпрограммы имена процедур и функций, к которым подпрограмма сможет обратиться. Рассмотрим пример.

$$Сумма = \sum_{i=m}^{n} a_i,\ a_i = \frac{5i^2 - 2}{3i + 1}$$

Опишем функцию Сумма, умеющую вычислять суммы числовых последовательностей, элементами a_i которых могут быть разнообразные выражения, зависящие от своего порядкового номера в последовательности. В качестве параметров будем передавать начальный номер элемента последовательности, конечный номер и имя функции, возвращающей значение элемента.

```
function p(i: integer) := (5 * i * i - 2) / (3 * i + 1); // p04006

function Сумма(m, n: integer; f: function(r: integer): real): real;
begin
  Result := 0;
  for var i := m to n do
    Result += f(i)
end;

begin
  Println(Сумма(3, 10, p)); // от 3 до 10, p – имя функции
end.
```

В случае изменения формулы a_i достаточно внести изменение в тело функции p. В заголовке описания функции Сумма указан формальный параметр f, которому при вызове будет сопоставлено имя процедуры p. Следует соблюдать соответствие количества, типа и порядка следования фактических параметров формальным. Поскольку процедурный тип данных является ссылочным, функция передается по ссылке несмотря на отсутствие ключевого слова **var**.

4.8 Процедурная переменная

Поскольку функция и процедура могут быть параметрами, совершенно логично иметь возможность определить в программе переменную, значением которой будет ссылка на подпрограмму. Такие переменные называются ***процедурными*** и имеют ссылочный тип.

Процедурная переменная хранит действие, которое совершит подпрограмма при вызове. Поскольку заранее неизвестно, когда это произойдет, действие называется ***отложенным***. Отложенное действие можно произвести, указав на месте процедуры или функции процедурную переменную. Это действие может быть совершено и в некоторой подпрограмме, в которую процедурная переменная передается в качестве параметра. В таком случае действие называется ***обратным вызовом*** (callback).

Процедуры без параметров можно складывать между собой и умножать на числовое значение. При сложении процедуры будут выполнены друг за другом, при умножении на число – циклически вызваны соответствующее число раз. Конечно, подобные выражения нельзя писать в виде самостоятельных операторов программы, они должны присваиваться процедурной переменной, которая затем вызывается, либо заключаться во внешние круглые скобки, что делает ее выражением, подлежащим обработке.

Рассмотрим пример программы с использованием процедурных переменных. Первые три процедуры выводят на монитор значения 1, 2 и «пробел». Процедура Go получает в качестве своего параметра p ссылку на процедурную переменную, которую автоматически формирует компилятор при указании выражения с процедурными переменными в качестве фактического параметра.

```
procedure p1:=Write(1);

procedure p2:=Write(2);

procedure ps:=Write(' ');

procedure Go(p:procedure):=p; // выполняет p

begin
  var q:=(p1+p2+2*p1+ps)*5; // отложенное присваивание
  q; // 1211 1211 1211 1211 1211 (результат обратного вызова)
  Println;
  ((2 * p2 + p1 * 3) * 2); // 2211122111 (внешние скобки обязательно)
  Println;
  Go(p2*4+ps+p1*3+ps+6*p1) // 2222 111 111111
end.
```

Второй пример основан на приведенной в 4.7 программе для вычисления суммы. Здесь введена процедурная переменная t. Ее тип автовыводится вследствие присваивания имени процедуры p. Далее, значение переменной t переопределяется присваиванием имени процедуры q. В обоих случаях вызов функции Сумма происходит с одним и тем же фактическим параметром t..

```
function p(i: real) := (5 * i * i - 2) / (3 * i + 1); // p04007

function q(i: real) := i * i;

function Сумма(m, n: integer; f: function(r: real): real): real;
begin
  Result := 0;
  for var i := m to n do
    Result += f(i)
end;

begin
  var t := p;
  Println(Сумма(3, 10, t));
  t := q;
  Println(Сумма(1, 20, t));
end.
```

Есть ли во всем этом какой-то смысл? Оказывается, есть и очень большой. Он станет понятен при прочтении следующего раздела. Лямбда-выражения позволяют не описывать функций отдельно, а создавать их по необходимости «на лету». С их использованием можно записать эту программу и короче, и нагляднее.

```
function Сумма(m, n: integer; f: real -> real): real; // p04008
begin
  Result := 0;
  for var i := m to n do
    Result += f(i)
end;

begin
  var t: real -> real := i -> (5 * i * i - 2) / (3 * i + 1);
  Println(Сумма(3, 10, t));
  t := i -> i * i;
  Println(Сумма(1, 20, t));
end.
```

В самом деле, «исчезли» строки описания функций p и q, определение значений t происходит непосредственно перед вызовом, что позволяет не искать место описания передаваемой функции. Пора познакомиться с этими замечательными «лямбдами» поближе.

4.9 Лямбда-выражения

Лямбда-выражения (или просто лямбды) – это термин из парадигмы функционального программирования. Объем книги и предполагаемая читательская аудитория не позволяют остановиться на ***лямбда-исчислении*** – этом достаточно интересном разделе математики, поэтому за более фундаментальными сведениями вам придется обращаться к другим источникам.

Теория функционального программирования предполагает сведение любого алгоритма к вычислению набора некоторых функций, параметрами которых являются исходные данные и результаты вычисления других функций. Это, строго говоря, не предполагает хранения в процессе вычисления каких-либо промежуточных данных, и тем более, каким-либо образом их изменения. В императивном программировании функции могут иметь побочный эффект и оперировать некоторыми внешними по отношению к своему телу данными (это связано с ***областями видимости*** – см. часть 11), но в функциональном программировании такое невозможно. Здесь функция всегда работает одинаково и результат ее работы зависит лишь от полученных параметров. Эта особенность повышает надежность программирования, поскольку функцию достаточно отладить один раз на нужном наборе аргументов. В чистом функциональном языке отсутствуют присваивания, поэтому переданный в функцию параметр изменить невозможно. Нет в функциональном языке и циклов – вместо них используются рекурсивные функции, обеспечивающие нужное число повторений. Функциональная парадигма позволяет составлять программу в стиле «что сделать?», который намного проще императивного «как сделать?». Программа выглядит короче и понятнее. А то, что коротко, быстро писать, быстро читать и удобно понимать.

PascalABC.NET, являясь мультипарадигменным языком, содержит лишь некоторые элементы функционального программирования. Практически все они строятся на основе ***лямбда-выражений***.

Лямбда-выражение представляет собой некоторое безымянное выражение, отражающее функциональную зависимость. На основе этого выражения компилятор формирует функцию и затем подменяет лямбда-выражение именем этой функции. Конечно, это лишь схематическое представление, приведенное для понимания сути происходящего.

Лямбда-выражение легко распознается в тексте программы по характерному знаку операции ->, который произносится «переходит в … ». Поскольку лямбда-выражение отражает функциональную зависимость, проще всего его и рассматривать, как аналог функции.

Синтаксис лямбда-выражений весьма сложен, поэтому он будет рассматриваться по мере необходимости и на примерах. Полный синтаксис приведен в Справочной системе PascalABC.NET.

```
x -> x * x * x + 5 * x - 3 // x переходит в x^3+5x+3
```

Это не что иное, как аналог следующей функции, пусть ее имя будет Foo:

```
function Foo(x: integer) := x * x * x + 5 * x - 3;
```

В отличие от описания функции Foo, в лямбда-выражении мы не указали тип аргумента x, а имя функции лямбда-выражению не нужно: его компилятор назначит сам.

Если функция имеет два или более аргумента, используются круглые скобки

```
(x, y) -> (2 * x – 3 * y + x / y)
```

И здесь можно провести полную аналогию с функцией

```
function Foo(x, y: real) := 2 * x – 3 * y + x / y;
```

Как видно из этих примеров, лямбда-выражение внешне отличается от функции, но смысл несет такой же.

Практически можно считать, что лямбда-выражение – это «одноразовая функция», не требующая описания в разделе подпрограмм и выступающая в программе наравне с обычными переменными. Принято говорить, что лямбда-выражение является ***объектом первого класса***, поскольку оно может быть сохранено в переменной, передано в подпрограмму и/или возвращено ей и может быть создано во время выполнения программы.

Рассмотрим примеры работы с лямбда-выражениями.

Пример 1. При помощи лямбда-выражения описана функция, которая присвоена переменной F. В описании F имеет непривычный тип **integer -> integer**. Мысленно

проговорим это выражение: «**integer** переходит в **integer**». Одна переменная на входе, результат на выходе, следовательно, это функция одной переменной.

```
begin
  var F: integer -> integer := x -> 3 * x - 1;
  Println(5 + F(4) - F(7))
end.
```

Пример 2. Этот пример уже интереснее. Вначале процедурная переменная F получает значение лямбда-выражения x -> 3 * x - 1. После вычисления и вывода значения F(4) делается присваивание ей описания другого лямбда-выражения, определенного с параметром p (как назвать, разницы нет, это же формальный параметр). Теперь F(28) вычисляется как 28 mod 10, по последнему присвоенному процедурной переменной лямбда-выражению

```
begin
  var F: integer->integer := x -> 3 * x - 1;
  Println(F(4));
  F := p -> p mod 10;
  Println(F(28))
end.
```

Пример 3. Лямбда-выражение, содержащее условную операцию.

```
begin
  var F: integer->integer := q -> q < 0 ? 10 * q : 3 * q - 1;
  Println(F(-4) + F(4));
end.
```

Пример 4. Лямбда-выражение без параметров (F – случайное число на [1;100])

```
begin
  var F: () -> integer := () -> Random(1, 100);
  Println(2 * F, F - 50);
end.
```

При использовании лямбда-выражений, описывающих функции без параметров, следует избегать указания имен таких функций в качестве единственного элемента в выражении, например, `Println(F)`. В подобных случаях указывайте пустую пару круглых скобок, уточняющих, что это имя функции: `Println(F())`.

4.9.1 Лямбда-функции и лямбда-процедуры

По своей природе лямбда-выражение ближе к функции, поскольку при его вычислении для заданного набора аргументов получается некоторое значение. При вызове процедуры, как мы помним, значение не возвращается.

Рассмотрим пример с лямбда-выражением вида i -> 2 * i - 1, порождающим натуральные нечетные числа для i = 1, 2, 3, ... Его можно рассматривать (и использовать) как некую функцию F(i) = 2 * i - 1. Такую функцию можно присвоить процедурной переменной, а можно передать непосредственно в вызываемую подпрограмму. Вы вольны выбирать любой из трех представленных способов

передачи «лямбды» в процедуру, а мне нравится последний, с непосредственной передачей.

Лямбда-выражение возвращает некоторое значение. Процедура значения не возвращает. Чтобы лямбда-выражение считалось процедурой, нужно заключить его правую часть в операторные скобки. Например, так: `i -> begin Print(2 * i - 1) end`. Описание лямбда-процедуры с одним параметром могло бы выглядеть как integer -> (). Именно такая процедура использована в приводимом примере.

```
procedure PRT(f: integer -> integer); // p04009 параметр - функция
begin
  for var i := 1 to 10 do
    Print(f(i));
  Println
end;

begin
  // предварительное описание процедурной переменной
  var t1: integer -> integer;
  t1 := i -> 2 * i - 1;
  PRT(t1);
  // описание с инициализацией
  var t2: integer -> integer := i -> 2 * i - 1;
  PRT(t2);
  // непосредственная передача лямбда-функции в процедуру
  PRT(i -> 2 * i - 1);
end.
```

```
procedure PRT(f: integer -> ()); // p04010 параметр - процедура
begin
  for var i := 1 to 10 do
    f(i);
  Println
end;

begin
  // предварительное описание процедурной переменной
  var t1: integer -> ();
  t1 := i -> begin Print(2 * i - 1) end;
  PRT(t1);
  // описание с инициализацией
  var t2: integer -> () := i -> begin Print(2 * i - 1) end;
  PRT(t2);
  // непосредственная передача лямбда-процедуры в процедуру
  PRT(i -> Print(2 * i - 1));
end.
```

4.9.2 Захват переменной

Лямбда-выражения могут содержать внутри себя обращения к переменным, которые им доступны. Доступность переменной определяется областью ее действия.

Пока будем считать, что область действия переменной (т.е. место, где она может быть использована) располагается от места ее описания до конца блока, в котором переменная описана. Использование в лямбда-выражении таких переменных, внешних по отношению к нему, называется ***захватом переменных***. При изменении значения захваченной переменной, в лямбда-выражении будет использоваться новое значение.

Рассмотрим пример нахождения суммы вводимых чисел с помощью лямбда-процедуры. Суммирование ведем до тех пор, пока не будет введен ноль.

```
begin // p04011
  var s := 0.0; // вещественное
  var x: real;
  var AddReal: real -> () := p -> begin s += p end;
  repeat
    x := ReadReal;
    AddReal(x)
  until x = 0;
  s.Println
end.
```

Пример сеанса работы с программой

```
1
2
3
4
0
10
```

Вводить данные можно и в строку – оператор Read это допускает.

```
4 7 -2 1.18 4.1523 0.2342 0
14.5665
```

4.10 Первое понятие класса

Классы – это основополагающая категория объектно-ориентированной парадигмы в программировании. Подробно классы рассматриваются в части 13, а здесь будут даны некоторые понятия, необходимые для дальнейшего изучения материала.

В объектно-ориентированном программировании (ООП) объекты представляются программисту именованными элементами, созданными на основе определенных классов. Класс – это абстрактное описание наподобие чертежа, по которому конструируется конкретный объект. Класс содержит в себе описания данных и средств управления этими данными. Данные в классе называются ***полями***, а средства для манипуляции полями называются ***методами***. Поле – это некоторый аналог переменной в процедурной парадигме, а метод можно сравнить с подпрограммой.

При обращении к полям и методам объекта используется ***точечная нотация***. После имени объекта (или класса) ставится разделяющая точка, вслед за которой указывается имя поля или метода. Например, все рассмотренные нами числовые типы являются классами, поэтому переменные этих типов – объекты, имеющие некоторые поля и методы. Обращение к полям мы уже встречали, когда знакомились с константами вида T.V, например `real.MaxValue` означает обращение к полю `MaxValue` класса **real**.

Полезно помнить, что каждый базовый (следовательно, заранее известный в программе) класс имеет методы `Print` и `Println`, позволяющие выводить значение объекта с последующим пробелом или сменой строки, как это делают стандартные процедуры Print и Println. Например, мы можем вывести значение `real.MaxValue`, записав `real.MaxValue.Print`. Если воспринимать точку, как разделитель в перечислении, такая запись читается вполне логично: «Класс **real**, поле `MaxValue`, вывести». Точки проявляют тенденцию множиться, но это нормальное явление и в дальнейшем станут понятны все преимущества такой записи, позволяющей в одной строке кода реализовать весьма большой алгоритм.

Особый вид «программистского хулиганства» может быть достигнут благодаря тому, что Print работает как функция, возвращающая в качестве результата переданные ей данные. Посмотрите пример, но не поступайте так, пока не научитесь глубоко понимать механизм происходящего.

```
begin // p04012
  integer.MaxValue.Print.Println.Sqrt.Println.Sqr.Println
end.
```

И это Паскаль? Да, современный Паскаль. Программа выведет следующие строки:

```
2147483647 2147483647
46340.950001052
2147483647
```

Рассмотрим, как получились такие результаты. Значение `integer.MaxValue` передается методу Print, который выводит это значение, и передает дальше в виде объекта класса **integer** с некоторым внутренним именем, например, `X`. Затем формируется выражение вида `X.Println`. Оно обращается к методу Println класса **integer** и выводит значение `X`, по-прежнему равное `integer.MaxValue`. И снова это значение передается дальше в виде объекта `X`. Получается конструкция вида `X.Sqrt`. Объект `X` автоматически приводится к классу **real**, как того требует аргумент метода Sqrt. Вычисляется значение квадратного корня и возвращается, как объект `X` класса **real**. Вновь получаем `X.Println`, значение `X` выводится, а объект `X` передается дальше. Остается выполнить `X.Sqr.Println`. Получаем с некоторой точностью исходное значение, но уже в типе **real**, и выводим его.

Если вы поняли все написанное, то уже можете пугать этой программой ничего не подозревающего школьного учителя информатики, почему-то все еще рассказывающего про Turbo Pascal, в то время как в компьютерном классе установлена достаточно свежая версия PascalABC.NET.

Каждый тип, реализованный в виде класса, можно расширить путем добавления новых методов, которые в этом случае называются ***методами расширения*** или просто ***расширениями***. В PascalABC.NET имеется большое количество расширений классов Microsoft .NET Framework, а также собственных классов. Как и любой метод, расширение записывается через точку. При изучении ООП вы научитесь создавать собственные методы расширения, используя ключевое слово **extension-method**. В работе программисту обычно непринципиально знать, использует ли он метод класса или расширение.

4.11 Для самостоятельного решения

Т4.1. Каждая вершина треугольника ABC задана парой вещественных координат x и y. Написать программу, позволяющую находить площадь треугольника ABC. В программе использовать функцию нахождения площади по формуле Герона:

$$S = \sqrt{p(p-a)(p-b)(p-c)};\ \ p = \frac{a+b+c}{2}$$

Здесь a, b, c – длины сторон треугольника. Расстояние между точками A и B, каждая из которых задана своими координатами, может быть определено по формуле

$$L_{AB} = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}$$

Т4.2. Написать программу, вычисляющую периметр и площадь квадрата, прямоугольника или треугольника, в зависимости от количества введенных ненулевых вещественных значений – стороны квадрата, длины и ширины прямоугольника, либо длин сторон треугольника. Площадь треугольника можно вычислить по формуле Герона. Использовать перегрузку подпрограмм в зависимости от числа параметров.

Т4.3. Запрограммировать функцию, возвращающую номер координатной четверти по координатам x и y точки. С помощью этой функции определить, каким четвертям принадлежит набор из 10 случайных точек, координаты которых принадлежат интервалу [-15.00; 15.00] и заданы с точностью до двух знаков после запятой.

Т4.4. Используя условие предыдущей задачи, определить наиболее удаленную от начала координат точку, не принадлежащую III четверти.

Т4.5. Вычислите значение F(100, 200, 300), для следующей функции

$$F(a,b,c) = \begin{cases} 1, \text{ при } a \le 0 \ \vee\ b \le 0 \ \vee\ c \le 0 \\ F(a,b,c-1) + F(a,b-1,c-1) - F(a,b-1,c), \text{ при } a < b < c \\ F(a-1,b,c) + F(a-1,b,c-1) - F(a-1,b-1,c-1) \text{ в прочих случаях} \end{cases}$$

Т4.6. Решить задачу Т3.7 из части 3 «Логика в программе», составив рекурсивную функцию, вычисляющую цифровой корень числа типа **int64**.

Часть 5

Последовательности

Что бы нового мы ни создавали, мы должны дать людям возможность переходить от старых инструментов и идей к новым.

Бьёрн Страуструп, автор языка C++

Индекс массива должен начинаться с 0 или 1? Мой компромисс по поводу 0,5, я считаю, был отвергнут без надлежащего рассмотрения.

Стэн Келли-Бутл,
автор книги «Введение в UNIX»

Последовательности в реальной жизни встречаются настолько часто, что мы не обращаем на это внимания. Привычная фраза в математике: «Пусть дана следующая последовательность чисел…». Тарелки в стопке посуды, которую предстоит перемыть. Список рейсов самолетов, прибывающих в аэропорт. Книги, стоящие на полке. Можно продолжать бесконечно.

Последовательность в программе – это объект в памяти компьютера, состоящий из однотипных элементов, которые можно перебирать в порядке от первого элемента к последнему. Перебор завершается на последнем элементе, либо может быть завершен досрочно при выполнении некоторого условия, что позволяет работать с последовательностями, содержащими неизвестное количество элементов и даже бесконечными. Простой пример: принимать с клавиатуры натуральные числа, пока не будет введен ноль или отрицательное число.

В PascalABC.NET включены собственно последовательности **sequence**, статические массивы, динамические массивы и множества. Кроме этого, можно напрямую обращаться к стандартным коллекциям Microsoft .NET Framework, реализованным в классах Queue (очередь), List (односвязный список), LinkedList (двусвязный список), SortedList (упорядоченный список), Dictionary (словарь) и некоторым другим (см. часть 10).

Для программиста все эти виды последовательностей отличаются способом, с помощью которого можно получить доступ к элементу последовательности. Особенность работы с любой последовательностью состоит в том, что в один момент времени имеется доступ только к одному ее элементу.

Если имеются последовательности R и S, то операция присваивания S := R не является операцией поэлементного копирования из R в S. В S помещается лишь ссылка на R, так что после этого R и S будут двумя именами одной и той же последовательности. Вследствие этого, любые изменения, если они будут сделаны в S, отразятся и в R. Это объясняется тем, что все последовательности имеют ссылочный тип. Исключение составляют статические массивы, имеющие размерный тип: для них выполняется именно поэлементное копирование.

5.1 Последовательности sequence

В последовательности sequence доступ к нужному элементу можно получить только в путем перебора всех предшествующих элементов. Перебор осуществляет оператор цикла **foreach**. Поскольку перебор может осуществляться достаточно долго, а элементы последовательности sequence в памяти не сохраняются, вся необходимая работа обычно выполняется за единственный просмотр последовательности.

Да, именно так: элементы последовательности sequence (далее по тексту части – последовательность) в памяти компьютера не сохраняются. До просмотра последовательности элементы могут где-то храниться (компилятор нам про это не сообщает), но могут и не храниться вообще. Последовательность может задаваться набором литералов при ее описании, совмещенном с инициализацией. Также, можно ввести элементы последовательности с клавиатуры или прочитать их из файла. Можно получить готовую последовательность из подпрограммы. И можно написать формулу, по которой вычисляется любой i-й член последовательности – тогда будет храниться подпрограмма, реализующая эту формулу. В последнем случае, если значениями элементов будут числа, последовательность называют ***числовой***.

Последовательность имеет длину – количество элементов в последовательности, причем длина может быть и неограниченной. Например, натуральный ряд чисел – это последовательность неограниченной длины. Пустая последовательность имеет длину, равную нулю. Тип последовательности определяет тип каждого из ее элементов. Созданная последовательность доступна только для чтения, изменить в ней ничего нельзя.

Важная особенность последовательности в PascalABC.NET состоит в том, что она является лишь отображением некоторого множества элементов, получаемых на основе перебора их порядковых номеров, а не самими элементами. Например, для числовой последовательности отображением может служить формула. Пусть дана формула, по которой можно получить k-й элемент последовательности a_k натуральных нечетных чисел: $a_k = 2 * k - 1$, где $k = 1, 2, \ldots$ Для такой последовательности не требуется отводить место в памяти и не нужно вычислять все ее элементы: достаточно в лишь обратиться к приведенной формуле. И памяти даже под бесконечную последовательность достаточно отвести лишь столько, сколько занимает один ее элемент.

В PascalABC.NET элементы последовательности не хранятся – хранится лишь способ их получения.

Мы определили последовательность, как совокупность элементов, которые можно перебирать. Перебор по своей природе – циклический процесс. PascalABC.NET поддерживает элементы функционального программирования, что часто позволяет обходиться без использования операторов цикла. Поскольку элементы последо-

вательности в памяти не хранятся, их желательно перебрать за один раз. Но «желательно» вовсе не означает «обязательно».

Если элементы последовательности вводятся с клавиатуры или читаются из файла, при их повторном переборе придется повторять ввод. Если элементы последовательности создаются на основе генератора псевдослучайных чисел, при их повторном переборе элементы будут иметь совсем другие значения. Такие последовательности желательно обрабатывать за один проход, так что если в PascalABC.NET нет готовой подпрограммы, реализующей нужный алгоритм, придется использовать традиционные циклы, либо написать собственную подпрограмму для реализации функционального стиля программирования.

Будем называть последовательность ***недетерминированной***, если хотя бы один ее элемент может изменить свое значение при повторном просмотре этой последовательности. Такие последовательности порождаются, в частности, путем чтения внешних данных (из-за возможных ошибок повторного ввода) или обращением к датчику псевдослучайных чисел. Помните, что даже самый обычный вывод элементов недетерминированной последовательности требует ее просмотра!

Будем называть последовательность ***детерминированной***, если независимо от количества ее просмотров ни один элемент не может изменить своего значения. Детерминированные последовательности создаются на основе формул, определяющих значение элемента в зависимости от его порядкового номера, на основе перечисления литералов в программном коде и некоторыми другими способами.

5.1.1 Создание последовательностей

Последовательность описывается с помощью ключевых слов **sequence of**:

```
var s: sequence of int64;
var q: sequnce of real;
```

Функция Seq(список типа T) позволяет создать последовательность, состоящую из значений элементов типа T, перечисленных в списке через запятую.

```
begin // p05001
  var a: sequence of integer;
  a := Seq(1, 8, -10, 14, 23, 19, 126);
  Println(a)
end.
```

Последовательность a будет состоять из перечисленных в Seq элементов, причем порядок элементов в ней будет совпадать с указанным при создании. Оператор Println выводит все элементы последовательности, заключая их в квадратные скобки: [1, 8, -10, 14, 23, 19, 126].

Описание последовательности можно объединять с ее инициализацией; при этом тип можно указывать явно или использовать автовыведение типа.

```
begin
  var P := Seq(0.25, -3.173, 0.000343, 100e-8, 145.23);
  Println(P)
end.
```

В приведенном примере будет выведено [0.25, -3.173, 0.000343, 1E-06, 145.23].

5.1.1.1 Генераторы

Инициализировать последовательности путем прямого перечисления значений их элементов на практике приходится нечасто. Намного чаще последовательности описываются некоторой математической зависимостью, в которой значение элемента вычисляются в зависимости от его порядкового номера. Подпрограммы, порождающие элементы последовательности, называются генераторами. Рассмотрим несколько функций-генераторов, реализованных в составе PascalABC.NET.

Для целых значений параметров генерируются последовательности элементов типа **integer**:

- `Range(a,b)` – значения от a до b с шагом 1;
- `Range(a,b,h)` – на интервале [a;b] с шагом h;
- `SeqFill(n,a)` – n элементов со значением a;
- `SeqRandomInteger(n,a,b)` – n случайных чисел на интервале [a; b]. Можно также использовать имя `SeqRandom`.

Пример программы.

```
begin // p05002
  var a := Range(-10, 7);
  Println(a);
  Range(14, 53, 6).Println;
  SeqFill(7, 777).Println(',');
  var r := SeqRandomInteger(5, -99, 99);
  loop 3 do
    Println(r)
end.
```

Полученные результаты:

```
[-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7]
14 20 26 32 38 44 50
777,777,777,777,777,777,777
[-6,59,38,50,-51]
[-29,-21,-33,28,-20]
[-64,61,82,-82,-17]
```

Содержание трех первых строк вывода понятно и ожидаемо. Обратите внимание, что метод .Println выводит последовательность без внешних скобок, но через пробел. Если пробел в качестве разделителя элементов не устраивает, можно в методе записать желаемый разделитель, указав его в одинарных кавычках. Почему различается содержание следующих трех, понять сложнее. В цикле трижды выводятся элементы последовательности r. Эта последовательность недетерминированная, поэтому три прохода по ней порождают три различных набора элементов.

Для генерации последовательностей элементов типа **integer** можно также использовать расширения.

- `n.Range` – возвращает последовательность 1, 2, 3, … n;
- `n.Times` – возвращает последовательность 0, 1, 2, … n-1;
- `a.To(b)` – возвращает последовательность a, a+1, a+2, … b;
- `a.DownTo(b)` – возвращает последовательность a, a-1, a-2, … b.

Генераторы последовательностей типа **real**:

- `PartitionPoints(a,b,n)` – на интервале `[a;b]`, разбитом на `n` отрезков; `a` и `b` типа **`real`**;
- `SeqFill(n,a)` – `n` элементов со значением `a` типа **`real`**;
- `SeqRandomReal(n,a,b)` – `n` случайных чисел на интервале `[a;b]`; `a,b` – типа **`real`**.

Генератор PartitionPoints – отличное средство для табуляции функций, позволяющее просто и эффективно создавать нужный набор аргументов. Если табуляция выполняется на интервале [a; b] с шагом h, достаточно записать генератор PartitionPoints(a, b, Trunc((b - a) / h)). Нужно иметь в виду, что если интервал b-a не кратен шагу, шаг будет изменен для обеспечения такой кратности. Такое поведение приемлемо не всегда.

5.1.1.2 Генераторы, использующие лямбда-выражения

Это самая мощная разновидность генераторов. Можно сгенерировать любую последовательность, для которой имеется математическая зависимость.

1. SeqGen(n, **integer** -> T) – последовательность типа T из n элементов; лямбда-выражение задает преобразование из i = 0, 1, … n-1 функцией F(i). Здесь i – номер элемента в последовательности – именно он указывается в «лямбде» с типом **integer**.

```
begin // p05003
  var s := SeqGen(10, i -> 2 * i + 1); // 10 нечетных целых
  Println(s);
  s := SeqGen(10, i -> 2 * i); // 10 четных целых
  Println(s);
  s := SeqGen(10, i -> i < 5 ? 2 * i + 1 : 2 * i ); // 2 x 5
  Println(s)
end.
```

Будут получены три строки:

```
[1,3,5,7,9,11,13,15,17,19] – 10 первых нечетных натуральных чисел
[0,2,4,6,8,10,12,14,16,18] – 10 первых неотрицательных целых чисел
[1,3,5,7,9,10,12,14,16,18] – 5 первых нечетных и 5 последующих четных.
```

2. SeqGen(n, **integer** -> T, m) – последовательность типа T из n элементов; лямбда-выражение задает преобразование из i = m, m+1, … m+n-1 функцией F(i). От предыдущего отличается тем, что m задает начальное значение i.

3. SeqGen(n, m, T -> T) – последовательность типа T из n элементов, начиная от значения m типа T; лямбда-выражение задает преобразование от предыдущего элемента к следующему. Генератор используется, когда имеется формула, связывающая функциональной зависимостью текущий член последовательности с предыдущим. Такая формула в математике называется ***рекуррентной***.

Рассмотрим программу получения значений первых десяти членов последовательности, в которой первый член равен 7.1, а каждый последующий получается путем увеличения предыдущего на 0.3.

```
begin
  SeqGen(10, 7.1, x -> x + 0.3).Println
end.
```

Программа выведет значения 7.1 7.4 7.7 8 8.3 8.6 8.9 9.2 9.5 9.8

4. SeqGen(n, m, k, (T,T) -> T) – последовательность типа T из n элементов, в которой заданы значения первого элемента m и второго k (оба типа T); лямбда-выражение задает преобразование от пары предыдущих элементов к следующему. Генератор используется, когда имеется формула, связывающая функциональной зависимостью текущий член последовательности с двумя предшествующими.

Примером такой последовательности может служить ряд чисел Фибоначчи: 1, 1, 2, 3, 5, 8, … Здесь текущий элемент равен сумме двух предыдущих.

```
begin
  SeqGen(13, 1, 1, (i , j) -> i + j).Println
end.
```

Программа выведет значения 1 1 2 3 5 8 13 21 34 55 89 144 233

5. SeqWhile(m, T -> T, T -> **boolean**) – последовательность элементов типа T, начиная от элемента со значением m. Первое лямбда-выражение задает преобразование от предыдущего элемента к следующему, ложное значение второго определяет завершение генерации последовательности. Своеобразный вариант цикла **while**. Генератор используется, когда имеется формула, связывающая функциональной зависимостью текущий член последовательности с предыдущим, но длина последовательности неизвестна.

Ниже приведена программа вывода последовательности степеней двойки, не превышающих значение 1628.

```
begin
  SeqWhile(1, i -> 2 * i, i -> i <= 1628).Println
end.
```

Программа выведет последовательность 1 2 4 8 16 32 64 128 256 512 1024

6. SeqWhile(m, k, (T,T) -> T, T -> **boolean**) – последовательность элементов типа T, в которой заданы значения первого элемента m и второго k. Первое лямбда-выражение задает преобразование от пары предыдущих элементов к текущему,

ложное значение второго определяет завершение генерации последовательности. Генератор используется, когда имеется формула, связывающая функциональной зависимостью текущий член последовательности с двумя предшествующими.

5.1.1.3 Генераторы бесконечных последовательностей

Бесконечные последовательности также можно задавать при помощи генераторов, поскольку генератор сам по себе лишь описывает способ получения (перебора) элементов последовательности. Элемент последовательности создается в момент обращения к нему, например, при выводе. При использовании бесконечной последовательности надо ограничить ее длину во избежание зацикливания программы. Эти генераторы реализованы в виде расширений.

- Любую последовательность s можно зациклить при помощи генератора s.Cycle, бесконечно ее повторяющего.
- Бесконечную последовательность целых чисел вида m, m+1, m+2, … можно получить посредством генератора m.Step.
- Бесконечную последовательность чисел типа T, начиная от значения m с шагом h можно получить при помощи генератора m.Step(h).

Например, мы могли бы получить значения аргумента для задачи табуляции функции, рассмотренной в конце части 3, с помощью генератора (8 / 3).Step(1 / 3).

Ограничить количество членов бесконечной последовательности можно разными способами. Так, расширение .Take(n), позволяет взять n первых членов последовательности. Оператор Seq(1, 6, 3, -2).Cycle.Take(12) создаст последовательность типа **integer** длиной 12 элементов, в которой циклически повторяется исходная последовательность 1, 6, 3, -2. В этой записи последовательность типа **integer**, созданная генератором Seq, расширяется с помощью .Cycle, а длина полученной бесконечной последовательности ограничивается расширением .Take.

5.1.1.4 Ввод элементов последовательности с клавиатуры

Достаточно часто в задачах встречаются формулировки вида «Дана последовательность из n чисел, вводимых с клавиатуры». Или, «С клавиатуры вводится последовательность чисел, ограниченная нулём». Традиционно для таких задач организуется цикл со вводом значений в некоторую переменную. Неопытные программисты исхитряются организовывать ввод данных в массив, создавая неэкономное как по памяти, так и по времени решение. Справедливости ради отметим, что для некоторых задач другого пути в базовом Паскале просто нет. Работая с такими последовательностями помните, что они являются в определенной степени недетерминированными, поскольку при повторном просмотре потребуется повторный ввод, а в процессе ввода вы можете сделать ошибку. Да и просто глупо вводить дважды одно и то же.

Пусть дана последовательность из n целых чисел (значение n вводится с клавиатуры) и требуется подсчитать сумму нечетных чисел в ней. Рассмотрим «традиционное» решение средствами PascalABC.NET.

```
begin
  var n := ReadInteger('n=');
  var s := 0;
  for var i := 1 to n do
  begin
    var d := ReadInteger;
    if Odd(d) then s += d
  end;
  s.Println
end.
```

Ниже показан пример работы программы.

```
n= 6
18 -13 40 0 51 14
38
```

Можно ли эту задачу решить, используя последовательности – ведь в задании говорилось именно о них? Да, если мы научимся формировать последовательности путем ввода значений с клавиатуры.

PascalABC.NET предлагает обширный набор средств для формирования последовательностей на основе ввода значений их элементов с клавиатуры.

- `ReadSeqInteger(n)` – ввод последовательности n элементов типа **integer**;
- `ReadSeqInteger('текст приглашения',n)` – ввод с приглашением последовательности n элементов типа **integer**;
- `ReadSeqIntegerWhile(integer -> boolean)` – ввод последовательности элементов типа **integer** до тех пор, пока лямбда-выражение истинно;
- `ReadSeqIntegerWhile('текст приглашения', integer -> boolean)` – ввод с приглашением последовательности элементов типа **integer** до тех пор, пока лямбда-выражение истинно;
- `ReadSeqReal(n)` – ввод последовательности n элементов типа **real**;
- `ReadSeqReal('текст приглашения',n)` – ввод с приглашением последовательности n элементов типа **real**;
- `ReadSeqRealWhile(real -> boolean)` – ввод последовательности элементов типа **real** до тех пор, пока лямбда-выражение истинно;
- `ReadSeqRealWhile('текст приглашения', real -> boolean)` – ввод с приглашением последовательности элементов типа **real** до тех пор, пока лямбда-выражение истинно.

При написании программ с последовательностями нужно учитывать, что работа с элементами последовательности организуется на основе так называемых ***отложенных (ленивых) вычислений***. Обращение к элементам последовательности происходит только в тот момент, когда их значение требуется для продолжения работы. С учебной целью включим в код дополнительный вывод, информирующий о выполнении операторов программы.

```
begin // p05004
  var n := ReadInteger('n=');
  Println('Запрашиваем ввод данных');
  var p := ReadSeqInteger('Вводите данные:', n);
  Println('Закончили ввод');
  var s := 0;
  Println('Начали цикл');
  foreach var d in p do // про foreach чуть позже
    if Odd(d) then s += d;
  Println('Закончили цикл');
  s.Println
end.
```

Просмотрите приведенный протокол работы программы, прежде чем ее запускать самостоятельно.

```
n= 6
Запрашиваем ввод данных
Вводите данные: Закончили ввод
Начали цикл
18 -13 40 0 51 14
Закончили цикл
38
```

Программа запросила ввод данных не в момент исполнения ReadSeqInteger (в тот момент она лишь вывела приглашение по вводу - забавно, не так ли?), а только когда начался цикл перебора элементов последовательности **foreach**. Это и есть демонстрация проявления отложенных вычислений. А также демонстрация того, что бывает, когда программы пишут недостаточно компетентные люди.

В следующем примере программа ведет себя еще хуже. Давайте после ввода сделаем контрольный вывод, а затем будем считать сумму.

```
begin // p05005
  var n := ReadInteger('n=');
  Println('Запрашиваем ввод данных');
  var p := ReadSeqInteger('Вводите данные:', n);
  Println('Закончили ввод, делаем контрольный вывод');
  p.Println;
  var s := 0;
  Println('Начали цикл');
  foreach var d in p do // про foreach чуть позже
    if Odd(d) then s += d;
  Println('Закончили цикл');
  s.Println
end.
```

Ниже дан протокол работы этой программы (обязательно попробуйте ее запустить):

```
n= 6 // вводим 6
Запрашиваем ввод данных
Вводите данные: Закончили ввод, делаем контрольный вывод
18 -13 40 0 51 14 // это тоже вводим
18 -13 40 0 51 14 // контрольный вывод
Начали цикл
18 -13 40 0 51 14 // недетерминированная последовательность! Повторный ввод.
Закончили цикл
38
```

Конечно, это все занимательно, но как же написать программу нормально? Посмотрите код, который вы вскоре научитесь писать и сравните его с предыдущим. Однострочное решение, ведь перенос сделан потому, что строка слишком длинная и тут не поместилась.

```
begin // p05006
  ReadSeqInteger('Вводите данные:', ReadInteger('n='))
      .Where(t -> t.IsOdd).Sum.Println
end.
```

5.1.1.5 Самостоятельное создание последовательностей (yield)

Несмотря на обилие средств создания последовательностей, рано или поздно встретится случай, когда ни одно из них не подойдет. Рассмотрим случай, когда требуется получить последовательность из n случайных целых чисел в диапазоне [13; 57], вывести значения ее элементов и найти их среднее арифметическое. Использовать SeqRandom не получится: будет сгенерирована недетерминированная последовательность. После вывода значений элементов этой последовательности потребуется второй проход для вычисления среднего арифметического, в результате чего элементы получат другие значения.

Вначале рассмотрим решение без использования последовательности.

```
begin
  var n := ReadInteger('n=');
  var s := 0;
  loop n do
  begin
    var m := Random(13, 57); // очередное случайное число
    m.Print;
    s += m; // накопление суммы
  end;
  Writeln(NewLine, s / n)
end.
n= 10 // для примера
54 56 23 39 25 39 48 23 24 22 // сгенерированные значения
35.3 // среднее арифметическое
```

В приведенном решении понадобился один цикл, следовательно, использованный алгоритм однопроходный. Теперь попробуем создавать элементы последовательности и обрабатывать их, перебирая при помощи единственного цикла **foreach**.

```
begin
  var n := ReadInteger('n=');
  var (s, p) := (0, SeqRandom(n, 13, 57));
  foreach var m in p do
  begin
    m.Print;
    s += m;
  end;
  Writeln(NewLine, s / n)
end.
```

Короче программа не стала. Стала ли она нагляднее? Если и стала, то незначительно. Следующим будет вариант с расширением .ForEach.

```
begin
  var n := ReadInteger('n=');
  var s := 0;
  SeqRandom(n, 13, 57).ForEach(t -> begin t.Print; s += t end);
  Writeln(NewLine, s / n)
end.
```

Это выглядит уже интереснее. Код стал короче вдвое. Но только мы пока так и не создали собственной последовательности.

Как бы вы ни старались обрабатывать недетерминированную последовательность, она будет менять значения элементов при повторном просмотре. Следовательно, будет неплохой идеей объединить генерацию последовательности с выводом значений ее элементов. И тогда можно найти среднее арифметическое посредством расширения .Average. Вот пример такой реализации программы.

```
function SeqRandomPrintln(n, a, b: integer): sequence of integer;
begin
  loop n do
  begin
    var m := Random(a, b);
    m.Print;
    yield m
  end;
  Println
end;

begin
  var n := ReadInteger('n=');
  SeqRandomPrintln(n, 13, 57).Average.Println
end.
```

В основной программе все выглядит коротко и понятно. Но появилась пользовательская функция SeqRandomPrintln, возвращающая последовательность. В ее теле вам впервые встретился оператор **yield**. Выше было сказано, что при работе с

последовательностями используются ленивые вычисления. Как только расширение .Average будет готово обработать очередной элемент последовательности, произойдет очередное обращение к функции SeqRandomPrintln, в ее теле произойдет следующий проход по циклу **loop** и оператор **yield** вернет на место вызова значение очередного элемента. Затем выполнение функции прервется до следующего запроса.

Последним вариантом решения этой же задачи будет программа с использованием расширения, осуществляющего вывод последовательности и передающая ее транзитом на выход. Благодаря такому подходу мы не даем последовательности проявить недетерминированность, а расширение .SeqPrintln становится универсальным. Здесь вы также познакомитесь с созданием собственных расширений.

```
function SeqPrintln(Self: sequence of integer): sequence of integer;
    extensionmethod; // p05007
begin
  foreach var m in Self do
  begin
    m.Print;
    yield m
  end;
  Println
end;

begin
  SeqRandom(ReadInteger('n='), 13, 57).SeqPrintln.Average.Println
end.
```

В заголовке функции использовано ключевое слово **extensionmethod**, которое сообщает, что функция должна вызываться подобно расширению, т.е. встраиваться в цепочку вызовов посредством точечной нотации, получая в качестве параметра **Self** результат, возвращенный предшествующей функцией. Подробнее такие функции будут описаны при рассмотрении ООП.

Необходимо четко представлять, что «ленивость» может сослужить плохую службу, если ей неправильно пользоваться. В качестве домашнего задания попробуйте ответить, почему приведенная ниже программа дает такой странный результат:

```
1 12  23  34  45  5
```

Думаете, это компилятор неверно работает? Нет, компилятор тут совершенно не причем. Это пример типичной программистской «криворукости», идущей от плохого знания материала. Еще одна иллюстрация того, что главное – изучать алгоритмы и их работу, а язык программирования осваивать лишь потом.

```
function SeqPrintln(Self: sequence of integer): sequence of integer;
    extensionmethod; // p05008
begin
  foreach var m in Self do
  begin
    m.Print;
    yield m
  end;
  Println
end;

begin
  var s := Seq(1, 2, 3, 4, 5);
  s.SeqPrintln.SeqPrintln;
  s.SeqPrintln.Println
end.
```

5.1.2 Операции с последовательностью в целом

В PascalABC.NET имеется очень большое количество операций, которые можно выполнять с последовательностями. Сюда относятся различного рода выборки элементов подпоследовательностей, подсчет значений, таких как сумма, на основе всех или выбранных элементов, преобразование элементов последовательности, разбиения и слияния последовательностей и многое другое.

Последовательность доступна только на чтение; после того, как последовательность создана, ее нельзя изменить. Но можно сформировать из нее новую последовательность и присвоить ее прежнему имени.

5.1.2.1 Пустые последовательности и метод Count

В результате выборки элементов из последовательности может получиться пустая последовательность, т.е. последовательность, у которой количество элементов равно нулю. Например, пустой будет последовательность в результате выборки элементов с отрицательными значениями из последовательности, содержащей только неотрицательные элементы. Количество элементов в последовательности s (длину последовательности) можно получить при помощи вызова метода s.Count.

*5.1.2.2 Операции с последовательностью + и **

Эти операции позволяют соединять между собой последовательности некоторого типа T или складывать последовательности с выражением типа T, присоединяя его значение к последовательности. Также можно умножать последовательность на целочисленное значение, сцепляя ее с собой несколько раз, как это делает расширение .Cycle.

```
begin
  var s1 := Seq(3, -5, 2, 0, 18);
  var s2 := Seq(4, 8, 0, 11);
  var s := s1 + s2; // соединение последовательностей
  s.Println; // 3 -5 2 0 18 4 8 0 11
  s := 100 + s + 200; // присоединение элементов
  s.Println; // 100 3 -5 2 0 18 4 8 0 11 200
  s := s1 * 3; // умножение на число (аналог .Cycle)
  s.Println // 3 -5 2 0 18 3 -5 2 0 18 3 -5 2 0 18
end.
```

5.1.2.3 Перебор элементов последовательности (цикл foreach)

Для перебора элементов последовательности служит цикл **foreach**. Он имеет следующий формат:

```
foreach var ИмяПеременной in Последовательность do
    оператор или блок;
```

Последовательность может задаваться как непосредственно, так и именем. В процессе выполнения цикла переменная *ИмяПеременной* принимает значение каждого элемента последовательности, начиная от первого и заканчивая последним. В качестве примера приведена программа, подсчитывающая сумму нечетных элементов случайной последовательности целых чисел. Вспомните, что метод c.IsOdd возвращает True для нечетного c и False для четного.

```
begin // p05009
  var (s, R) := (SeqRandom(ReadInteger('n='), -20, 20), 0);
  foreach var c in s do
  begin
    c.Print;
    if c.IsOdd then R += c
  end;
  Println(NewLine, R)
end.
```

5.1.2.4 Перебор элементов последовательности (.ForEach)

Функциональная парадигма предоставляет еще один способ перебора элементов. Расширение последовательности s.ForEach(T -> ()) позволяет задать лямбда-процедуру, которая применяется к каждому элементу последовательности. Запишем с помощью ForEach рассмотренную выше задачу:

```
begin // p05010
  var s := SeqRandom(ReadInteger('n='), -20, 20);
  var R := 0;
  s.ForEach(d -> begin Print(d); if d.IsOdd then R += d end);
  Println(NewLine, R)
end.
```

5.1.2.5 Проецирование (метод Select)

Проецирование – понятие из функционального программирования. В PascalABC.NET оно базируется на технологии LINQ to Objects библиотек .NET Framework. Суть проецирования состоит в поочередном применении некоторой функ-

ции, возвращающей значение типа T, к каждому элементу исходной последовательности. В результате получается новая последовательность, состоящая из элементов, значения которых вернула функция. Сама функция задается лямбда-выражением.

В качестве примера рассмотрим вывод последовательности из двенадцати кубов целых чисел, случайно заданных на интервале [-10; 20].

```
begin
  SeqRandom(12, -10, 20).Select(i -> i * i * i).Println
end.
```

Пример работы программы:

```
-216 -27 1331 -216 6859 3375 -125 1000 -729 -512 216 125
```

5.1.2.6 Проецирование (метод SelectMany)

В отличие от описанного выше метода Select, к результату повторно применяется проецирование. Следовательно, если после первого проецирования получаются несколько последовательностей, они затем будут объединены в одну.

Пусть дана последовательность, состоящая из нескольких строк. Если применить к ней Select, результатом будет также последовательность такого же количества строк. Метод SelectMany превратит каждую строку в последовательность символов, а затем объединит эти последовательности в одну, состоящую из односимвольных строк (не сделайте ошибку, посчитав, что получится последовательность символов sequence of char).

```
begin
  var s1 := Seq('один', 'два', 'три','четыре','пять');
  Println(s1); // [один,два,три,четыре,пять]
  var s2 := s1.Select(t -> t);
  Println(s2); // [один,два,три,четыре,пять]
  var s3 := s1.SelectMany(t -> t);
  Println(s3); // [о,д,и,н,д,в,а,т,р,и,ч,е,т,ы,р,е,п,я,т,ь]
end.
```

Еще один пример. Пусть последовательность состоит из одной строки, содержащей несколько слов, разделенных пробелами. Метод .ToWords, примененный к подобной строке, превращает ее в массив слов (см. 6.2.15).

```
begin
  var s:=Seq('  один   два  три     четыре пять  ');
  Writeln(s.Select(t-> t.ToWords)); // [[один,два,три,четыре,пять]]
  Writeln(s.SelectMany(t-> t.ToWords)); // [один,два,три,четыре,пять]
end.
```

Видно, что проецирование с помощью Select порождает последовательность, состоящую из последовательности слов. SelectMany дает возможность получить обыкновенную последовательность слов.

Метод SelectMany имеет несколько разновидностей. Рассмотрим две из них.

- s.SelectMany(ss -> ss1) – проецирует каждый элемент ss последовательности s в новую последовательность ss1 и возвращает результирующие последовательности, объединенные в одну общую последовательность;
- s.SelectMany((ss, i) -> ss1) – проецирует каждый элемент ss последовательности s в новую последовательность ss1 с учетом индекса элемента i, возвращая результирующие последовательности, объединенные в одну общую последовательность. Под индексом элемента подразумевается порядковый номер элемента в последовательности. Считается, что нумерация производится от нуля.

Ниже приведен достаточно надуманный пример использования SelectMany.

```
begin
  var s := Seq('один', 'два', 'три', 'четыре', 'пять');
  s.SelectMany((t, i)->
      t.ToWords.Select(t-> i + 1 + '. ' + t)).PrintLines;
end.
```

```
1. один
2. два
3. три
4. четыре
5. пять
```

Конечно, такого результата можно было добиться и менее экзотическим способом.

5.1.2.7 Максимум, минимум, сумма, произведение и среднее

Для последовательности s в языке имеются методы, позволяющие получать максимальное s.Max и минимальное s.Min значения элементов этой последовательности. Существуют также расширения s.MaxBy и s.MinBy, позволяющие указать лямбда-выражение, задающее предварительное преобразование. Как будет показано далее, можно применять эти методы и расширения также и для объектов с сложной структурой, например, кортежей.

Пусть имеется некоторая последовательность целых чисел, в которой нужно найти значение элемента, минимальное по абсолютной величине. Рассмотрим два способа решения этой задачи. Здесь мы можем себе позволить просматривать последовательность s несколько раз, поскольку она является детерминированной.

```
begin
  var s := Seq(7, 9, -6, 3, 5, -1, 4, 3, -9);
  s.Println;
  s.MinBy(p -> Abs(p)).Println; // решение 1, ниже - второе решение
  var (Min, Value) := (integer.MaxValue, 0);
  foreach var p in s do
    if Abs(p) < Min then (Min, Value) := (Abs(p), p);
  Value.Println;
end.
```

Результаты, которые дают оба способа, конечно же, одинаковы.

```
7 9 -6 3 5 -1 4 3 -9
-1
-1
```

Если минимумов или максимумов несколько, будет найден первый из них. А если нужен не первый? Имеются расширения s.LastMaxBy и s.LastMinBy, возвращающие последний из найденных элементов. Когда и этого недостаточно, приходится прибегать к более сложным способам решения.

Сумма значений всех элементов последовательности s может быть получена при помощи метода s.Sum, произведение – при помощи метода s.Product, а метод s.Average вернет среднее арифметическое. Если нужно найти сумму, произведение или среднее арифметическое значений не самих элементов последовательности, а некоторой функции от них, при вызове расширения можно указать соответствующее лямбда-выражение. Обратите внимание, что метод .Product возвращает значение типа real.

В приведенном ниже примере вначале находится сумма квадратов пяти введенных с клавиатуры членов последовательности, а затем произведение четырех введенных с клавиатуры членов другой последовательности.

```
begin
  ReadSeqInteger('Введите 5 целых чисел:', 5).Sum(k -> k * k).Println;
  ReadSeqInteger('Введите 4 целых числа:', 4).Product.Println
end.
```

Пример работы программы:

```
Введите: 6 -2 4 8 -5
145
Введите 4 целых числа: 13 -23 18 121
-651222
```

Отметим, что у метода Average есть особенность, способная доставить неприятность. Чему равна средняя величина значений пустой последовательности? А ничему и уж никак не нулю. Если сумма тут корректна, поскольку ее вычисление начинается с обнуления будущего результата и при пустой последовательности ноль остается, то со средним арифметическим все сложнее. Оно должно вычисляться, как сумма значений, деленная на их количество. Сумма ноль, но и количество – ноль! Что получим при делении? Ничего хорошего. Посмотрим, что будет в таком случае делать программа.

```
begin
  0.Range.Average.Println;
end.
```

Программа создает пустую последовательность целочисленных элементов и применяет к ней метод Average. Он не настолько глуп, чтобы заниматься какими-то подсчетами там, где считать нечего. Поэтому мы получим сообщение:

```
«Ошибка времени выполнения: Последовательность не содержит элементов»
```

Если это и вся наша программа, - ничего страшного. А если это фрагмент где-то в середине? На такой случай нужно предусматривать дополнительные «меры безопасности». Но об этом позже.

5.1.2.8 Агрегирование элементов (метод Aggregate)

Метод Aggregate((T, T) -> T) возвращает результат применения к элементам последовательности функции, заданной лямбда-выражением. **Агрегатная функция** (функция, производящая накопление) вначале применяется к первому и второму элементам последовательности, затем к возвращенному ей результату и третьему элементу, далее – к результату и четвертому элементу и так далее.

```
begin
  Seq(6, -3, 11, 8, -2).Println.Aggregate((s, j)-> s + j).Println
end.
```

Программа выведет следующий результат:

```
6 -3 11 8 -2
20
```

Первоначально в качестве s берется ноль, в качестве j – значение первого элемента. В соответствии с лямбда-выражением значения суммируются и получается число 6, которое теперь принимается за s. В качестве j берется очередной элемент последовательности, равный -3. Далее действия повторяются и конечный результат будет равен сумме всех элементов последовательности.

Метод s.Aggregate(a, (T, T) -> T) возвращает результат применения к элементам последовательности агрегатной функции, заданной лямбда-выражением, принимая значение a в качестве начального. На первом шаге функция применяется к начальному значению и первому элементу последовательности. На втором шаге аргументами функции будут возвращенный результат и второй элемент последовательности и так далее.

```
begin
  Seq(6, -3, 11, 8, -2).Println.Aggregate(1,(s, j) -> s * j).Println
end.
```

Программа выведет следующее:

```
6 -3 11 8 -2
3168
```

Вычисляется произведение элементов последовательности, поэтому в качестве начального значения задана единица. Тип результата определяется типом начального значения, поэтому в случае умножения целых величин остерегайтесь переполнения.

Давай теперь выберем начальное значение типа **BigInteger**.

```
begin
  50.Range.Aggregate(BigInteger(1),(s, i) -> s * i).Println
end.
```

Это отличный способ вычислить 50! как произведение членов последовательности из 50 первых натуральных чисел. В результате получим значение

30414093201713378043612608166064768844377641568960512000000000000

Агрегирование – удобный способ получить произведение элементов последовательности, не прибегая к циклу.

5.1.2.9 Перестановка элементов в обратном порядке (метод Reverse)

Метод s.Reverse возвращает последовательность, элементы которой расположены в обратном порядке. Вы знаете, как за один просмотр последовательности сделать такую перестановку? Никак. Поэтому метод поступает «нечестно»: он запоминает все элементы последовательности, а потом возвращает их, просматривая в порядке от последнего к первому. Но ведь мы всегда можем сделать вид, что об этом даже не подозревали!

```
begin
  Seq(7, 9, -6, 3, 5, -1, 4, 3, -9).Println.Reverse.Println
end.
```

```
7 9 -6 3 5 -1 4 3 -9
-9 3 4 -1 5 3 -6 9 7
```

5.1.3 Выборка подпоследовательности

Из последовательности s некоторого типа T можно выделить некоторую подпоследовательность, содержащую элементы, идущие подряд. Для этого используются большой набор методов и расширений.

5.1.3.1 Выборка элементов от начала последовательности

- s.First – первый элемент;
- s.FirstOrDefault – первый элемент; если последовательность пустая - значение по умолчанию;
- s.First(T -> **boolean**) – первый элемент, для которого лямбда-выражение истинно;
- s.FirstOrDefault(T -> **boolean**) – первый элемент, для которого лямбда-выражение истинно. Если такой элемент не найден, значение по умолчанию;
- s.Take(n) – последовательность из первых n элементов;
- s.TakeWhile(T -> **boolean**) - – последовательность от начала и до тех пор, пока для каждого элемента лямбда-выражение истинно.

5.1.3.2 Пропуск части первых элементов

- s.Skip(n) – последовательность после пропуска n первых элементов;

s.SkipWhile(T -> **boolean**) – последовательность после пропуска первых элементов, для которых лямбда-выражение истинно.

5.1.3.3 Пропуск части последних элементов

- s.SkipLast – последовательность без последнего элемента;
- s.SkipLast(n) – последовательность без n последних элементов.

5.1.3.4 Выборка элементов от конца последовательности

- s.Last – последний элемент;
- s.LastOrDefault – последний элемент; если последовательность пустая - значение по умолчанию;
- s.Last(T -> **boolean**) – последний элемент, для которого лямбда-выражение истинно;
- s.LastOrDefault(T -> **boolean**) – последний элемент, для которого лямбда-выражение истинно. Если такой элемент не найден, значение по умолчанию.

Пример 1. Из заданной последовательности 15 целых чисел получить новую последовательность, взяв 6 ее первых элементов и 4 последних.

```
begin
  var s := Seq(27, 15, 2, -10, 14, 11, -3, 19, 28, -1, -18, 13, 2, -8, 16);
  s.Println;
  s := s.Take(6) + s.TakeLast(4);
  s.Println
end.
```

Пример 2. Дана последовательность, каждый элемент которой получается по формуле $a_i=4i^2-12i-8$, i=1,2, … n. Получить из нее новую последовательность длины 10, пропустив отрицательные элементы. Возьмем первых, например, 15 элементов:

```
begin
  var s := SeqGen(15, i -> 4 * i * i - 12 * i - 8)
      .SkipWhile(q -> q < 0).Take(10).Println
end.
```

Конечно, здесь оператор присваивания не нужен и можно было начать прямо с SeqGen, но поскольку в задании сказано «получить», а не просто вывести последовательность, она будет присвоена переменной s.

5.1.4 Кортежи

Результатами некоторых выборок могут быть кортежи последовательностей и последовательности кортежей.

Кортеж – это последовательность элементов, доступная только на чтение и содержащая от двух до семи элементов. Каждый элемент кортежа нумеруется; нумерация ведется от нуля. Для доступа к элементу кортежа указывается его имя, за которым в квадратных скобках следует номер элемента, например, c[3]. Есть и альтернативная запись c.ItemN, где N – номер элемента с нумерацией уже от единицы. Почему так? Так сложилось, ничего тут не поделать, поэтому c[3] и c.Item4 – одно и то же.

Для кортежей в PascalABC.NET имеется общий тип **tuple**. Элементы кортежа могут иметь различный тип, в том числе, элемент может в свою очередь быть кортежем. Тип конкретного кортежа может оказаться достаточно сложным, поэтому лучше пользоваться автовыведением типа.

```
var c := (1, 2.5, Seq(-2, 6, 0));
```

В приведенном примере создается кортеж c. Его первый элемент c[0] имеет тип integer, второй c[1] – тип real, третий c[2] – тип sequence of integer. А какой же тогда тип у переменной c? Давайте это выясним.

```
begin
  var c := (1, 2.5, Seq(-2, 6, 0));
  c.Println; // вывод кортежа
  Println(c.GetType) // вывод типа этого кортежа
end.
```

Программа выведет две строки:

```
(1,2.5,[-2,6,0])
System.Tuple`3[System.Int32,System.Double,System.Collections.Generic.IEnumerable`1[System.Int32]]
```

Конечно, если вы можете точно указать подобный тип для любого кортежа, вам можно дальше эту книгу не читать.

Последовательности, как уже не раз было сказано, не хранятся, а кортеж хранится. Противоречия тут нет, потому что в кортеже хранится не сама последовательность, а лишь ссылка на нее. Отправив в кортеж недетерминированную подпоследовательность, приготовьтесь к неприятным сюрпризам. Даже невинная с виду программа из одной строки может потребовать дважды вводить данные.

```
begin // p05011
  ReadSeqIntegerWhile(t -> t <> 0).SplitAt(3).Println;
end.
```

Посмотрите на приведенный ниже протокол ее работы и попробуйте самостоятельно выполнить эту программу со своими данными.

```
3 6 -4 6 7 -2 1 9 0
3 6 -4 6 7 -2 1 9 0
([3,6,-4],[6,7,-2,1,9])
```

Расширение SplitAt(3) разрезает последовательность на две подпоследовательности так, что в первой находятся ровно три элемента, а затем помещает их в кортеж. Реализация этого расширения использует двухпроходный алгоритм, поэтому его имеет смысл применять только с детерминированными последовательностями.

При необходимости кортеж можно распаковать в отдельные переменные с помощью кортежного присваивания. Мы делали это не раз, описывая несколько переменных с их одновременной инициализацией.

В качестве примера использования кортежей рассмотрим обмен местами значений двух переменных. Традиционное решение состоит в цепочке присваиваний через промежуточную переменную.

```
begin
  var (a, b) := (5, 18);
  Println(a, b); // 5 18
  var t := a;
  a := b;
  b := t;
  Println(a, b) // 18 5
end.
```

Если переменные имеют числовое значение, возможен «арифметический фокус», чтобы не использовать промежуточной переменной.

```
begin
  var (a, b) := (5, 18);
  Println(a, b); // 5 18
  b := a + b;
  a := b - a;
  b := b - a;
  Println(a, b) // 18 5
end.
```

Еще один «фокус» – использование логической операции **xor** – исключающего «ИЛИ». Он применим только к типу **boolean** и целочисленным типам за исключением **BigInteger**.

```
begin
  var (a, b) := (5, 18);
  Println(a, b); // 5 18
  a := a xor b;
  b := a xor b;
  a := a xor b;
  Println(a, b)// 18 5
end.
```

Кортежи дают короткое и универсальное решение этой задачи:

```
begin
  var (a, b) := (5, 18);
  Println(a, b); // 5 18
  (a, b) := (b, a);
  Println(a, b)// 18 5
end.
```

И, наконец, существует процедура Swap(a, b), которая также совершает обмен значениями двух переменных, работая несколько быстрее кортежного обмена.

5.1.5 Выборка на основе условия

В состав PascalABC.NET включены методы и расширения, позволяющие производить выборку элементов последовательности, удовлетворяющую условиям любой сложности.

5.1.5.1 Выборка неповторяющихся элементов (.Distinct)

Расширение s.Distinct выбирает из последовательности неповторяющиеся элементы.

Пример. На основе последовательности из 15 целых чисел получить новую последовательность, все элементы которой различны.

```
begin
  var s := Seq(5, 1, -3, 1, 1, 6, 5, 7, 2, 1, -3, 9, 8, -3, 1);
  s.Println;
  s := s.Distinct;
  s.Println
end.
```

В результате получаем следующий вывод:

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

5.1.5.2 Фильтрация (метод Where)

Фильтрация – еще одно понятие из функционального программирования. Как и проецирование, она также базируется на LINQ to Objects библиотек Microsoft .NET Framework. Суть фильтрации состоит в последовательном применении логической функции, задаваемой лямбда-выражением, к каждому элементу исходной последовательности. В результате получается новая последовательность, состоящая из элементов, для которых функция вернула True. В отличие от проецирования, которое порождает выходную последовательность той же длины, что и исходная последовательность, фильтрация, как правило, уменьшает количество элементов. Сочетание фильтрации и проецирования позволяет получать мощные выборки и выполнять сложные преобразования.

Пусть дана последовательность из десяти элементов. Требуется вывести элементы этой последовательности с нечетными значениями.

```
begin // p05012
  var s := Seq(-13, -41, -48, 31, 31, -41, 38, 26, 8, 7);
  s.Println.Where(t -> t.IsOdd).Println
end.
```

Ниже показан пример работы программы:

```
-13 -41 -48 31 31 -41 38 26 8 7
-13 -41 31 31 -41 7
```

5.1.5.3 Последовательность на основе пар элементов

Расширение s.Incremental формирует из последовательности s новую последовательность, каждый член которой равен разности двух соседних элементов s – последующего и предыдущего.

```
begin
  Seq(7, 9, 6, 3, 5, 1, 4, 3, 9).Println.Incremental.Println
end.
```

Создается последовательность из указанных значений, выводится на монитор, затем из нее формируется новая последовательность, которая также выводится:

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

Расширение s.Incremental((T, T) -> T) формирует из последовательности s новую последовательность, каждый член которой равен некоторой функции от двух соседних элементов. В частности, задавая s.Incremental((i, j) -> j – i), получим тот же результат, что и с использованием s.Incremental.

```
begin
  Seq(7, 9, 6, 3, 5, 1, 4, 3, 9).Println.Incremental((i, j)-> 2 * i + j).Println
end.
```

Будут выведены значения

```
7 9 6 3 5 1 4 3 9
23 24 15 11 11 6 11 15
```

Расширение s.Pairwise формирует из последовательности s новую последовательность, каждый член которой представляет собой кортеж из пары соседних элементов

```
begin
  Seq(7, 9, 6, 3, 5, 1, 4, 3, 9).Println.Pairwise.Println
end.
```

Пример выводит следующие значения

```
7 9 6 3 5 1 4 3 9
(7,9) (9,6) (6,3) (3,5) (5,1) (1,4) (4,3) (3,9)
```

Расширение s.Pairwise((T, T) -> T) формирует из последовательности s новую последовательность, применяя функцию к каждой паре соседних элементов. Это полный аналог расширения s.Incremental((T, T) -> T).

5.1.6 Нумерация элементов последовательности

Расширение s.Numerate формирует из последовательности s новую последовательность двухэлементных кортежей вида (n, a), где n = 1, 2, 3, ... - порядковый номер элемента в последовательности при нумерации от единицы, a – элемент последовательности.

```
begin
  Seq(7, 9, 6, 3, 5, 1, 4, 3, 9).Println.Numerate.Println
end.
```

Получаем следующий вывод:

```
7 9 6 3 5 1 4 3 9
(1,7) (2,9) (3,6) (4,3) (5,5) (6,1) (7,4) (8,3) (9,9)
```

Расширение s.Numerate(m) формирует из последовательности s новую последовательность двухэлементных кортежей вида (n, a), где n = m, m+1, m+2, ... - порядковый номер элемента в последовательности при нумерации от m, a – элемент последовательности.

5.1.7 Разбиение последовательности

Последовательность может быть «разрезана» на подпоследовательности равной длины; при этом допускается, чтобы последняя из подпоследовательностей оказалась короче остальных. Результатом такой операции будет последовательность, каждый элемент которой является последовательностью, т.е. «последовательность последовательностей» с типом **sequence of sequence of** ТипЭлемента. С ней можно работать, как с обыкновенной последовательностью.

5.1.7.1 Разбиение на части указанной длины (.Batch)

Разделение последовательности s осуществляет расширение s.Batch(k), где k – требуемая длина каждой из получаемых подпоследовательностей.

В приведенном примере последовательность разделяется на серию подпоследовательностей длиной 4 с именем p. Далее, каждый элемент p выводится в цикле **foreach**.

```
begin // p05013
  var s := Seq(7, 9, 6, 3, 5, 1, 4, 3, 9);
  s.Println;
  var p := s.Batch(4).Println;
  foreach var r in p do
    r.Println;
end.
```

Результат работы программы:

```
7 9 6 3 5 1 4 3 9
[7,9,6,3] [5,1,4,3] [9]
7 9 6 3
5 1 4 3
9
```

5.1.7.2 Разбиение на две части по условию (.Partition)

Последовательность s может быть также разделена на две по некоторому условию, заданному лямбда-выражением в функции типа T -> boolean. При этом в первую подпоследовательность попадут элементы, для которых заданное выражение

истинно, а во вторую – для которых оно ложно. Запись такого расширения имеет вид s.Partition(T -> boolean). Реализация использует двухпроходный алгоритм, поэтому .Partition имеет смысл применять **только с детерминированными последовательностями**. Результатом работы будет кортеж из двух подпоследовательностей. Ниже показан пример распаковки кортежа в две последовательности.

```
begin // p05014
  var s := Seq(7, 9, 6, 3, 5, 1, 4, 3, 9);
  s.Println;
  var (Чет, Нечет) := s.Partition(v -> v.IsEven);
  Чет.Println;
  Нечет.Println
end.
```

Расширение целочисленного типа v.IsEven возвращает True, если v четное. Последовательность `Чет` будет содержать четные элементы, а последовательность `Нечет` – нечетные.

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

5.1.7.3 Разбиение на две части по длине (.SplitAt)

Расширение s.SplitAt(n) разбивает последовательность на две подпоследовательности так, что первая из подпоследовательностей содержит ровно n элементов. Возвращается кортеж из полученных подпоследовательностей. Реализация использует двухпроходный алгоритм, поэтому имеет смысл применять его только с детерминированными последовательностями. Пример приведен выше (см. 5.1.4).

5.1.7.4 Срез последовательности (.Slice)

Расширение вида s.Slice(a, h) формирует из последовательности s ***срез*** (см. 5.3.5) – новую последовательность, полученную выбором элементов, начиная с номера a и с шагом h

```
begin
  Seq(7, 9, 6, 3, 5, 1, 4, 3, 9).Println.Slice(1, 3).Println
end.
```

Здесь выбирается каждый третий элемент, начиная с первого, причем отсчет ведется от нуля.

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

Расширение вида s.Slice(a, h, n) формирует из последовательности срез, начиная с номера a, с шагом h и длиной не более n элементов.

```
begin
  Seq(7, 9, 6, 3, 5, 1, 4, 3, 9).Println.Slice(0, 2, 4).Println
end.
```

Будет выбрано не более четырех первых элементов с четными номерами

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

5.1.8 Операции с несколькими последовательностями

Результатом операции над несколькими последовательностями является последовательность элементов или кортежей. В зависимости от операции, из исходных последовательностей могут извлекаться как все элементы, так и лишь уникальные.

5.1.8.1 Объединение, пересечение и разность

Если рассмотреть совокупность элементов последовательности как множество, для двух последовательностей можно получить результат их объединения, пересечения или разности. Фактически множество – это разновидность последовательности, в которой нет элементов с одинаковым значением. Для последовательности s такой эффект дает s.Distinct.

Метод p.Union(q) объединяет последовательности p и q, создавая новую последовательность, содержащую все элементы p и q без повторения их значений. В отличие от работы с множеством, в котором порядок следования элементов несущественен, здесь элементы полученной последовательности следуют в том же порядке, в котором они находились в исходных последовательностях.

```
begin
  var p := Seq(7, 9, 6, 3, 5, 1, 4, 3, 9);
  var q := Seq(8, 0, 3, 6, 2, 4, 7, 2);
  p.Println;
  q.Println;
  p.Union(q).Println
end.
```

Будет получен следующий результат:

```
7 9 6 3 5 1 4 3 9
8 0 3 6 2 4 7 2
7 9 6 3 5 1 4 8 0 2
```

Несколько забегая вперед, посмотрим, как объединение работает с множествами.

```
begin
  var p: set of integer := [7, 9, 6, 3, 5, 1, 4, 3, 9];
  var q: set of integer := [8, 0, 3, 6, 2, 4, 7, 2];
  Writeln(p);
  Writeln(q);
  Writeln(p+q)
end.
```

В этом случае вывод будет выглядеть совсем иначе, но будет содержать в полученном множестве точно такие же значения, как и прежде.

```
{9,7,6,5,4,3,1}
{8,7,6,4,3,2,0}
{9,8,7,6,5,4,3,2,1,0}
```

Метод p.Intersect(q) возвращает последовательность, являющуюся пересечением последовательностей p и q. Пересечение содержит только те элементы, которые имеются в обеих исходных последовательностях. Как и в случае с Union, полученная последовательность не будет содержать повторяющихся элементов.

```
begin
  var p := Seq(7, 9, 6, 3, 5, 1, 4, 3, 9);
  var q := Seq(8, 0, 3, 6, 2, 4, 7, 2);
  p.Println;
  q.Println;
  p.Intersect(q).Println
end.
```

Результат выполнения программы:

```
7 9 6 3 5 1 4 3 9
8 0 3 6 2 4 7 2
7 6 3 4
```

Метод p.Except(q) возвращает разность последовательностей, состоящую из элементов последовательности p без повторов, из которой исключены элементы, присутствующие в q.

```
begin
  var p := Seq(7, 9, 6, 3, 5, 1, 4, 3, 9);
  var q := Seq(8, 0, 3, 6, 2, 4, 7, 2);
  p.Println;
  q.Println;
  p.Except(q).Println
end.
```

Результат выполнения программы:

```
7 9 6 3 5 1 4 3 9
8 0 3 6 2 4 7 2
9 5 1
```

5.1.8.2 Декартово произведение (.Carthesian)

Каждый элемент декартова произведения получается, как результат выполнения заданной операции между каждым элементом первой последовательности p и каждым элементом второй q, причем составляются все возможные комбинации пар элементов. Результат представляет собой последовательность кортежей.

```
begin // p05015
  var s1 := Range('1', '5').Println(' ');
  var s2 := Range('a', 'd').Println(' ');
  s1.Cartesian(s2).Println
end.
```

Эта программа выведет на монитор следующие строки:

```
1 2 3 4 5
a b c d
(1,a) (1,b) (1,c) (1,d) (2,a) (2,b) (2,c) (2,d) (3,a) (3,b) (3,c) (3,d) (4,a)
(4,b) (4,c) (4,d) (5,a) (5,b) (5,c) (5,d)
```

Пять элементов первой последовательности и четыре второй дали в результате 5×4=20 элементов выходной последовательности.

Расширение вида p.Carthesian(q, (T, T1) -> T2) создает последовательность, каждый элемент s_k которой является результатом вычисления лямбда-выражения с элементами p_i и q_j.

```
begin // p05016
  var s1 := Range('1', '5').Println(' ');
  var s2 := Range('a', 'd').Println(' ');
  s1.Cartesian(s2, (p, q)-> p + q).Println
end.
```

Программа выполняет конкатенацию значений каждой пары элементов, заданных декартовым произведением.

```
1 2 3 4 5
a b c d
1a 1b 1c 1d 2a 2b 2c 2d 3a 3b 3c 3d 4a 4b 4c 4d 5a 5b 5c 5d
```

5.1.8.3 Чередование элементов (.Interleave)

Расширение позволяет чередовать элементы двух, трех и четырех последовательностей таким образом, что сначала идут все первые элементы, затем – все вторые и т.д.

```
begin
  var s1 := SeqGen(6, p -> 2 * p * 2 + 3).Println;
  var s2 := SeqGen(5, p -> 3 * p - 2).Println;
  s1.Interleave(s2).Println
end.
```

Если последовательности имеют различную длину, то из каждой последовательности выбирается столько элементов, сколько их в самой короткой последовательности.

```
3 7 11 15 19 23
-2 1 4 7 10
3 -2 7 1 11 4 15 7 19 10
```

5.1.8.4 Соединение элементов (метод Zip)

Метод соединяет элементы двух последовательностей с одинаковыми порядковыми номерами, выполняя с ними операцию, заданную лямбда-выражением. Как и в случае с расширением .Interleave, в операции участвует количество элементов, определяемое по более короткой последовательности.

```
begin
  var s1 := SeqGen(6, p -> 2 * p * 2 + 3).Println;
  var s2 := SeqGen(5, p -> 3 * p - 2).Println;
  s1.Zip(s2, (p, q) -> p * q).Println
end.
```

В этом примере создается последовательность, каждый элемент которой представляет собой произведение элементов исходных последовательностей.

```
3 7 11 15 19 23
-2 1 4 7 10
-6 7 44 105 190
```

5.1.8.5 Создание последовательности кортежей (метод ZipTuple)

Метод создает последовательность, каждый элемент которой является кортежем, полученным из элементов двух, трех или четырех последовательностей с одинаковыми порядковыми номерами.

```
begin // p05017
  var s1 := SeqGen(6, p -> 2 * p * 2 + 3).Println;
  var s2 := SeqGen(5, p -> 3 * p - 2).Println;
  var s3 := SeqGen(7, p -> p * p - 5 * p + 1).Println;
  s1.ZipTuple(s2, s3).Println
end.
```

В приведенном примере в кортежи объединяются элементы трех последовательностей.

```
3 7 11 15 19 23
-2 1 4 7 10
1 -3 -5 -5 -3 1 7
(3,-2,1) (7,1,-3) (11,4,-5) (15,7,-5) (19,10,-3)
```

5.1.8.6 Распаковка последовательности кортежей (метод UnZipTuple)

Этот метод делает действие, обратное действию метода ZipTuple. Он распаковывает последовательность двух-, трех- или четырехэлементных кортежей в две, три или четыре последовательности соответственно. Метод использует алгоритм с двумя и более проходами, поэтому его использование с недетерминированными последовательностями может преподнести сюрприз.

```
begin // p05018
  var s1 := SeqGen(6, p -> 2 * p * 2 + 3).Println;
  var s2 := SeqGen(5, p -> 3 * p - 2).Println;
  var s3 := SeqGen(7, p -> p * p - 5 * p + 1).Println;
  var s4 := s1.ZipTuple(s2, s3).Println;
  (s1, s2, s3) := s4.UnZipTuple;
  s1.Println;
  s2.Println;
  s3.Println
end.
```

В приведенном примере сначала создается последовательность кортежей s4, а затем распаковывается в три последовательности при помощи метода UnZipTuple и кортежного присваивания.

```
3 7 11 15 19 23
-2 1 4 7 10
1 -3 -5 -5 -3 1 7
(3,-2,1) (7,1,-3) (11,4,-5) (15,7,-5) (19,10,-3)
3 7 11 15 19
-2 1 4 7 10
1 -3 -5 -5 -3
```

Компактный способ установить длину трех последовательностей равной длине самой короткой из них – написать что-то наподобие

```
(s1, s2, s3) := s1.ZipTuple(s2, s3).UnZipTuple;
```

5.1.9 Сортировка последовательности

В PascalABC.NET имеются методы и расширения, позволяющие выполнять сортировки различной сложности.

5.1.9.1 Сортировка по возрастанию (.Sorted)

Расширение s.Sorted возвращает последовательность s, упорядоченную по возрастанию. Устаревшее, рекомендуется использовать .Order (5.1.9.3).

```
begin
  Seq(7, 9, -6, 3, 5, -1, 4, 3, -9).Println.Sorted.Println
end.
```

```
7 9 -6 3 5 -1 4 3 -9
-9 -6 -1 3 3 4 5 7 9
```

5.1.9.2 Сортировка по убыванию (.SortedDescending)

Расширение s.SortedDescending возвращает последовательность s, упорядоченную по убыванию. Устаревшее, рекомендуется использовать .OrderDescending (5.1.9.4).

```
begin
  Seq(7, 9, -6, 3, 5, -1, 4, 3, -9).Println.SortedDescending.Println
end.
```

```
7 9 -6 3 5 -1 4 3 -9
9 7 5 4 3 3 -1 -6 -9
```

5.1.9.3 Сортировка по возрастанию (.Order)

Расширение s.Order возвращает последовательность s, упорядоченную по возрастанию.

```
begin
  Seq(7, 9, -6, 3, 5, -1, 4, 3, -9).Println.Order.Println
end.
```

```
7 9 -6 3 5 -1 4 3 -9
-9 -6 -1 3 3 4 5 7 9
```

5.1.9.4 Сортировка по убыванию (.OrderDescending)

Расширение s.OrderDescending возвращает последовательность s, упорядоченную по убыванию

```
begin
  Seq(7, 9, -6, 3, 5, -1, 4, 3, -9).Println.OrderDescending.Println
end.
```

```
7 9 -6 3 5 -1 4 3 -9
9 7 5 4 3 3 -1 -6 -9
```

5.1.9.5 Сортировка по возрастанию ключа (метод OrderBy)

Метод s.OrderBy(T -> Key) возвращает последовательность s, упорядоченную по возрастанию ключа, заданного лямбда-выражением. Ключ сортировки – это некоторая величина, связанная со значением элемента так, чтобы после упорядочивания по ключам элементы расположились в нужном порядке. В простейшем случае в качестве ключа может использоваться значение элемента. Если указать p -> -p, получим сортировку по убыванию, p -> Abs(p) даст сортировку по возрастанию абсолютных значений элементов и т.д.

В примере создается, а затем сортируется последовательность двухэлементных кортежей по возрастанию значений второго элемента кортежа.

```
begin // p05019
  var s := Seq(5, 8, -4, 9, 2, 6).ZipTuple(Seq(-4, 9, 0, 5, 1, - 7));
  s.Println;
  s.OrderBy(p -> p[1]).Println
end.
```

```
(5,-4) (8,9) (-4,0) (9,5) (2,1) (6,-7)
(6,-7) (5,-4) (-4,0) (2,1) (9,5) (8,9)
```

5.1.9.6 Сортировка по убыванию ключа (метод OrderByDescending)

Метод s.OrderByDescending(T -> Key) возвращает последовательность s, упорядоченную по убыванию ключа, заданного лямбда-выражением. Работает совершенно аналогично методу s.OrderBy(T -> -Key) – обратите внимание на минус перед ключом.

5.1.9.7 Вторичная сортировка по возрастанию ключа (метод ThenBy)

Встречаются случаи, когда нужно упорядочить последовательность более чем по одному ключевому атрибуту. Например, упорядочить список людей по году рождения, а для при одинаковом годе рождения – по фамилиям. Такую добавочную (вторичную) сортировку по возрастанию ключа делает метод s.ThenBy(T -> Key). Метод указывается следующим за методом s.OrderBy или s.OrderByDescending.

```
begin // p05020
  var s1 := Seq(5, 8, -4, 9, 2, 6, -2, 5);
  var s2 := Seq(-4, 9, 0, 5, 1 - 7, 0, 5);
  var s := s1.ZipTuple(s2).Println;
  s.OrderBy(p -> p[1]).ThenBy(p -> -p[0]).Println
end.
```

```
(5,-4) (8,9) (-4,0) (9,5) (2,-6) (6,0) (-2,5)
(2,-6) (5,-4) (6,0) (-4,0) (9,5) (-2,5) (8,9)
```

В приведенном примере элементы последовательности упорядочиваются по возрастанию значений второго элемента кортежа, а при равных значениях – по убыванию первых элементов кортежа.

5.1.9.8 Вторичная сортировка по убыванию ключа (метод ThenByDescending)

Метод указывается следующим за методом s.OrderBy или s.OrderByDescending. Работает аналогично методу s.ThenBy, но упорядочивает последовательность по убыванию ключа.

5.1.10 Поиск и проверка выполнения условий

Последовательности можно поэлементно сравнивать между собой, проверять их на содержание элементов, удовлетворяющих некоторым условиям и узнавать, является ли такой элемент единственным, выбирать из последовательности конкретный элемент и т.д.

5.1.10.1 Элемент с указанным номером (метод ElementAt)

Метод s.ElementAt(n) выбирает из последовательности s элемент с номером n, при этом счет элементов, как уже было не раз сказано, ведется от нуля. Если элемента с таким номером нет, то выдается ошибка вида «Ошибка времени выполнения: Индекс за пределами диапазона. Индекс должен быть положительным числом, а его размер не должен превышать размер коллекции. Имя параметра: index». Получение такого сообщения с последующим аварийным завершением работы программы неприятно. Путей решения проблемы несколько.

- предварительно проверить, чтобы номер лежал в пределах от 0 до s.Count-1;
- принять меры по собственной обработке ***исключений*** (см. часть 12);
- воспользоваться методом s.ElementAtOrDefault(n).

5.1.10.2 Элемент с указанным номером (метод ElementAtOrDefault)

Метод s.ElementAtOrDefault(n) выбирает из последовательности s элемент с номером n, а если такого элемента нет, возвращает ***значение по умолчанию***. Значение по умолчанию – это некоторое значение, которое назначается каждому объекту в программе. Для последовательности, элементы которой имеют тип T, значением по умолчанию будет элемент типа T, который в свою очередь будет иметь значение по умолчанию, принятое для этого типа. Значениями по умолчанию, в частно-

сти, инициализируются переменные, если для них не задано начального значения. Для числовых последовательностей метод s.ElementAtOrDefault при неуспешном поиске элемента вернет элемент со значением 0. Понятно, что если элементы с таким значением в последовательности уже имеются, это может только запутать.

```
begin
  var s := Seq(2.3, 5.951, 9.4, -3.001).Println(' | ');
  s.ElementAtOrDefault(2).Println;
  s.ElementAtOrDefault(-2).Println;
  s.ElementAtOrDefault(10).Println;
end.
```

```
2.3 | 5.951 | 9.4 | -3.001
9.4
0
0
```

5.1.10.3 Если последовательность пустая... (метод DefaultIfEmpty)

Метод s.DefaultIfEmpty вернет одноэлементную последовательность со значением элемента по умолчанию, если коллекция s пуста. Неплохой вариант при нахождении среднего арифметического значения членов последовательности, выбираемых по некоторому условию.

```
begin // p05021
  var s := PartitionPoints(-1.5, 1.0, 10).Select(p -> Round(p,5)).
      Println;
  s.Where(t -> Abs(Tan(t)) > 1).DefaultIfEmpty.Average.Println;
  s.Where(t -> Tan(t) > 2).DefaultIfEmpty.Average.Println
end.
```

В приведенном примере интервал [-1.5;1] разбивается на 10 равных частей, давая последовательность из 11 точек. С помощью проецирования значения элементов округляются до пяти знаков после запятой, выводятся и образуют последовательность s типа **sequence of real**. С помощью фильтрации отбираются элементы, для которых абсолютная величина тангенса превышает единицу и находится среднее арифметическое отобранных элементов. Далее, вычисление среднего арифметического проводится для элементов, у которых значение тангенса не превышает 2. Поскольку таких элементов нет, то с помощью метода DefaultIfEmpty возвращается последовательность из одного элемента со значением 0. Но это лучше, чем получить аварийное завершение работы программы.

```
-1.5 -1.25 -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75 1
-0.6875
0
```

На случай, если ноль все же не устраивает, имеется разновидность метода, позволяющая задать значение по умолчанию: s.DefaultIfEmpty(значение типа T). Отметьте, что тип должен быть не приводимым, а точно таким же, как последовательность, которую обрабатывает этот метод. Например, мы могли бы написать

```
s.Where(t -> Tan(t) > 2).DefaultIfEmpty(7777.0).Average.Println
```

5.1.10.4 Наличие элемента в последовательности (метод Contains)

Метод s.Contains(a) возвращает True, если в последовательности имеется хотя бы один элемент со значением a и False в противном случае.

```
begin
  var s := Range(-5, 8).Println;
  s.Contains(4).Println
end.
```

```
-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8
True
```

5.1.10.5 Наличие элемента в последовательности (операция in)

Проверить, имеется ли в последовательности элемент с заданным значением можно при помощи операции **in**. Она используется в логическом выражении вида

```
Элемент in Последовательность
```

Операция возвращает True, если в последовательности имеется хотя бы один элемент с указанным значением и False в противном случае. Работает на ничтожное время медленнее метода .Contains.

```
begin
  var s := Range(-5, 8).Println;
  (4 in s).Println
end.
```

```
-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8
True
```

5.1.10.6 Есть ли в последовательности элементы? (метод Any)

Метод s.Any возвращает True, если последовательность непустая и False в противном случае. Эквивалентен выражению s.Count<>0.

В формате a.Any(T -> **boolean**) метод возвращает True, если хотя бы для одного элемент последовательности лямбда-выражение вернет True.

```
begin
  var s := Range(-5, 8).Println;
  s.Any(r -> r > 10).Println;
  s.Any(r -> r < -3).Println;
end.
```

```
-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8
False
True
```

5.1.10.7 Все ли элементы удовлетворяют условию? (метод All)

Метод s.All(T -> **boolean**) возвращает True, если для каждого элемента последовательности s лямбда-выражение вернет True. Например, так можно проверить, все ли элементы последовательности положительны, перед тем как вычислять выражение с логарифмом.

```
begin // p05022
  loop 10 do
    ArrRandom(10, -5, 50).All(p -> p > 0).Print;
  Println
end.
```

В примере десять раз генерируется последовательность из десяти случайных целых чисел, находящихся в диапазоне [-5; 50] и для каждой последовательности выводится True, если все значения в ней положительны и False в противном случае:

```
False False True False True False False False False False
```

5.1.10.8 Сравнение последовательностей (метод SequenceEqual)

Две последовательности можно поэлементно сравнить между собой. Последовательности p и q считаются равными, если для всех p_i и q_i выполняется условие $p_i=q_i$. Конечно, недопустимо писать **if** p=q **then** ...

Метод p.SequenceEqual(q) возвращает True, если последовательности равны и False в противном случае.

5.1.11 Табуляция функции с помощью последовательности

Табуляцией называют составление таблицы значений некоторой функции для заданного набора аргументов, обычно заданных на некотором интервале и одинаково отстоящими друг от друга. Такие наборы аргументов можно создавать с помощью генераторов. Для получения таблицы значений можно затем перебрать элементы в цикле foreach, вычисляя и выводя каждое значение функции, либо воспользоваться имеющимся в PascalABC.NET расширением Tabulate.

5.1.11.1 Использование цикла foreach

```
begin // p05023
  var s := PartitionPoints(-1.5, 1.0, 10).Select(p -> Round(p, 5));
  foreach var x in s do
    Println(x, Tan(x))
end.
```

Получим следующий результат, пусть пока некрасиво выданный, но сейчас у нас другая задача.

```
-1.5 -14.1014199471717
-1.25 -3.00956967386283
-1 -1.5574077246549
-0.75 -0.931596459944072
-0.5 -0.54630248984379
-0.25 -0.255341921221036
0 0
0.25 0.255341921221036
0.5 0.54630248984379
0.75 0.931596459944072
1 1.5574077246549
```

5.1.11.2 Табуляция последовательностью (.Tabulate)

Расширение s.Tabulate(T -> T1) для каждого элемента последовательности s вычисляет значение указанного лямбда-выражения, задающего функцию и создает последовательность двухэлементных кортежей вида (аргумент, функция).

```
begin // p05024
  var s := PartitionPoints (-1.5, 1.0, 10).Select(p -> Round(p, 5));
  s.Tabulate(r -> Tan(r)).Println
end.
```

В этом случае код получается короче. Но и вывод совсем иного вида:

```
(-1.5,-14.1014199471717) (-1.25,-3.00956967386283) (-1,-1.5574077246549) (-
0.75,-0.931596459944072) (-0.5,-0.54630248984379) (-0.25,-0.255341921221036)
(0,0) (0.25,0.255341921221036) (0.5,0.54630248984379) (0.75,0.931596459944072)
(1,1.5574077246549)
```

Чтобы ситуация с выводом не сильно огорчала, посмотрите один возможных способов оформления вывода, использующий лямбда-процедуру:

```
begin // p05025
  var s := PartitionPoints (-1.5, 1.0, 10).Select(p -> Round(p, 5));
  s.Tabulate(r -> Tan(r)).ForEach(r -> Writeln(r[0]:5:2, r[1]:15:8))
end.
```

```
-1.50   -14.10141995
-1.25    -3.00956967
-1.00    -1.55740772
      . . .
 0.75     0.93159646
 1.00     1.55740772
```

Имеется также и другой вариант решения, не использующий табуляцию. Здесь оформление произведено при помощи ***интерполированных строк*** (см. 6.2.26.5).

```
begin // p05026
  var s := PartitionPoints(-1.5, 1.0, 10).Select(p -> Round(p, 5));
  s.Select(r -> $'{r,5:f2}{Tan(r),15:f8}').PrintLines
end.
```

5.2 Множества set of T

Основатель теории множеств Георг Кантор определил множество, как соединение в одно целое определенных, вполне различаемых объектов, называемых элементами. Это определение слишком общее и в области информатики требует некоторой конкретизации.

Множество - редактируемая неупорядоченная совокупность уникальных перечисляемых элементов, обладающих каким-либо общим свойством, позволяющим отнести все элементы к определенному типу T.

В программах мы всегда будем иметь дело с конечными множествами, имеющими определенное количество элементов. Это количество в теории множеств именуют мощностью множества. Множество похоже на последовательность, но в отличие от нее, под все элементы множества выделяется память. Во множестве не выделяют первого, следующего и последнего элемента – множество неупорядоченно. Тем не менее, элементы множества можно перебрать по одному, но при этом нельзя указать, с какого элемента начать перебор. Еще одной особенностью множества является то, что все его элементы уникальны. Например, во множестве чисел не могут находиться два числа с одинаковыми значениями. И последнее важное отличие множества от последовательности состоит в том, что его можно редактировать: изменять значения элементов, включать дополнительные элементы и исключать существующие.

5.2.1 Создание множества

Множество описывается с использованием ключевого слова **set of**, за которым указывается тип элементов множества.

```
var s1 : set of integer;
var s2 : set of real;
```

Описание, подобное выше, приводит к созданию пустого множества, т.е. множества с мощностью ноль. Описание можно объединить с заполнением множества некоторыми элементами, перечислив их в квадратных скобках (такая запись называется ***конструктором множества***).

```
      begin
        var s1:set of integer:=[1,2,3];
        var s2:set of real:=[5,0.12];  // значения приводятся к типу real
        Writeln(s1,NewLine,s2)
      end.
{3,2,1}
{5,0.12}
```

Обратите внимание, что содержимое последовательности оператор Write выводит в фигурных скобках. Можно также пользоваться оператором Print и методом .Print в точечной нотации.

Основная цель поддержки множеств **set of T** в PascalABC.NET – совместимость с базовым Паскалем. Эти множества плохо укладываются в концепции современного программирования, поскольку для операций с ними предусмотрен скудный набор средств. Полноценной заменой им могут служить ***коллекции*** Microsoft .NET Framework (см. часть 10), такие как HashSet и SortedSet.

Генераторы множеств в языке отсутствуют, но конструктор множества существует и представляет собой список включаемых во множество значений, разделенных запятыми и заключенных в квадратные скобки. Если список пуст, создается пустое множество. Чтобы узнать мощность множества, нужно перебрать все его элементы в цикле **foreach**, либо обратиться к методу .Count, возвращающему количество элементов во множестве.

Множество можно опустошить, присвоив ему пустое множество, например, s:=[];

5.2.2 Конструктор множества

При создании множества используется конструктор, содержащий список элементов множества, разделенных запятыми и заключенный в квадратные скобки. Список может состоять из единственного элемента и даже быть пустым. В качестве элемента множества могут быть использованы литералы, выражения соответствующего типа, а также диапазонные значения вида m..n, соответствующие перечислению значений от m до n. Конструктор указывается в качестве правой части в операторе присваивания. Если тип множества не указан, а в конструкторе записаны элементы различных типов, определяется наиболее общий тип, который и объявляется базовым типом множества.

```
begin // p05027
  var s1: set of real;
  var s2 := [-10, 0, -32, 5..12, 28..34, 19];
  s1 := [8.5, 11, 13.7, -5.192];
  Println(s1);
  s2.Println
end.
```

```
{-5.192,11,13.7,8.5}
34 33 32 -10 31 29 28 -32 19 30 12 11 10 9 8 7 6 5 0
```

5.2.3 Операции над множествами

В PascalABC.NET реализовано большинство операций, которые определены в математике для множеств; остальные операции могут быть легко смоделированы.

5.2.3.1 Перебор элементов множества в цикле foreach

Собственно, ничего нового тут нет, все так же, как для последовательностей. Но это единственный способ получить доступ к элементам множества с тем, чтобы произвести с ними какое-то действие. Это понятно: если элементы множества не упорядочены и не нумерованы, нельзя напрямую обратиться к какому-то конкретному элементу. С точки зрения современных концепций программирования это архаизм.

```
begin // p05028
  var s := [5, 8, 11, 6, -3, 18, 40, 26];
  Println(s);
  var n := 0;
  foreach var c in s do
    if c.IsOdd then n += 1;
  Println('Число нечетных элементов в множестве равно', n)
end.
```

```
{11,-3,26,8,40,5,6,18}
Число нечетных элементов в множестве равно 3
```

Вот такое «письмо турецкому султану» приходится писать для того, чтобы узнать, сколько нечетных элементов содержит множество. Параметр цикла (c) поочередно предоставляет доступ к каждому элементу множества.

5.2.3.2 Добавление элемента ко множеству (Include и +=)

Классический способ, описанный Н.Виртом – использовать процедуру Include. Ее первым параметром является имя множества, вторым – добавляемый элемент. Второй способ – использовать операцию +=; при этом добавляемый элемент заключается в квадратные скобки, что превращает его в одноэлементное множество.

Оба способа проиллюстрированы в примере. Обратите внимание, что порядок следования элементов во множестве меняется хаотично.

```
begin // p05029
  var s := [5, 8, 11, 6, -3, 18, 40, 26];
  Println(s);
  s += [7];
  s += [12];
  Include(s, 0);
  Include(s, 13);
  Println(s)
end.
```

```
{11,-3,26,8,40,5,6,18}
{40,13,12,11,26,8,7,-3,5,6,18,0}
```

5.2.3.3 Удаление элемента из множества (Exclude и -=)

Н.Вирт использовал для этой цели процедуру Exclude. Ее первым параметром является имя множества, вторым – удаляемый элемент. PascalABC.NET предлагает второй способ – использовать операцию -=. Если запрашиваемый для удаления элемент во множестве отсутствует, попытка удаления ошибкой не считается. Вы можете даже получить пустое множество и бесконечно заниматься удалением из него любых элементов, что иногда и делают начинающие программировать, вместо того, чтобы просто написать s:=[ ].

```
begin // p05030
  var s := [5, 8, 11, 6, -3, 18, 40, 26];
  Println(s);
  s -= [11];
  Exclude(s, 18);
  s -= [9]; // не ошибка
  Exclude(s, 10); // не ошибка
  Println(s)
end.
```

```
{11,-3,26,8,40,5,6,18}
{40,-3,26,8,6,5}
```

5.2.3.4 Проверка наличия элемента во множестве (in)

С помощью операции **in** можно проверить, имеется ли во множестве элемент с заданным значением. Она используется в логическом выражении вида

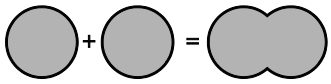
```
Элемент in Множество
```

Ниже приводится пример использования операции in

```
begin // p05031
  var n := ReadInteger;
  if n in [1..9] then
    Println('Введено натуральное однозначное число')
end.
```

Здесь множество [1..9] используется непосредственно, без объявления, и это тот самый случай, когда множество оказывается весьма полезным.

5.2.3.5 Объединение множеств (+)

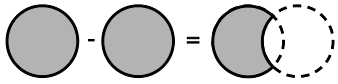


Два или более множеств можно объединить, соединив их имена знаком операции «плюс» (+). При объединении получается множество, содержащее без повторов все элементы, имеющиеся в исходных множествах.

```
begin
  Println([1..3] + [7..12] + [-3, 0, 2, 5, 9])
end.
```

```
{12,11,10,9,8,7,-3,5,3,2,1,0}
```

5.2.3.6 Разность множеств (-)

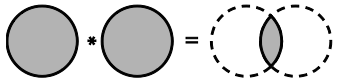


Для получения разности множеств используется операция «минус» (-). Разностью множеств p и q является множество, в которое войдут лишь те элементы из p, которых нет в q.

```
begin
  Println([1..15] - [-3, 0, 2, 5, 9])
end.
```

```
{15,14,13,12,11,10,8,7,6,4,3,1}
```

5.2.3.7 Пересечение множеств (*)



Пересечение множеств выполняется при помощи операции «звездочка» (*). Результат операции пересечения множеств – новое множество, состоящее только из тех элементов, которые присутствуют в обоих исходных множествах.

```
begin
  Println([1..15] * [-3, 0, 2, 5, 9])
end.
```

```
{5,9,2}
```

5.2.3.8 Равенство множеств (=)

Два множества равны, если они содержат одни и те же элементы.

```
begin
  Println([3, 1, 2] = [1, 2, 3], [1, 2, 3] = [1, 2, 4])
end.
```

```
True False
```

5.2.3.9 Неравенство множеств (<>)

Два множества не равны, если существует хотя бы один элемент, который содержится в одном множестве и отсутствует в другом. Это операция, обратная равенству множеств, поэтому если выполняется равенство, не выполняется неравенство и наоборот.

5.2.3.10 Строгое вложение (<)

Для множеств p и q отношение p < q истинно, когда все элементы p содержатся в q, но не все элементы q содержатся в p.

```
begin
  Println([3, 1, 2] < [1, 2, 3, 4], [1, 2, 3] < [2, 3, 1])
end.
```

```
True False
```

5.2.3.11 Нестрогое вложение (<=)

Для множеств p и q отношение p <= q истинно, когда все элементы p содержатся в q.

```
begin
  Println([3, 1, 2] <= [1, 2, 3, 4], [1, 2, 3] <= [2, 3, 1])
end.
```

```
True True
```

5.2.3.12 Строго содержит (>)

Для множеств p и q отношение p > q истинно, когда все элементы q содержатся в p, но не все элементы p содержатся в q. Это операция, обратная строгому вложению.

```
begin
  Println([1, 2, 3, 4] > [3, 1, 2], [1, 2, 3] > [2, 3, 1])
end.
```

```
True False
```

5.2.3.13 Нестрого содержит (>=)

Для множеств p и q отношение p >= q истинно, когда все элементы q содержатся в p. Это операция, обратная нестрогому вложению.

```
begin
  Println([1, 2, 3, 4] >= [3, 1, 2], [1, 2, 3] >= [2, 3, 1])
end.
```

```
True True
```

5.3 Динамические массивы array of T

В языках программирования массивы появились раньше всех других разновидностей объединения элементов.

Массив – это хранимая нумерованная последовательность однотипных элементов с непосредственным доступом к любому элементу по его номеру. Нумерация в массивах может начинаться от различных значений, а сами номера называются ***индексами*** элементов массива. В ***динамических*** массивах индексы начинаются с нуля. В отличии от последовательности, все элементы массива хранят присвоенные им значения, а также позволяют их изменять. Количество элементов в динамическом массиве также может меняться.

Поскольку динамические массивы является разновидностью последовательности, к ним применимы **все методы и расширения**, предназначенные для работы с последовательностями. Эти методы и расширения возвращают **не массив, а последовательность.** Для массивов имеются также собственные методы и расширения.

Динамический массив создается в том месте программы, где он впервые требуется. Как и у последовательности, размер массива – это количество элементов, которое в нем в данный момент содержится. Доступ к любому элементу массива осуществляется путем указания его имени, за которым в квадратных скобках следует индекс. Где-то мы это уже встречали… ну конечно, кортежи! Те же скобки и нумерация элементов от нуля.

Массивы могут иметь различное число измерений. Мы будем рассматривать одномерные массивы, а также двухмерные, которые в PascalABC.NET называются ***матрицами*** (см. часть 8). Пожалуйста запомните, что к тем матрицам, которые изучаются в курсе математики, эти матрицы имеют примерно такое же отношение, как морские свинки к морю и к свиньям. Их роднит одно: они имеют два измерения – строки и столбцы. Но об этом в свое время.

Далее по тексту термин «массив» следует понимать, как одномерный массив, а термин «матрица» – как двухмерный.

Динамический массив в программе является объектом, поэтому с ним можно работать в парадигме ООП. Как и в случае с последовательностями, наибольший комфорт в работе достигается в соединении ООП с функциональной парадигмой.

5.3.1 Создание динамических массивов

Динамический массив описывается в виде

```
var ИмяМассива : array of Тип;
```

Переменная *ИмяМассива* будет являться всего лишь ссылкой на некоторое место в памяти, где должны располагаться элементы массива. Объявленный таким образом массив еще не создан, память под него не выделена и попытка обратиться к этому массиву вызовет ошибку. Поэтому следующим шагом должно быть распределение памяти под элементы массива.

Наиболее просто распределить память под динамический массив, используя операцию **new**. С точки зрения ООП она создает объект необходимой структуры и выделяет под него память.

```
ИмяМассива := new Тип[КоличествоЭлементов]
```

Можно объединить описание массива с его созданием и, при желании, воспользоваться автовыведением типа.

```
var a := new integer[15]; // самый компактный способ
var b : array of real;
b := new real[7];
```

Имеется еще один способ распределить (а главное, перераспределить) память под массив. Этим способом чаще всего пользуются при необходимости добавить или удалить элементы массива, поскольку он не меняет значений уже имеющихся элементов. Конечно, можно всегда присвоить имени массива новый массив, созданный на основе старого, но это гораздо менее эффективно. Память перераспределяется при помощи процедуры SetLength(ИмяМассива, ЧислоЭлементов).

```
begin // p05032
  var a: array of integer;
  SetLength(a, 5); // теперь в массива 5 элементов
  (a[0], a[1], a[2], a[3], a[4]) := (10, 11, 12, 13, 14);
  a.Println;
  SetLength(a, 7); // увеличим число элементов до 7
  a[6] := 16;
  a.Println
end.
```

```
10 11 12 13 14
10 11 12 13 14 0 16
```

Элементу a[5] значение не было присвоено, поэтому оно осталось значением по умолчанию, которым инициализируются все элементы массива при выделении памяти под них. Как и ожидалось, значения существующих элементов не изменилось.

Процедура SetLength также умеет работать с массивом, память под который она не выделяла.

```
begin // p05033
  var a := new integer[4];
  (a[0], a[1], a[2], a[3]) := (10, 11, 12, 13);
  a.Println;
  SetLength(a, 7);
  a[6] := 16;
  a.Println
end.
```

```
10 11 12 13
10 11 12 13 0 0 16
```

Создание массива можно совместить с инициализацией его элементов. В этом случае также можно использовать автовыведение типа и размера массива. Поскольку тип массива задан, можно при инициализации массива вещественного типа указывать значения целочисленных типов.

```
begin // p05034
  var a: array of integer := (1, 2, 3, 4);
  a.Println;
  var b: array of real := (1.2, 5, -3.05); // 5 - целочисленная
  b.Println;
  var c := new real[4](10, 11, 12, 13); // все целочисленные
  c.Println
end.
```

```
1 2 3 4
1.2 5 -3.05
10 11 12 13
```

В языке PascalABC.NET имеется функция Arr( ), очень полезная при инициализации массивов. Она создает динамический массив из значений, заданных в качестве аргументов, поэтому все описание массива сводится лишь к указанию ключевого слова **var**. Важно запомнить, что все элементы в списке должны быть единого типа – никаких автоматических приведений!

```
begin // p05035
  var a := Arr(1, 2, 3, 4, 5, 6, 7);
  a.Println;
  var b := Arr(3.5, -2.7, 0.0, 13.9);
  b.Println
end.
```

```
1 2 3 4 5 6 7
3.5 -2.7 0 13.9
```

Отметьте также, что тип динамического массива всегда **array of** Тип и количество элементов в нем не указано. Это делает динамические массивы, содержащие элементы одного и того же типа данных, совместимыми между собой, что важно при передаче их в качестве параметров.

Актуальное количество элементов в динамическом массиве можно, как и для любой последовательности, получить при помощи вызова ИмяМассива.Count. Такой же результат дает Имя Массива.Length.

Проецирование и фильтрация, примененные к динамическим массивам, возвращают последовательность, а не массив. Это же происходит при использовании метода .Print. Последовательность можно преобразовать в динамический массив используя метод .ToArray, но злоупотреблять этим не следует.

```
begin // p05036
  var a := 10.Range.Select(p -> p / 10).Println.ToArray;
  Println((a[3] + a[7]) / 5)
end.
```

```
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
0.24
```

В приведенном примере использован генератор, создающий последовательность типа **sequence of integer**, содержащую элементы со значениями 1, 2, ... 10. Эта последовательность проецируется при помощи лямбда-выражения на последовательность типа **sequence of real**, элементы в которой имеют значения 0.1, 0.2, ... 1.0. Результат выводится на монитор, а затем последовательность преобразуется в динамический массив типа array of real, состоящий из десяти элементов с индексами от 0 до 9. Далее производится вычисление значения выражения $(a_3+a_7)/5$ и его результат также выводится. Пример демонстрирует, как легко и просто осуществляется взаимодействие динамических массивов с последовательностями.

5.3.2 Генераторы динамических массивов

Конечно, можно сгенерировать подходящую последовательность, а затем преобразовать ее в массив, используя .ToArray. Но это лишняя промежуточная операция, снижающая производительность и ухудшающая понимание алгоритма, которые реализует программа. В связи с этим для генерации динамических массивов в PascalABC.NET включен ряд специализированных генераторов. Многие из них схожи с аналогичными генераторами для последовательностей; такие генераторы легко отличить по имени: вместо Seq у них указывается Arr.

С одним из генераторов вы уже знакомы – это Arr(элемент1, элемент2, ...). Возвращается динамический массив с типом, которому соответствует каждый из элементов списка.

```
Arr(3.5, -2.7, 0.0, 13.9)
Arr(2, -5, 14, 0, 51243)
```

Имеется еще одна разновидность этого генератора, когда аргументом служит последовательность.

```
begin // p05037
  var s := SeqGen(10, p -> p * p, 1);
  var a := Arr(s);
  a.Println;
  var b := Arr(PartitionPoints(-3.0, 4.0, 14));
  b.Println;
end.
```

```
1 4 9 16 25 36 49 64 81 100
-3 -2.5 -2 -1.5 -1 -0.5 0 0.5 1 1.5 2 2.5 3 3.5 4
```

Для целых значений параметров генерируются массивы элементов типа **integer**:

`ArrFill(n,a)` – n элементов со значением a;
`ArrRandom(n,a,b)` – n случайных чисел на интервале [a;b];
`ArrRandomInteger(n,a,b)` – аналог `ArrRandom`.

```
begin // p05038
  var a := ArrFill(12, 1);
  a.Println;
  var b := ArrRandom(8, -20, 40);
  b.Println;
end.
```

```
1 1 1 1 1 1 1 1 1 1 1 1
-4 -18 -14 -9 32 -5 7 25
```

Генераторы массивов из элементов типа **real**:

`ArrFill(n,a)` – n элементов со значением a типа **real**;
`arrRandomReal(n,a,b)` – n случайных чисел на интервале [a; b]; a,b – типа **real**.

5.3.2.1 Генератор на основе лямбда-выражения (ArrGen)

Имеются четыре разновидности такого генератора.

1. ArrGen(n, **integer** -> T) – возвращает массив типа T из n элементов; лямбда-выражение задает преобразование из i = 0, 1, ... n-1 функцией F(i).

```
begin // p05039
  var a := ArrGen(10, i -> 2 * i + 1); // 10 нечетных целых
  Println(a);
  a := ArrGen(10, i -> 2 * i); // 10 четных целых
  Println(a);
  a := ArrGen(10, i -> i < 5 ? 2 * i + 1 : 2 * i ); // 2 x 5
  Println(a)
end.
```

Будут получены три строки:

```
[1,3,5,7,9,11,13,15,17,19] – 10 первых нечетных натуральных чисел
[0,2,4,6,8,10,12,14,16,18] – 10 первых неотрицательных целых чисел
[1,3,5,7,9,10,12,14,16,18] – 5 первых нечетных и 5 последующих четных.
```

2. ArrGen(n, **integer** -> T, m) – массив типа T из n элементов; лямбда-выражение задает преобразование из i = m, m+1, … m+n-1 функцией F(i). От предыдущего отличается тем, что m задает начальное значение i.

3. ArrGen(n, m, T -> T) – массив типа T из n элементов, начиная от значения m типа T; лямбда-выражение задает преобразование от предыдущего элемента к следующему. Генератор используется, когда имеется формула, связывающая функциональной зависимостью текущий элемент массива с предыдущим.

Рассмотрим программу получения значений десяти элементов массива, в котором первый элемент равен 7.1, а каждый последующий получается путем увеличения предыдущего на 0.3.

```
begin
  ArrGen(10, 7.1, x -> x + 0.3).Println
end.
```

Программа выведет значения 7.1 7.4 7.7 8 8.3 8.6 8.9 9.2 9.5 9.8

4. ArrGen(n, m, k, (T,T) -> T) – массив типа T из n элементов, в котором заданы значения первого элемента m и второго k (оба типа T); лямбда-выражение задает преобразование от пары предыдущих элементов к следующему. Генератор используется, когда имеется формула, связывающая функциональной зависимостью текущий элемент массива с двумя предшествующими.

Примером такого массива может служить ряд чисел Фибоначчи: 1, 1, 2, 3, 5, 8, … Здесь текущий элемент равен сумме двух предыдущих.

```
begin
  var a := ArrGen(13, 1, 1, (i , j) -> i + j);
  a.Println
end.
```

Будут выведены значения 1 1 2 3 5 8 13 21 34 55 89 144 233

5.3.2.2 Ввод элементов массива с клавиатуры

PascalABC.NET предлагает четыре варианта расширений для создания и/или заполнения формирования последовательностей на основе ввода значений их элементов с клавиатуры.

- ReadArrInteger(n) – создание массива на основе ввода n элементов типа integer;
- ReadArrInteger('текст приглашения',n) – то же, с приглашением ко вводу;
- ReadArrReal(n) – создание массива на основе ввода n элементов типа real;
- ReadArrReal('текст приглашения',n) – то же, с приглашением ко вводу.

Найдем среднее арифметическое s для n случайных чисел в диапазоне [a;b]. Значения n, a, b введем с клавиатуры. Затем извлечем квадратный корень из среднего арифметического суммы квадратов отклонений каждого числа от s. Полученная величина в математике называется среднеквадратичным отклонением.

```
begin // p05040
  var (n, a, b) := ReadInteger3('Введите n, a, b:');
  var m := ArrRandom(n, a, b);
  m.Println;
  var s := m.Average;
  m.Select(p -> (p - s) ** 2).Average.Sqrt.Println
end.
```

```
Введите n, a, b: 10 10 20
17 16 17 12 18 16 16 11 11 17
2.5475478405714
```

Рассмотрим подробно предпоследнюю строку кода. Элементы массива проецируются на квадрат разности со значением s. Проецирование порождает последовательность, для которой .Average возвращает среднее значение типа **real**. Для этого типа определен метод .Sqrt, возвращающий значение квадратного корня. Его значение мы выводим на монитор.

Давно мы не обращались к кодам в базовом Паскале. Посмотрим, как эту же задачу можно написать на ТурбоПаскаль. В этом языке динамических массивов нет, поэтому приходится заранее объявлять массив, содержащий заведомо больше элементов, чем может понадобиться (в реальности почему-то стоит только предположить, что чисел будет не больше ста, как сразу находится пользователь, которому позарез нужно сто одно число). Приходится объявлять максимальный размер массива константой, чтобы потом уж если исправлять, так в одном месте.

```
const
  nn = 100;

var
  a, b, n, c, i: integer;
  m: array[1..nn] of integer;
  s, s1: real;

begin
  Randomize;
  Write('Введите n, a, b: ');
  Read(n, a, b);
  c := 0;
  for i := 1 to n do
  begin
    m[i] := a + Random(b - a + 1);
    Write(m[i], ' ');
    c := c + m[i]
  end;
  Writeln;
  s := c / n;
  s1 := 0;
  for i := 1 to n do
    s1 := s1 + Sqr(m[i] - s);
  Writeln(Sqrt(s1 / n))
end.
```

А ведь на самом деле «это» - всего лишь элементарная задачка. Можете себе представить, насколько быстрее, проще и нагляднее писать на PascalABC.NET более сложные вещи.

И еще одно. Знаете, сколько нужно времени, чтобы приведенная выше программа на PascalABC.NET стала работать с числами типа real? Ровно столько, сколько

требуется дописать четыре буквы к ArrRandom с тем, чтобы получить ArrRandomReal. В программе на ТурбоПаскаль придется перенести описание переменной с в real, изменить тип у массива m, и поколдовать с выражением a + Random(b - a + 1).

Можно ли было решить эту задачу без массивов, на одних только последовательностях – ведь они работают быстрее? Увы, но нет. Подумайте почему, вспомнив о ленивости при работе с последовательностями и о том, что порожденная генератором последовательность SeqRandom является недетерминированной.

5.3.3 Операции со всеми элементами массива

Рассмотрим возможности, которые PascalABC.NET предоставляет в работе с массивом, как с единым объектом, и в работе со всеми без исключения элементами массива.

Для динамического массива a длиной n определены следующие функции и методы расширения:

- a.Count, a.Length и Lengh(a) возвращают n - длину массива;
- a.High и High(a) возвращают значение индекса последнего элемента массива длины n, равного n-1;
- a.Low и Low(a) возвращают 0 – значение индекса первого элемента массива.

Копию массива a можно получить при помощи функции Copy(a). Такую копию можно присвоить другому динамическому массиву.

```
begin // p05041
  var a := Arr(Range(10, 19));
  a.Println;
  var b := Copy(a); // копия массива a
  var c := a; // копируется ссылки и c - просто еще одно имя для a
  c[0] := 77; // a[0] тоже изменится
  a.Println;
  b.Println;
  c.Println
end.
```

```
10 11 12 13 14 15 16 17 18 19 (a)
77 11 12 13 14 15 16 17 18 19 (a)
10 11 12 13 14 15 16 17 18 19 (b)
77 11 12 13 14 15 16 17 18 19 (c)
```

Если определены массивы a и b, то как и в случае с последовательностями, присваивание вида b := a не приведет к поэлементному копированию массива a в b – скопируется лишь ссылка, после чего имя b станет еще одним именем массива a. Следует также помнить, что при размещении в b другой ссылки, массив снова будет связан только с именем a.

5.3.3.1 Перебор элементов массива в цикле

Элементы массива можно перебрать в любом из пяти видов циклов, имеющихся в языке. Для перебора всего массива удобнее всего использовать циклы ***for*** и ***foreach***. Цикл ***for*** используют, если требуется работа с индексом массива, поскольку в нем доступ к элементу массива производится именно по индексу. Если нужно только получить значение элемента, проще использовать цикл ***foreach***, но в теле этого цикла массив считается доступным лишь на чтение.

```
begin
  var a := ArrRandom(10, -20, 30);
  a.Println;
  for var i := 0 to a.High do
    a[i] := 2 * a[i] + 1;
  a.Println
end.
```

```
-9 -17 24 -4 6 22 -16 16 -8 26
-17 -33 49 -7 13 45 -31 33 -15 53
```

```
begin
  var a := ArrRandom(10, -20, 30);
  a.Println;
  var s := 0;
  foreach var b in a do
    s += b * b - 5;
  s.Println
end.
```

```
21 -5 19 12 -15 12 -1 9 -17 -2
1665
```

Отметьте, что в случае использования цикла ***foreach*** не имеет значения, какую последовательность перебирать – ***sequence*** или ***array*** – код выглядит одинаково. Но вы же помните, что в данном, конкретном случае, SeqRandom сослужил бы нам плохую службу, перебрав в ***foreach*** совсем не те значения, которые вывел в Println?

В прочих видах цикла начальное значение индекса нужно задавать перед входом в цикл, а затем менять в теле цикла и проверять на конечное значение.

Рассмотрим еще один, «творческий подход» к использованию цикла ***foreach***. Создадим из нужного набора индексов последовательность и переберем ее в цикле.

```
begin // p05042
  var a := ArrRandom(ReadInteger('n='), -20, 30);
  a.Println;
  foreach var i in a.High.Times do
    a[i] := 2 * a[i] + 1;
  a.Println
end.
```

```
n= 8
4 -4 -4 8 30 14 6 17
9 -7 -7 17 61 29 13 17
```

a.High возвращает максимальное значение индекса, а расширение .Times – это генератор последовательности, создающий ряд значений, начиная от нуля. Получаем 0, 1, 2, … a.High – это и есть нужный набор индексов. Запомните этот прием – он будет полезен при «экзотических» выборках элементов массива, когда последовательность индексов окажется не такой простой. Это «ответ» PascalABC.NET тем, кто критикует Паскаль за цикл for с шагом только 1 или -1.

5.3.3.2 Перебор элементов массива (.ForEach)

Расширение a.ForEach(T -> ()) позволяет задать лямбда-процедуру, которая применяется к каждому элементу массива a.

```
begin // p05043
  var a := ArrRandom(ReadInteger('n='), -20, 20);
  a.Println;
  var s := 0;
  a.ForEach(d-> begin s += d end);
  s.Println
end.
```

```
n= 8
-10 -19 14 16 9 9 -10 5
14
```

Лямбда-процедура может использовать не только значение элемента массива, но также его индекс: a.ForEach((v,i) -> ()), где v – значение элемента, i – индекс.

```
begin
  var a := ArrRandom(ReadInteger('n='), -20, 20);
  a.ForEach((v, i)-> begin Writeln('a[', i+1, ']=', v) end)
end.
```

```
n= 3
a[1]=13
a[2]=-9
a[3]=9
```

5.3.3.3 Арифметические операции + и * с массивами

Если a и b – массивы, то операция a + b создает массив, в котором элементы массива b следуют за элементами массива a, т.е. осуществляет слияние массивов. Операция вида n * a (а также, a * n), формируют массив, состоящий из n повторения массива a.

```
begin // p05044
  var a := ArrRandom(4, -20, 30);
  a.Println;
  var b := ArrRandom(5, 0, 35);
  b.Println;
  (a + b + a).Println;
  (2 * a + b).Println;
end.
```

```
19 28 30 -1
22 29 35 8 33
19 28 30 -1 22 29 35 8 33 19 28 30 -1
19 28 30 -1 19 28 30 -1 22 29 35 8 33
```

5.3.3.4 Проецирование (метод Select)

Как и в случае с последовательностями (см. 5.1.2.5), метод .Select переводит каждый элемент массива в новое состояние. На выходе получается последователь-

ность, поэтому если нужен массив, дополнительно используется расширение .ToArray.

Заменим элементы массива из 12 целых чисел, случайно заданных на интервале от -10 до 20, их кубами.

```
begin // p05045
  var a := ArrRandom(12, -10, 20);
  a.Println;
  a := a.Select(b -> b * b * b).ToArray;
  a.Println
end.
```

```
17 12 11 6 -2 0 -7 9 -4 11 11 -5
4913 1728 1331 216 -8 0 -343 729 -64 1331 1331 -125
```

Если вы забудете указать .ToArray и попытаетесь присвоить результат проецирования переменной, объявленной как массив, компилятор выдаст ошибку «Нельзя преобразовать тип IEnumerable<тип> к array of тип»

5.3.3.5 Проецирование (метод SelectMany)

Работа метода .SelectMany была описана для последовательностей (см. 5.1.2.6). В отличие от метода .Select, к результату повторно применяется проецирование. Следовательно, если после первого проецирования получаются несколько последовательностей, они будут объединены в одну.

Метод SelectMany имеет несколько разновидностей. Приведем две самые употребительные из них.

- a.SelectMany(ai -> si) – проецирует каждый элемент ai массива a в последовательность si и возвращает последовательность, полученную объединением si;
- s.SelectMany((ai, i) -> si) – проецирует каждый элемент ai массива a в последовательность si с учетом индекса элемента i и возвращает последовательность, полученную объединением si.

Рассмотрим пример. Пусть дан массив из десяти случайных чисел и требуется найти наиболее часто встречающуюся цифру. Если таких цифр несколько, выведем меньшую из них. Ниже (см. 5.3.9) приведено короткое и эффективное решение подобной задачи при помощи расширения .GroupBy, а пока используем цикл.

```
begin // p05046
  var a := ArrRandom(10, 1, 1000);
  a.Println;
  var s := a.Select(t -> t.ToString).SelectMany(t -> t);
  var r := new integer[10]; // счетчики повторов цифр 0..9
  for var i := 0 to 9 do // очистка счетчиков
    r[i] := 0;
  foreach var t in s do
    r[t.ToDigit] += 1;
  var j := r.IndexMax;
  Println('Цифра', j, 'встретилась', r[j], 'раз(а)')
end.
```

```
288 343 998 904 461 916 601 980 987 563
Цифра 9 встретилась 6 раз(а)
```

Конечно, такого результата можно было добиться и менее экзотическим способом.

5.3.3.6 Максимум, минимум, сумма, произведение и среднее

И здесь все точно так же, как для последовательностей (см. 5.1.2.7). Можно получить максимальное a.Max и минимальное a.Min значения элементов массива. Имеются методы расширения a.MaxBy и a.MinBy, позволяющие указать лямбда-выражение, задающее предварительное преобразование, а также a.LastMaxBy и a.LastMinBy, возвращающие последний из найденных после преобразования элементов.

Сумма значений всех элементов массива a может быть получена при помощи метода a.Sum, произведение – при помощи метода a.Product, а метод a.Average вернет среднее арифметическое. Если нужно найти сумму, произведение или среднее арифметическое значений не самих элементов массива, а некоторой функции от них, при вызове расширения можно указать соответствующее лямбда-выражение. Как и в случае с последовательностями, метод a.Product возвращает значение типа real.

Пример использования этих возможностей был показан выше (см. 5.3.2.2). В качестве еще одного примера рассмотрим нахождение суммы квадратов элементов массива.

```
begin
  var a:=ArrRandom(8,-20,30);
  a.Println;
  a.Sum(k -> k * k).Println
end.
```

```
9 -18 16 -9 17 -7 -3 9
1170
```

Надо отметить, что приведенный пример, как и многие другие, лишь иллюстрирует работу с расширением .Sum для массивов. На практике вся задача решается одним единственным оператором:

```
begin // p05047
  ArrRandom(8, -20, 30).Println.Sum(k -> k * k).Println
end.
```

5.3.3.7 Агрегирование элементов (метод Aggregate)

Метод .Aggregate((T, T) -> T) возвращает результат применения к элементам массива агрегатной функции, заданной лямбда-выражением. Функция вначале применяется к первому и второму элементам массива (a[0] и [1]), затем к возвращенному ей результату и третьему элементу, далее – к результату и четвертому элементу и так далее.

```
begin // p05048
  Arr(6, -3, 11, 8, -2).Println.Aggregate((s, j)-> s + j).Println
end.
```

Программа выведет следующий результат:

```
6 -3 11 8 -2
20
```

Вначале в качестве s берется значение a[0]=6, в качестве j – значение a[1]=-3. В соответствии с лямбда-выражением значения суммируются и получается число 3, которое теперь принимается за s. В качестве j берется a[2]=11. Далее действия повторяются, и конечный результат будет равен сумме всех элементов массива.

Метод a.Aggregate(b, (T, T) -> T) возвращает применения к элементам массива агрегатной функции, заданной лямбда-выражением, принимая значение b в качестве начального. На первом шаге функция применяется к b и a[0]. На втором шаге аргументами функции будут возвращенный результат и a[1], и так далее.

```
begin // p05049
  Arr(6, -3, 11, 8, -2).Println.Aggregate(1, (s, j) -> s * j).Println
end.
```

Программа выведет следующий результат:

```
6 -3 11 8 -2
3168
```

Вычисляется произведение элементов, поэтому в качестве начального значения задана единица. Тип результата определяется типом начального значения, поэтому в случае умножения целых величин остерегайтесь переполнения.

Оба рассмотренных выше примера – чисто иллюстративные. Здесь массив не имеет никаких преимуществ перед последовательностями, более того, проигрывает им вследствие больших затрат на реализацию.

5.3.3.8 Перестановка элементов в обратном порядке (метод Reverse)

Метод s.Reverse возвращает последовательность из элементов массива, расположенных в обратном порядке.

```
begin // p05050
  var a := Arr(7, 9, -6, 3, 5, -1, 4, 3, -9);
  a.Println;
  var b:=a.Reverse;
  b.Println
end.
```

```
7 9 -6 3 5 -1 4 3 -9
-9 3 4 -1 5 3 -6 9 7
```

5.3.4 Выборки элементов из массива

Из массива a некоторого типа T можно выбирать подпоследовательность. Выборки делаются так же, как и для последовательностей, давая в результате последовательность (см. 5.1.3). Расширение .Distinct (отбор уникальных элементов) и метод фильтрации Where также создают последовательности. Для формирования из последовательности массива используйте расширение .ToArray.

В качестве примера рассмотрим задачу вычисления среднего геометрического значений p и q по приведенной формуле, где p – сумма квадратов элементов массива из 12 случайных чисел в диапазоне от -20 до 30 с четным значением, а q – сумма квадратов элементов этого же массива с нечетным значением без повторов.

$$m = \sqrt{p \times q}$$

```
begin // p05051
  var a := ArrRandom(12, -20, 30);
  a.Println;
  var (b, c) := a.Partition(t -> t.IsEven);
  var (p, q) := (b.Sum(t -> t * t), c.Distinct.Sum(t -> t * t));
  Sqrt(p * q).Println
end.
```

```
16 -3 -14 -4 -13 17 25 -20 19 24 28 17
1799.24539738191
```

В программе речь о массиве и его элементах, но хорошо видно, что никаких новых знаний не потребовалось. Выборки все время стремятся превратиться в последовательности, а работать с последователями вы уже умеете. В программе не потребовалось ни разу (!) явно обратиться к какому-либо элементу массива – это особенность программирования в функциональном стиле. Переменные *b* и *c* содержат ссылки на последовательности, но поскольку вычисления с последовательностями отложенные (ленивые), как таковые, последовательности создаваться не будут, а их элементы по мере получения сразу будут использоваться для вычисления значений *p* и *q*.

Сравните код PascalABC.NET с программой на диалекте Borland Pascal 7.0.

```
Const
  n = 12;

var
  a, a1: array[1..n] of integer;
  i, j, k: integer;
  nf: boolean;
  p, q: real;

begin
  Randomize;
  for i := 1 to n do
  begin
    a[i] := Random(51) - 20;
    Write(a[i], ' ');
  end;
  Writeln;
  p := 0;
  j := 1;
  a1[j] := a[j];
  for i := 1 to n do
    if Odd(a[i]) then
    begin
      nf := True;
      for k := 1 to j do
        if a[i] = a1[k] then
        begin
          nf := False;
          break
        end;
      if nf then
      begin
        Inc(j);
        a1[j] := a[i]
      end
    end
    else
      p := p + Sqr(a[i]);
  q := 0;
  for i := 1 to j do
    q := q + Sqr(a1[i]);
  Writeln(Sqrt(p * q))
end.
```

«Веселенький» код получился. Больше всего хлопот в нем оказалось из-за получения уникальности в подвыборке элементов с нечетными значениями, но что поделаешь – встречаются еще и не такие проблемы.

5.3.5 Срезы

Срезы «пришли» в PascalABC.NET из языка Python и имеют похожий синтаксис, но гораздо раньше они появились в языке Fortran-90 под именем сечений. Срез позволяет получить из динамического массива некоторый динамический подмассив элементов того же типа, индексы которого будут начинаться от нуля. В общем случае срез конструируется при помощи указания в квадратных скобках трех выражений, приводящихся к типу **integer** и разделенных двоеточиями.

Срез массива a[m:n:h] возвращает подмассив элементов a, имеющих индексы от m до n (элемент номер n не рассматривается). Значение h указывает шаг, с которым выбираются элементы. Разработчики PascalABC.NET не определили специального названия для конструкции m:n:h, поэтому в данной книге по аналогии с языком Fortran-90, эта тройка будет называться ***индексным триплетом***. По умолчанию полагается, что значение m равно нулю, значение n равно длине массива и значение h равно единице, т.е. индексный триплет охватывает весь массив. Если используется какое-либо из значений по умолчанию, соответствующий элемент можно опустить. При шаге h = 1 можно индексный триплет указывать в виде [m:n], поэтому конструкции [:n] и [m:] также синтаксически верны.

Рассмотрим несколько примеров записи срезов массива.

a[3:7] – элементы массива с индексами 3, 4, 5, 6 (элементы с номерами 4, 5, 6, 7);

a[:4] – элементы с индексами 0, 1, 2, 3;

a[2:] – элементы с индексами от 2 и до конца массива;

a[::2] – элементы массива с четными индексами (0, 2, 4, …);

a[1::2] – элементы массива с нечетными индексами (1, 3, 5, …);

a[3,18,4] – элементы с индексами 3, 7, 11, 15;

a[-2:6] – ошибка: элемент с индексом -2 не существует;

a[5,9,-1] – если конечный индекс больше начального, при отрицательном шаге не будет выбрано ни одного элемента;

a[10:0:-1] – элементы с индексами от 10 до 0 в обратном порядке (реверс).

Приведенная ниже программа создаст массив из 20 случайных элементов и переставит в обратном порядке элементы с индексами с 8 по 13.

```
begin // p05052
  var a := ArrRandom(20, -50, 50);
  a.Println;
  a := a[:8] + a[13:7:-1] + a[14:];
  a.Println
end.
```

```
36 9 36 31 38 -43 38 25 5 12 -3 -46 -45 -14 27 -30 8 50 -48 27
36 9 36 31 38 -43 38 25 -14 -45 -46 -3 12 5 27 -30 8 50 -48 27
```

Просто и эффектно срезы осуществляют циклический сдвиг.

```
begin // p05053
  var a := Range(10, 19).ToArray;
  a.Println;
  var b:=a[1:] + a[:1]; // циклический сдвиг влево на 1
  b.Println;
  b:=a[a.High:] + a[:a.High]; // циклический сдвиг вправо на 1
  b.Println;
  b:=a[5:] + a[:5]; // циклический сдвиг влево на 5
  b.Println
end.
```

```
10 11 12 13 14 15 16 17 18 19
11 12 13 14 15 16 17 18 19 10
19 10 11 12 13 14 15 16 17 18
15 16 17 18 19 10 11 12 13 14
```

Рассмотрим пример функции, принимающей массив и возвращающей в качестве результата массив после циклического сдвига. Функция будет иметь параметр k, указывающий, величину и направление сдвига. При k>0 сдвиг производится вправо на k элементов, иначе - влево на –k элементов. Если k кратно длине массива, то

сдвиг не осуществляется, поэтому имеет смысл предварительно найти остаток от деления k на длину массива.

```
// p05054
function ЦиклСдвиг(a: array of integer; Сдвиг: integer): array of integer;
begin
  var СдвигВправо := (Сдвиг >= 0); // направление сдвига
  var n := a.Length; // длина массива
  var k := Abs(Сдвиг) mod n; // величина сдвига
  if k = 0 then
    Result := Copy(a)
  else
    Result := СдвигВправо ? a[n - k:n] + a[:n - k] : a[k:] + a[:k]
end;

begin
  var a := Range(10, 19).ToArray;
  for var i := 0 to 10 do
    ЦиклСдвиг(a, i).Println;
  Writeln;
  for var i := 0 to 10 do
    ЦиклСдвиг(a, -i).Println
end.
```

Тестирующая программа организует сдвиги вправо и влево на количество элементов, изменяющееся от нуля до десяти. Ее быстродействие достаточно велико, потому что она сразу строит результирующий массив, а не сдвигает элементы в цикле по одному. Ниже приводится результат работы тестового примера.

```
10 11 12 13 14 15 16 17 18 19
19 10 11 12 13 14 15 16 17 18
18 19 10 11 12 13 14 15 16 17
17 18 19 10 11 12 13 14 15 16
16 17 18 19 10 11 12 13 14 15
15 16 17 18 19 10 11 12 13 14
14 15 16 17 18 19 10 11 12 13
13 14 15 16 17 18 19 10 11 12
12 13 14 15 16 17 18 19 10 11
11 12 13 14 15 16 17 18 19 10
10 11 12 13 14 15 16 17 18 19

10 11 12 13 14 15 16 17 18 19
11 12 13 14 15 16 17 18 19 10
12 13 14 15 16 17 18 19 10 11
13 14 15 16 17 18 19 10 11 12
14 15 16 17 18 19 10 11 12 13
15 16 17 18 19 10 11 12 13 14
16 17 18 19 10 11 12 13 14 15
17 18 19 10 11 12 13 14 15 16
18 19 10 11 12 13 14 15 16 17
19 10 11 12 13 14 15 16 17 18
10 11 12 13 14 15 16 17 18 19
```

5.3.6 О «мягких» срезах

Если индексный триплет укажет несуществующий индекс, во время выполнения программы возникнет исключение.

```
begin // p05055
  var a := ArrRandom(20, -50, 50);
  a.Println;
  a := a[:8] + a[13:7:-1] + a[14:21];
  a.Println
end.
```

Массив a имеет 20 элементов, индексы которых находятся в диапазоне 0..19. Попытка выполнить срез a[14:21] приведет к получению сообщения «Ошибка времени выполнения: Параметр to за пределами диапазона».

Избежать генерации подобных исключений позволяют так называемые «мягкие» срезы, которые ограничивают срез только реально существующими индексами. Для получения мягкого среза после имени массива указывают вопросительный знак.

```
begin // p05056
  var a := ArrRandom(20, -50, 50);
  a.Println;
  a := a[:8] + a[13:7:-1] + a?[14:21];
  a.Println
end.
```

```
-17 -14 -28 13 49 11 -35 -25 -24 1 30 -40 -41 20 27 -27 39 -15 -13 1
-17 -14 -28 13 49 11 -35 -25 20 -41 -40 30 1 -24 27 -27 39 -15 -13 1
```

Мягкий срез – в некоторой степени «аварийное» средство, злоупотреблять им не следует. Если все срезы в программе мягкие, можно говорить о безалаберности, либо недостаточной квалификации ее автора.

5.3.7 Операции над массивом

Операции над массивом осуществляют процедуры, либо расширения, реализованные как процедуры, потому что функции возвращают уже другой, новый динамический массив. Понятно, что тип массива при этом поменять невозможно.

5.3.7.1 Преобразование элементов массива (.ConvertAll)

Расширение a.ConvertAll(T -> T1) возвращает в качестве результата массив, состоящий из элементов исходного массива a, функция преобразования которых задана лямбда-выражением. Если массив не нужен, а достаточно последовательности, пользуйтесь проецированием Select. В приведенном примере из значения каждого элемента массива извлекается квадратный корень, а затем результат округляется до трех цифр после запятой.

```
begin // p05057
  var a := ArrGen(6, i -> i + 10);
  a.Println;
  a.ConvertAll(p -> Round(Sqrt(p), 3)).Println
end.
```

```
10 11 12 13 14 15
3.162 3.317 3.464 3.606 3.742 3.873
```

Другой вариант расширения позволяет дополнительно сопоставить преобразование с индексом элемента. Пусть нам нужно сделать такое же преобразование, как в предыдущем случае, но только для четных по порядку элементов массива. Индексация ведется от нуля, поэтому четные по порядку элементы имеют нечетные индексы. Условная операция нам в помощь, как говорится.

```
begin // p05058
  var a := ArrGen(6, i -> i + 10);
  a.Println;
  a.ConvertAll((p, i) -> i.IsOdd ? Round(Sqrt(p), 3) : p).Println
end.
```

```
10 11 12 13 14 15
10 3.317 12 3.606 14 3.873
```

5.3.7.2 Заполнение массива значениями (.Fill)

Процедура a.Fill(**integer** -> T), реализованная как расширение, заполняет элементы массива a типа T значениями, задаваемыми лямбда-выражением, описывающим значение элемента в виде некоторой функции от его индекса.

```
begin // p05059
  var a := new real[8];
  a.Fill(i -> 3 * (i + 1) / 5 - 1);
  a.Println
end.
```

```
-0.4 0.2 0.8 1.4 2 2.6 3.2 3.8
```

5.3.7.3 Замена значения элемента по всему массиву (.Replace)

Процедура a.Replace(x, y), реализованная как расширение, находит в массиве a элементы со значением x и заменяет их значением y. Если не найдет ни одного, - ничего не заменит и ошибки не возникнет.

```
begin // p05060
  var a := ArrRandom(15, 1, 9);
  a.Println;
  a.Replace(2, 0);
  a.Println
end.
```

```
4 7 2 1 2 9 4 2 8 7 3 2 1 5 7
4 7 0 1 0 9 4 0 8 7 3 0 1 5 7
```

5.3.7.4 Перестановка элементов в обратном порядке (Reverse)

Процедура Reverse(a) переставляет элементы массива a в обратном порядке.

```
begin // p05061
  var a := ArrGen(10, i -> i + 10);
  a.Println;
  Reverse(a);
  a.Println
end.
```

```
10 11 12 13 14 15 16 17 18 19
19 18 17 16 15 14 13 12 11 10
```

Вторая разновидность процедуры, Reverse(a, m, n), переставляет в обратном порядке n элементов массива a, начиная с элемента, имеющего индекс m.

```
begin // p05062
  var a := ArrGen(10, i -> i + 10);
  a.Println;
  Reverse(a, 3, 4);
  a.Println
end.
```

```
10 11 12 13 14 15 16 17 18 19
10 11 12 16 15 14 13 17 18 19
```

5.3.7.5 Перемешивание элементов случайным образом (Shuffle)

Процедура Shuffle(a) перемешивает элементы массива a случайным образом.

Расширение a.Shuffle возвращает массив случайным образом перемешанных элементов массива a.

```
begin // p05063
  var a := ArrGen(10, i -> i + 10);
  a.Println;
  var b := a.Shuffle;
  b.Println;
  Shuffle(a);
  a.Println
end.
```

```
10 11 12 13 14 15 16 17 18 19
17 19 18 13 14 16 15 11 10 12
13 19 14 17 12 16 18 11 10 15
```

5.3.7.6 Сортировка элементов массива (Sort)

- Процедура Sort(a) и расширение a.Sort сортируют элементы массива a по возрастанию их значений.
- Процедура Sort(a,(p, q) -> **integer**) и расширение a.Sort(p, q) -> **integer**) сортируют элементы массива a в соответствии со значением функции сравнения двух соседних элементов массива p и q, заданной лямбда-выражением целочисленного типа.

- Процедура Sort(a,(p, q) -> **boolean**) сортирует элементы массива a в соответствии со значением функции сравнения двух соседних элементов массива p и q, заданной лямбда-выражением логического типа типа.

Функции сравнения позволяют сформировать достаточно сложные критерии сортировки, но для большинства случаев достаточно запомнить, что критерии вида (p, q) -> q - p и (p, q) -> q < p задает сортировку по невозрастанию, а критерии вида (p, q) -> p – q и (p, q) -> p < q задает сортировку по неубыванию.

```
begin // p05064
  var a := ArrRandom(10, 0, 9);
  a.Println;
  Sort(a, (p, q) -> q - p);
  a.Println;
  Sort(a, (p, q) -> q < p);
  a.Println;
  Sort(a, (p, q) -> p - q);
  a.Println;
  Sort(a, (p, q) -> p < q);
  a.Println;
  a.Sort; // пример вызова расширения
  a.Println
end.
```

```
1 7 0 6 6 0 9 2 2 7
9 7 7 6 6 2 2 1 0 0
9 7 7 6 6 2 2 1 0 0
0 0 1 2 2 6 6 7 7 9
0 0 1 2 2 6 6 7 7 9
0 0 1 2 2 6 6 7 7 9
0 0 1 2 2 6 6 7 7 9
```

5.3.7.7 Преобразование элементов массива (.Transform)

Расширение a.Transform(T -> T) преобразует элементы массива a по правилу, заданному лямбда-выражением. Тип элементов массива менять, естественно, нельзя.

```
begin // p05065
  var a := ArrRandom(10, 0, 9);
  a.Println;
  a.Transform(p -> 3 * p * p - 5);
  a.Println
end.
```

```
5 8 2 1 2 4 8 5 2 1
70 187 7 -2 7 43 187 70 7 -2
```

5.3.7.8 Поиск элементов массива по условию

- Расширение a.Find(T -> **boolean**) возвращает значение первого из элементов массива a, для которого истинно указанное лямбда-выражение. Если такого элемента нет, возвращается нулевое значение для типа, который имеют элементы массива.

- Расширение a.FindAll(T -> **boolean**) возвращает массив (не последовательность!) значений элементов массива a, для которых истинно указанное лямбда-выражение. Если ни одного элемента не найдено, возвращается массив нулевой длины.

```
begin // p05066
  var a := ArrRandom(20, 0, 9);
  a.Println;
  a.Find(p -> p > 6).Println;
  a.FindAll(p -> p > 6).Println
end.
```

```
2 9 1 4 6 0 6 8 5 2 2 9 4 6 1 7 8 9 5 5
9
9 8 9 7 8 9
```

5.3.8 Операции с индексами массива

До сих пор мы рассматривали операции, которые проводились с элементами массива. Но элементы массива, в отличие от элементов последовательности, имеют индексы, определяющие их местоположение в массиве. В PascalABC.NET имеется набор операций, позволяющий выполнять ряд действий, в которых участвуют индексы. Примером могут быть срезы (см. 5.3.5), основанные на индексах. Кроме этого, можно определять индексы элементов, удовлетворяющих тем или иным условиям с тем, чтобы затем выбирать эти элементы или изменять их значения.

5.3.8.1 Индекс максимального элемента в массиве

- Расширение a.IndexMax возвращает индекс первого максимального элемента в массиве a.
- Расширение a.IndexMax(k) возвращает индекс первого максимального элемента в массиве a, начиная его поиск с элемента, имеющего индекс k.
- Расширение a.LastIndexMax возвращает индекс последнего максимального элемента в массиве a.
- Расширение a.LastIndexMax(k) возвращает индекс последнего максимального элемента в массиве a, рассматривая элементы от начала массива и ограничивая поиск элементом с индексом k.

```
begin // p05067
  var a := ArrRandom(10, 0, 9);
  a.Println;
  a.IndexMax.Println; // первый максимальный
  a.IndexMax(5).Println; // первый максимальный, начиная с элемента №6
  a.LastIndexMax.Println; // последний максимальный
  a.LastIndexMax(7).Println; // последний максимальный, просмотр первых 8
end.
```

```
7 0 1 8 7 8 2 3 7 8
3
5
9
5
```

5.3.8.2 Индекс минимального элемента в массиве

- Расширение a.IndexMin возвращает индекс первого минимального элемента в массиве a.
- Расширение a.IndexMin(k) возвращает индекс первого минимального элемента в массиве a, начиная его поиск с элемента, имеющего индекс k.
- Расширение a.LastIndexMin возвращает индекс последнего минимального элемента в массиве a.
- Расширение a.LastIndexMin(k) возвращает индекс последнего минимального элемента в массиве a, рассматривая элементы от начала массива и ограничивая поиск элементом с индексом k.

5.3.8.3 Бинарный поиск индекса элемента (.BinarySearch)

Расширение a.BinarySearch(x) производит бинарный поиск индекса элемента со значением x в массиве a, предварительно **упорядоченном по неубыванию**. Если элемент не найден, возвращается **произвольное** отрицательное значение. Если элементов с искомым значением несколько, будет найден индекс любого (не обязательно первого от начала) из них.

```
begin // p05068
  var a := ArrRandom(10, 0, 9);
  a.Sort;
  a.Println;
  a.BinarySearch(5).Println
end.
```

```
0 3 3 5 5 5 6 9 9 9
4
```

5.3.8.4 Поиск индекса элемента, удовлетворяющего условию

- Расширение a.FindIndex(T -> **boolean**) возвращает индекс первого из элементов массива a, для которого будет истинным указанное в качестве аргумента лямбда-выражение. Если такой элемент не будет найден, возвращается -1.
- Расширение a.FindIndex(k, T -> **boolean**) возвращает индекс первого из элементов массива a, для которого будет истинным указанное в качестве

аргумента лямбда-выражение, просматривая элементы начиная с имеющего индекс k. Если такой элемент не будет найден, возвращается -1.

- Расширение a.FindLastIndex(T -> **boolean**) возвращает индекс последнего из элементов массива a, для которого будет истинным указанное в качестве аргумента лямбда-выражение. Если такой элемент не будет найден, возвращается -1.
- Расширение a.FindLastIndex(k, T -> **boolean**) возвращает индекс последнего из элементов массива a, для которого будет истинным указанное в качестве аргумента лямбда-выражение, просматривая элементы от начала и не дальше элемента, имеющего индекс k. Если такой элемент не будет найден, возвращается -1.

```
begin // p05069
  var a := ArrRandom(20, 0, 9);
  a.Println;
  a.FindIndex(5, p -> p >= 7).Println;
  a.FindLastIndex(8, p -> p = 3).Println
end.
```

```
9 2 8 1 2 6 1 0 2 1 2 2 8 9 0 8 0 3 2 0
12
-1
```

5.3.8.5 Поиск индексов элементов, удовлетворяющих условию

- Расширение a.Indices(T -> **boolean**) возвращает последовательность индексов тех элементов массива a, для которых будет истинным указанное в качестве аргумента лямбда-выражение. Если таких элементов не найдено, возвращается пустая последовательность. При необходимости получить индексы всех элементов массива можно указать просто a.Indices.
- Расширение a.Indices((T, i) -> **boolean**) возвращает последовательность индексов тех элементов массива a, для которых будет истинным указанное в качестве аргумента лямбда-выражение, учитывающее эти индексы i. Если таких элементов не найдено, возвращается пустая последовательность.

```
begin // p05070
  var a := ArrRandom(10, 0, 9);
  a.Println;
  a.Indices(q -> q.InRange(4, 7)).Println;
  a.Indices((p, i) -> (p >= 5) and i.IsOdd).Println
end.
```

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

В приведенном примере сначала отыскиваются индексы элементов со значениями в интервале [4;7], а затем нечетные индексы элементов со значением, не меньшим пяти.

5.3.8.6 Поиск индекса заданного элемента (.IndexOf)

- Расширение a.IndexOf(x) возвращает индекс первого из элементов массива a, равного x или -1, если такой элемент не найден.
- Расширение a.IndexOf(x, k) возвращает индекс первого из элементов массива a, равного x, просматривая элементы начиная с имеющего индекс k. Если такой элемент не будет найден, возвращается -1.
- Расширение a.LastIndexOf(x) возвращает индекс последнего из элементов массива a, равного x или -1, если такой элемент не найден.
- Расширение a.LastIndexOf(x, k) возвращает индекс последнего из элементов массива a, равного x, просматривая элементы от начала и не дальше элемента, имеющего индекс k. Если такой элемент не будет найден, возвращается -1.

```
begin // p05071
  var a := ArrRandom(20, 0, 9);
  a.Println;
  a.IndexOf(5, 7).Println;
  a.LastIndexOf(8, 3).Println
end.
```

```
4 9 1 1 2 9 4 4 0 3 2 1 6 5 8 0 7 4 2 0
13
-1
```

5.3.9 Продвинутые операции с массивами

В PascalABC.NET включены несколько операций с массивами, реализующих сложные выборки и группировки. Под группировкой понимается формирование некоторых структур, включающих исходные элементы, объединенные по определенному признаку, например, идущие подряд с одинаковым значением.

- Расширение a.AdjacentFind возвращает индекс первого из двух соседних элементов, значения которых одинаковы или -1, если таких элементов нет.
- Расширение a.AdjacentFind(k) возвращает индекс первого из двух соседних элементов, значения которых одинаковы, начиная поиск с элемента, имеющего индекс k, или -1, если таких элементов нет.
- Расширение a.AdjacentGroup возвращает последовательность из массивов, содержащих элементы с одинаковым значением.

```
begin // p05072
  var a := ArrRandom(15, 1, 5);
  a.Println;
  a.AdjacentGroup.Println
end.
```

```
1 2 4 4 1 1 3 5 5 5 2 2 3 2 5
[1] [2] [4,4] [1,1] [3] [5,5,5] [2,2] [3] [2] [5]
```

Вот так можно решить задачу о нахождении количества элементов в самой длинной подпоследовательности идущих подряд элементов с одинаковым значением:

```
begin // p05073
  var a := ArrRandom(15, 1, 5);
  a.Println;
  a.AdjacentGroup.Select(s -> s.Count).Max.Println
end.
```

Не нужно количество, нужна сама подпоследовательность? Это тоже не проблема:

```
begin // p05074
  var a := ArrRandom(15, 1, 5);
  a.Println;
  a.AdjacentGroup.MaxBy(s -> s.Length).Println
end.
```

```
1 3 1 5 5 5 1 4 4 2 2 1 4 3 5
5 5 5
```

Как это все работает? Рассмотрим по шагам на приведенном примере. Вначале a.AdjacentGroup сформирует последовательность из массивов

```
[1] [3] [1] [5,5,5] [1] [4,4] [2,2] [1] [4] [3] [5]
```

Затем MaxBy(s -> s.Length) выберет элементы последовательности, для которых указанное лямбда-выражение имеет максимальное значение. В нем s – элемент последовательности (массив), поэтому s.Length – его длина. Остается лишь вывести найденные эти элементы.

Усложним задачу: нужно получить и последовательность, и ее длину. Можно, конечно, сначала найти последовательность, а затем определить ее длину. Но можно сделать двухэлементный кортеж, объединив в нем и последовательность в виде массива, и длину этого массива.

```
begin // p05075
  var a := Arr(1, 3, 1, 5, 5, 5, 1, 4, 4, 2, 2, 1, 4, 3, 5);
  a.Println;
  a.AdjacentGroup.Select(s -> (s, s.Count)).MaxBy(p -> p[1]).Println
end.
```

```
1 3 1 5 5 5 1 4 4 2 2 1 4 3 5
([5,5,5],3)
```

- Расширение a.GroupBy(T -> T) преобразует массива a в последовательность, каждый элемент которой является объектом, созданным на основе элементов исходного массива, имеющих одинаковые значения. Такой объект содержит, в частности, поле ключа key, хранящее значение элемента, а расширение .Count возвращает количество найденных элементов со значением, равным ключу.

Найдем, сколько раз встречается в массиве каждый элемент. Такая задача относится к задачам построения ***частотного словаря***. Как будет показано в дальнейшем,

частотный словарь Dictionary имеется в коллекции Microsoft .NET Framework, но сейчас мы обойдемся без него.

```
begin // p05076
  var a := Arr(1, 3, 1, 5, 5, 5, 1, 4, 4, 2, 2, 1, 4, 3, 5);
  a.Println;
  a.GroupBy(t -> t).Select(o -> (o.Key, o.Count))
      .OrderBy(p -> p[0]).Println
end.
```

```
1 3 1 5 5 5 1 4 4 2 2 1 4 3 5
(1,4) (2,2) (3,2) (4,3) (5,4)
```

Проекция Select строит двухэлементный кортеж из ключа и количества элементов, а расширение .OrderBy сортирует последовательность полученных кортежей по возрастанию ключей (так удобнее искать нужный ключ). Мы видим, что единица встретилась в массиве 4 раза, двойка – два раза и т.д.

Отметим, что расширение .GroupBy работает не только для массива, но и для других видов последовательностей.

5.3.10 Ключевое слово params в подпрограммах

В части 4 был приведен без объяснений пример с функцией, содержащей в заголовке ключевое слово **params**.

```
function Min(params a: array of real) := a.Min; // p05077

begin
  Min(3.0, 4.9, 2.7, -5.1, 9.4, -2.0, 0.0, -8.2, 16.5).Println // -8.2
end.
```

За **params** всегда должно следовать описание динамического массива нужного типа, элементы которого будут переданы по значению. Если подпрограмма имеет несколько параметров, то описатель **params** может быть указан только для последнего из них; при этом не допускается указывать значение параметра по умолчанию.

Зачем указывать **params**, если массив можно передать и без этого? Если фактическим параметром будет только массив, наличие **params** действительно никакой роли не играет. Но, кроме массива, **params** позволяет передать в подпрограмму набор параметров, перечисленных через запятую, т.е. реализует обращение к подпрограмме с переменным числом параметров. Именно такой вызов функции Min и приведен в примере.

В описании функции Min указано, что элементы массива имеют тип **real**. Если мы вместо массива будем передавать список параметров, как это указано в примере, в этом списке могут быть целочисленные литералы и будет выполнено автоприведение типа. Min(3, 4.9, 2.7, -5.1, 9.4, -2, 0, -8.2, 16.5) – вполне корректный вызов. Но если предварительно сформировать их этих значений массив, например, с помо-

щью Arr( ), то все литералы должны будут изображать числа типа real. Еще одна причина использовать именно **params**.

5.4 Статические массивы array [] of T

Статические массивы – наследие базового Паскаля. В PascalABC.NET они не имеют ничего общего с последовательностями, поэтому не следует пытаться применять к ним методы и расширения.

Статический массив имеет размерный тип и память под него распределяется на этапе компиляции программы. Одновременно может быть выполнена инициализация элементов массива. Это означает, что границы индексов статического массива неизменны и должны быть указаны в программе константами или выражениями, содержащими только константы.

Метафора статического массива состоит в том, что под его элементы выделяется некоторая непрерывная область памяти, а поскольку каждый элемент массива занимает строго фиксированный объем памяти k байт, то для доступа к элементу с порядковым номером i нужно сместиться от начала области на k(i-1) байт. Такой алгоритм призван обеспечить высокоскоростной (по сравнению с прочими) доступ к произвольному элементу массива. В ряде языков программирования индексы массивов строятся от нуля. В таком случае доступ по индексу массива i обеспечивается смещением на k×i байт.

Статические массивы также могут быть одномерными и многомерными. В этой части мы рассматриваем одномерные массивы.

> К статическим массивам нельзя обращаться как к объектам (использовать точечную нотацию) ! Их также нельзя использовать при программировании в функциональной парадигме. Исключение составляют расширение .Count, возвращающее длину массива, и .Println, позволяющее вывести все элементы массива.

5.4.1 Создание статических массивов

Статический массив описывается в виде

```
var ИмяМассива: array[m..n] of Тип; или
var ИмяМассива: array[t] of Тип;
```

Конструкция вида m..n описывает минимальное и максимальное значение, которое может принимать индекс массива, причем допускаются и отрицательные значения. Эта конструкция задается константой порядкового типа. Количество элементов в массиве можно вычислить по формуле n-m+1. Объявленный таким образом массив создается компилятором и под него отводится необходимое место в памяти в соответствии с типом элементов, поэтому для статических массивов описание и создание массива не разделяются.

```
var a: array[0..12] of byte;
var b, c: array[-5..8] of real;
```

Массивы также могут быть описаны в разделе описаний (см. часть 11).

Значение нижней и верхней границы индекса массива a можно получить при помощи функций Low(a) и High(a) соответственно.

Описание массива можно совместить с инициализацией его элементов. Как и для динамических массивов, поскольку тип массива задан, можно при инициализации массива вещественного типа указывать в списке значения целочисленных типов.

```
begin // p05078
  var a: array[3..6] of integer := (1, 2, 3, 4);
  Println(a);
  var b: array[0..2] of real := (1.2, 5, -3.05); // 5 - целочисленная
  Println(b);
  var c: array [1..4] of real := (10, 11, 12, 13); // все целочисленные
  Println(c)
end.
```

```
[1,2,3,4]
[1.2,5,-3.05]
[10,11,12,13]
```

5.4.2 Работа со статическими массивами

Казалось бы, запрет на ООП и функциональную парадигму должен поставить статические массивы в положение изгоев. Но не все так печально: ведь у нас есть магическое расширение .Print, возвращающее последовательность, – это прекрасный способ вдохнуть жизнь в статические массивы. Ну а с последовательностями вы уже умеете работать и даже превращать их в динамические массивы посредством расширения .ToArray. С другой стороны, если мы планируем работать с последовательностями и динамическими массивами, зачем нам объявлять массив, как статический? Вот пример: нам может понадобиться получить массив из написанной когда-то в базовом Паскале подпрограммы и дальше работать с ним. Почему бы после получения такой массив не преобразовать? Ну а при случае, конечно, старый код надо будет переписать.

Несмотря на свою относительную новизну в Паскале, оператор цикла foreach умеет перебирать элементы статических массивов. К вашим услугам также циклы for, while и repeat.

Поскольку статические массивы относятся к размерному типу данных, их можно присваивать друг другу с помощью обычного оператора присваивания: это приводит к копированию всех элементов.

Статические массивы можно передавать в подпрограммы по значению или по ссылке. При передаче по значению осуществляется прямое копирование всех элементов массива. Это существенно расходует память и увеличивает время рабо-

ты программы, так что статический массив лучше всегда передавать по ссылке, указывая **var**, если элементы массива нужно менять и **const** в противном случае.

```
// p05079
type
  Arr = array [2..10] of integer;

procedure Prt(const a: Arr);
begin
  for var i := Low(a) to High(a) do
    (a[i] ** 3).Print
end;

begin
  var a: Arr := (1, 3, 5, 7, 9, 2, 4, 6, 8);
  Prt(a)
end.
```

```
1 27 125 343 729 8 64 216 512
```

Здесь раздел типов Type обязателен, поскольку в заголовке описания подпрограммы должен быть указан известный компилятору тип. В разделе типов описан нужный тип Arr, который затем использован в процедуре Prt.

Когда могут быть полезны статические массивы? В дальнейшем, при рассмотрении работы с файлами будет показано, что одна из их разновидностей (так называемые типизированные файлы) позволяет работать только со статическими массивами. В прочих случаях такие массивы использовать смысла нет.

В программах, написанных на базовом Паскале, встречаются довольно экзотические случаи использования статических массивов, индексы которых определяются не на основе целочисленных констант. Это возможно благодаря тому, что индексы принадлежат к порядковому типу, который может быть также диапазонным (например, 'a'.. 'z') или перечислимым (например, Красный, Синий, Зеленый). Такой код полноценно поддерживается, но в новых разработках его лучше не применять.

Рассмотрим задачу подсчета количества вхождений каждой цифры в строку, содержащую сорок знаков после запятой числа π.

Вариант решения на диалекте Free Pascal:

```
// p05080
var
  a: array['0'..'9'] of integer;
  s: string;
  i: char;
  j, n: integer;

begin
  s := 'Пи = 3,1415926535 8979323846 2643383279 5028841972';
  for i := '0' to '9' do
    a[i] := 0;
  n := Length(s);
  for j := 1 to n do
    if s[j] in ['0'..'9'] then Inc(a[s[j]]);
  for i := '0' to '9' do
    Write('(', i, ')-', a[i], ' ')
end.
```

```
(0)-1 (1)-3 (2)-6 (3)-7 (4)-4 (5)-4 (6)-3 (7)-3 (8)-5 (9)-5
```

Вариант решения на PascalABC.NET

```
begin // p05081
  var s := 'Пи = 3,1415926535 8979323846 2643383279 5028841972';
  s.MatchValues('\d').GroupBy(t -> t).Select(t -> (t.Key, t.Count))
    .OrderBy(t -> t[0]).ForEach(t -> begin Print($'({t[0]})-{t[1]}') end)
end.
```

```
(0)-1 (1)-3 (2)-6 (3)-7 (4)-4 (5)-4 (6)-3 (7)-3 (8)-5 (9)-5
```

Мы получили короткий код и обошлись без статических массивов. Если смущает запись после Print – это интерполированная строка и о них будет сказано в конце следующей части. Можно было написать традиционный Write, но код был бы менее нагляден и на десяток символов длиннее.

5.5 Для самостоятельного решения

T5.1. Написать функцию, возвращающую последовательность n случайных вещественных чисел на интервале [a;b], округленных до $m \geqslant 0$ знаков после запятой.

T5.2. Определить, сколько раз в последовательности n целых чисел встречается значение m, введенное с клавиатуры.

T5.3. В последовательности n целых чисел из интервала [-99; 99] определить значение среднего геометрического чисел, попадающих в интервал [15; 60]. Среднее геометрическое m чисел равно корню степени m из произведения этих чисел.

T5.4. В последовательности n натуральных чисел, не превышающих 10^9, найти значение элемента, имеющего наибольший простой делитель. Вывести значение элемента и значение делителя.

T5.5. Среди n точек, заданных парами случайных целочисленных координат, отыскать две точки, наиболее далеко отстоящие друг от друга. Если таких точек несколько, выбрать любую пару. Значения координат лежат в интервале [-99; 99]. Формулу для определения расстояния между двумя точками можно взять из задачи Т4.1.

T5.6. Задача из анекдота, опубликованного на Интернет-ресурсе «Десять букв» 27.03.2010: «В магазин пришло бесконечное множество математиков. Первый попросил килограмм сахара, второй – полкило, третий – четверть килограмма… – Так! - прервал их продавец, – Забирайте свои два килограмма и проваливайте.»

Найдите сумму бесконечной последовательности вида

$$1+\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{2^n},\ \mathrm{n}=0,1,\ldots\infty$$

Примечание: Можно проводить суммирование до тех пор, пока очередной член последовательности не станет меньше либо равен некоторой разумно малой (с точки зрения арифметики чисел с плавающей точкой) величине, например, 10^{-15}.

T5.7. Метод Монте-Карло предполагает использование последовательности n случайных чисел с последующим вычислением отношения количества элементов последовательности, удовлетворяющих некоторому условию, к n. Точность метода растет с увеличением n, а слишком малые значения n задавать бессмысленно. Вычислите методом Монте-Карло значение числа π, задав n = 1 000 000.

Поскольку площадь круга радиуса r=1 равна π, а площадь описанного вокруг него квадрата равна 4 (удостоверьтесь), можно сформировать последовательность точек со случайными вещественными координатами (x, y) на интервале [-1; 1] и подсчитать, сколько из них будет удовлетворять условию $x^2+y^2\leqslant 1$. Умноженное на 4 отношение этого количества к 1 000 000 даст приближенное искомое значение.

Часть 6

Символы и строки

Три самых дорогостоящих ошибки всех времен были вызваны изменением одного символа в ранее корректных программах.
Стивен Макконнелл,
автор книг по программному обеспечению

Не путайте: Машины обрабатывают числа, а не символы. Мы измеряем свое понимание (и контроль) степенью арифметизации деятельности.

Строка – это застывшая структура данных, и повсюду, куда она передается, происходит значительное дублирование процесса. Это идеальное средство для сокрытия информации.
Алан Перлис, американский ученый

До сих пор речь шла о числах, изредка – о логическом типе данных (**boolean**). Несколько раз в тексте упоминались какие-то таинственные «строки» и «символы», которые будут рассмотрены в дальнейшем. Это время наступило.

Текст любой (и этой тоже) книги набран при помощи букв, цифр и различных знаков, часть из которых даже не отображается (они называются служебными). В совокупности всё это составляет некий ***алфавит***. Алфавит рассматривается как упорядоченный ***набор символов***.

Текст программы тоже состоит из символов, составляющих алфавит языка программирования. Символы складываются в более крупные конструкции, образуя ***строки символов***. Чтобы отделить в тексте одну строку от другой, договариваются, какие символы будут служить разделителями строк. Обычно для этой цели выбираются один или два служебных символа. Исторически сложилось, что признаком окончания строки являлся переход на новую строчку при печати. В зависимости от типа печатающего устройства, символ завершения строки мог иметь шестнадцатеричный код \$0D (13_{10}) – «перевод строки», \$0A (10_{10}) – «возврат каретки», либо использовалась их последовательность \$0D0A (и сейчас используется, к примеру, в Windows).

За каждой клавишей на клавиатуре компьютера закреплены один или более символов. При нажатии клавиши в компьютер посылается соответствующий код и этому коду сопоставляется некоторый символ. (На самом деле клавиатура посылает не один код, а два – один при нажатии клавиши, второй – при ее отпускании. Это позволяет при зажатой клавише генерировать повтор символов.) В разных странах клавиатура одна и та же, но часть значков, нарисованных на ней, в каждой стране своя. Ничего удивительного, ведь алфавит любого языка всегда содержит какие-то, характерные для этого языка символы. Но часть символов никогда не меняется, за исключением очень малого количества языков, поэтому было решено таким символам присвоить коды от нуля до 127 и назвать их ***интернациональными симво-***

лами. Остальные символы в каждом языке, имея одинаковые коды, могут различаться, образуя символы национальных алфавитов.

Интернациональные символы и символы национальных алфавитов помещаются в так называемые ***кодовые таблицы***. Если код символа однобайтный, в такой таблице можно закодировать $2^8 = 256$ различных символов. Получается, что ровно половина кодов (0 .. 127) отведена для интернациональных символов, а другая половина (128 .. 255) – для символов национального алфавита. Говорят, что первая (или нижняя) половина у всех кодовых таблиц одинакова, а вторая (верхняя) отведена под ***локализацию***. В данном случае локализация – это предоставление возможности операционной системе работать не на международном (английском) языке, а на языке конкретной (локальной) страны. Для доступа к символам того или иного национального алфавита нужно всего лишь сменить кодовую таблицу.

В настоящее время однобайтные кодовые таблицы (KOI-8, ASCII) устарели. В самом деле, если требуется одновременно отображать символы, которые принадлежат разным кодовым таблицам, то нужно между этими таблицами постоянно переключаться. Но в универсальных языках программирования понятия «кодовая таблица» не существует. Приходится для выбора кодовой таблицы обращаться из программы к функциям операционной системы, что очень неудобно. Этот недостаток устраняется при переходе на двухбайтные коды символов.

Двухбайтный код позволяет адресоваться к 2^{16}=65536 различным символам. Этот код получил название Unicode («Юникод»). Но и тут есть разновидности. Существуют смешанные кодировки, в которых интернациональные символы кодируются одним байтом, а прочие – двумя. Современные версии Windows работают с Unicode.

Но давайте все же вернемся к PascalABC.NET.

6.1 Символьный тип данных

Данные символьного типа имеют тип **char** и занимают **в памяти** 2 байта. Используется кодировка Unicode. В тексте программы символьная константа (литерал) всегда заключается в одинарные кавычки. Если надо записать сам символ одинарной кавычки, то кавычка удваивается: ' ' ' '.

Нужно сразу уяснить, что в программе встречаются понятия как символа, так и его внутреннего кода. В базовом Паскале использовалась однобайтная кодировка, а кодовая таблица выбиралась в зависимости от локализации операционной системы. Так, Turbo/Borland Pascal и Free Pascal работали с кириллицей («русскими буквами») преимущественно в кодовой таблице CP866, Borland Delphi, работая в среде Windows, мог использовать таблицу CP1251. PascalABC.NET использует кодировку Unicode.

Новички в программировании часто смешивают понятия символа, обозначающего цифру и имеющего тип **char**, и самой этой цифры, которая может быть неотрицательным однозначным числом одного из целочисленных типов. Важно понимать, что символ – это «картинка», рисунок, изображение числа, и за символом в программе кроется не значение изображенной цифры, а код этого символа, взятый из кодовой таблицы.

Как и целочисленные типы, символьный тип относится к так называемому ***порядковому*** типу данных. Вероятно, тут имеет значение тот факт, что порядок следования символов друг за другом определяется порядком следования их внутренних кодов в кодовой таблице. Благодаря такой классификации в большинстве операторов, методов и расширений PascalABC.NET на месте, где можно указать данное числового типа, разрешено указывать и символьные данные. Это сделает изучение данной части книги достаточно комфортным, потому что материал в основе уже знаком.

Данные символьного типа, как и любого другого, описываются с указанием ключевого слова **var**:

```
var c1, p135, rz: char; // три переменные
var Символ1: char; // одна переменная
var s: sequence of char; // последовательность символов;
var ar: array of char; // динамический массив символов
```

Описание можно соединить с инициализацией:

```
var a: char := 'a'; // тип указан явно
var b:= 'b'; // автовыведение типа
var kt:= ('A', 'B', 'C'); // кортеж из трех символов
```

Далее по тексту этой части книги при описании конструкций языка как символы, так и переменные типа **char** будут обозначаться «c», если прямо не указано иное.

Выводить символы можно любыми средствами, которые использовались для вывода других типов данных: Print и Println, Write и Writeln, а также при помощи расширений .Print и .Println. По умолчанию инициализация данных типа **char** производится двоичными нулями, что может создать проблемы при выводе.

Получить десятичный код символа можно несколькими способами:
- Ord(c) – код символа c в Unicode (тип **word**);
- **char**.Code – то же самое, точечная нотация;
- OrdAnsi(c) – десятичный код символа в однобайтной кодировке Windows (тип **byte**).

Произвести обратную операцию – получить символ по его внутреннему коду, можно тоже несколькими способами:

- Chr(**word**) – символ с указанным кодом Unicode;
- **#word** – символ с указанным кодом Unicode; принимает только литерал;
- ChrAnsi(**byte**) – символ с кодом в однобайтной кодировке Windows.

Приведенная ниже программа выводит коды букв кириллицы из таблицы Unicode. Используется знание о том, что кириллица в этой таблице идет подряд, исключая буквы Ё и ё. Генератор Range строит последовательность из 64 символов кириллицы (вот оно – первое проявление порядкового типа данных: Range работает и с символами).

```
begin
  Range('А', 'я').Select(c -> c.Code).Println
end.
```

```
1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055
1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071
1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087
1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103
```

Эту же задачу можно решить и более традиционно, с помощью цикла. Вспомним, что параметр цикла for может быть любого порядкового типа. Следовательно, и типа **char**.

```
begin
  for var c := 'А' to 'я' do
    Print(Ord(c))
end.
```

Соединяясь друг с другом, символы образуют ***строки*** (см. 6.2). Пока мы будем считать строки последовательностью символов. А еще лучше – динамическим массивом с индексами от единицы. Потому что, написав для строки s, например s[5], мы получим пятый от начала строки символ. Для соединения символов служат знаки операции «+» и «*». Знак + сцепляет символы между собой, знак * позволяет размножить символ нужное число раз путем умножения его на число.

```
begin // p06001
  Println((2 * ('м' + 'а') + ' ') * 5)
end.
```

```
мама мама мама мама мама
```

А вот еще приятная неожиданность: если с символом сложить число, оно будет автоматически преобразовано к строке символов и эта строка сцепится с указанным в выражении символом.

```
begin
  ('$' + 132.5).Println
end.
```

```
$132.5
```

6.1.1 Анализ символов на принадлежность к группе

Несмотря на то, что компьютеры возникли из-за потребности людей к счету, их работа во многом опирается на обработку символов. Мы нажали на клавиатуре клавишу с цифрой «5» – что получил компьютер? Правильно, числовой код нажатой клавиши. Особая программа – драйвер клавиатуры – обработала этот код и превратила его в другой – код символа ′5′ в кодовой таблице. А где число 5? Его нет. Компьютер не получает набираемые с клавиатуры числа – он получает коды соответствующих символов. Чтобы начать считать, компьютер сначала должен сформировать числа, участвующие в счете. Но пока что он получает лишь коды символов – цифр и иных знаков, составляющих число . Поэтому дальше подключается еще одна программа, так называемый ***парсер***, которая выделяет из кодов соседних символов те, которые могут дать изображение числа. Парсер преобразует коды в само число, в соответствии с его типом формирует внутреннее представление этого числа и только потом помещает в память на отведенное место. Вот сколько всего потребовалось написать, чтобы дать лишь примитивное представление о том, как компьютер обрабатывает нажатие клавиши. Получается, что без обработки символов нет и быть не может вычислений.

Всегда ли нам нужно превращать символы в числа? Нет, лишь тогда, когда числа участвуют в вычислениях. Например, указанный в подзаголовке номер 6.1.1 вовсе не предполагает проведения над ним каких-то вычислительных манипуляций. Следовательно, имеются ситуации, когда символ должен оставаться символом. Например, пара символов «а» и «с», - это слово «ас» или число AC_{16}? Тут все зависит от того, какой текст их окружает (говорят, что интерпретация последовательности символов зависит от окружения - ***контекста***).

Итак, понятно, что нужно уметь различать символы. Различает их, конечно же, программа. Которую, в свою очередь, пишет программист. Т.е. и вы в том числе. Различая символы, мы относим их к той или иной группе – буквы, цифры, знаки препинания, скобки, пробелы и переносы строк и т.п. А помогают нам в этом имеющиеся в языке функции, процедуры, методы и расширения.

До сих пор мы не обращали особого внимания на понятие ***статических*** (классовых) методов. Это потому, что время серьезно разобраться с классами пока не подошло. При вызове метода используется точечная нотация, в которой сначала указывается имя объекта, а потом, через точку, записывается вызываемый для объекта метод. Но в случае статических методов при вызове надо указывать не имя объекта, а класс (для символов - **char**). Имя объекта при этом записывается в круглых скобках, как обычный параметр. Поскольку ранее мы договорились, что **char** означает переменную или непосредственно символ, при упоминании классовых методов будем писать с вместо char. Например, в формате записи **char**.IsLetter(c), **char** – это имя класса, а с означает переменную или непосредственно символ.

> Если явно не указано иное, подразумевается, что коды символов имеют двухбайтную кодировку (word) и принадлежат кодовой таблице Unicode

6.1.1.1 Является ли символ буквой?

- c.IsLetter – расширение возвращает True, если символ с принадлежит к группе букв и False в противном случае.
- **char**.IsLetter(c) – статический метод класса, делающий то же самое.

Буквами считаются латинские и кириллические буквы в обоих регистрах, а также буквы иных алфавитов, например, греческого. Именно эти буквы PascalABC.NET позволяет использовать в именах объектов программы. Давайте попытаемся оценить их количество.

```
begin
  Range(0, word.MaxValue).Where(n-> Chr(n).IsLetter).Count.Println // 48718
end.
```

Немало, оказывается! 48.7 тысяч. Жаль, большая часть из них – иероглифы. Но давайте разберемся, как работает эта «однострочная» программа.

word.MaxValue возвращает максимальное значение, которое может поместиться в переменную типа **word**. Range строит последовательность от нуля до полученного значения, что позволяет получить все коды, теоретически помещающиеся в кодовую таблицу. Последовательность фильтруется методом Where, использующим выражение Chr(n).IsLetter, которое строит символ с очередным кодом и проверяет, является ли он буквой. Метод .Count возвращает количество элементов последовательности, прошедшее фильтр.

6.1.1.2 Является ли символ цифрой?

- c.IsDigit – расширение возвращает True, если символ с принадлежит к группе цифр (0, 1, … 9) и False в противном случае.
- char.IsDigit(c) – статический метод класса, делающий то же самое.

6.1.1.3 Является ли символ буквой или цифрой?

char.IsLetterOrDigit(c) – статический метод класса возвращает True, если символ с принадлежит к группе букв или цифр и False в противном случае.

6.1.1.4 Является ли символ пробельным?

char.IsWhiteSpace(c) – статический метод класса возвращает True, если символ с принадлежит к группе пробельных символов и False в противном случае.

Пробельные символы – термин, пришедший из типографской практики. Пробел – это пустое место, интервал между символами. Но приходилось ли вам когда-нибудь задумываться, что длина пробела бывает различной? Пробельными в типографском и издательском деле называют непечатаемые (и неотображаемые) символы. Давайте посмотрим, какие коды им принадлежат.

```
begin
  Range(0, word.MaxValue).Where(n -> char.IsWhiteSpace(Chr(n))).Println
end.
```

```
9 10 11 12 13 32 133 160 5760 8192 8193 8194 8195 8196 8197 8198 8199 8200 8201
8202 8232 8233 8239 8287 12288
```

Символы с кодами 9 .. 13 относятся к так называемым управляющим символам – раньше они управляли внешними устройствами, имеющими печатающую каретку, например такими, как механический принтер. Символ с кодом 32 – это привычный пробел. Коду 160 соответствует – неразрывный пробел, запрещающий в его позиции менять строку. Символы с кодом 8192 и последующие используются в программах верстки текстов – именно они обеспечивают пробелы различной длины.

6.1.1.5 Является ли символ знаком препинания?

char.IsPunctuation(c) – статический метод класса возвращает True, если символ с принадлежит к группе знаков пунктуации (разделителям) и False в противном случае. Познакомиться с этими символами нам также поможет программа.

```
begin
  Range(0, 1300).Select(n -> Chr(n))
      .Where(c -> char.IsPunctuation(c)).Println
end.
```

```
!"#%&'()*,-./:;?@[\]_{}¡«·»¿;·
```

Здесь просмотр ограничен символом с кодом 1300, чтобы не заполнять вывод знаками, которые в России никогда не используются и скорее всего, корректно не будут отображены при попытке их напечатать. Нам достаточно убедиться, что среди выведенных символов имеются все употребляемые знаки препинания.

6.1.1.6 Принадлежит ли буква к верхнему регистру?

- c.IsUpper – расширение возвращает True, если символ с принадлежит к буквенным символам верхнего регистра (прописным) и False в противном случае.
- char.IsUpper(c) – статический метод класса, делающий то же самое.

В качестве маленькой премии отметьте, что если символ не является буквой, возвращается False, какому бы регистру он не принадлежал. Это позволяет не писать лишних условий вида **if** c.IsLetter **then if** c.IsUpper **then** ...

6.1.1.7 Принадлежит ли буква к нижнему регистру?

- c.IsLower – расширение возвращает True, если символ с принадлежит к буквенным символам нижнего регистра (строчным) и False в противном случае.
- char.IsLower(c) – статический метод класса, делающий то же самое.

Здесь также, если символ не является буквой, возвращается False, какому бы регистру он не принадлежал.

6.1.1.8 Располагается ли символ в указанном интервале?

- c.InRange(c1, c2) – расширение возвращает True, если код символа с принадлежит интервалу, в котором находятся коды символов, начиная от c1 и заканчивая c2. В противном случае возвращается False. Например, буква L находится в интервале от 0 до R, но не находится в интервале от Д до Я.

6.1.2 Операции преобразования символов

К операциям преобразования символов отнесены: смена регистра буквенных символов, переход к следующим и предыдущим символам (в порядке следования их кодов в таблице), а также преобразование символа из группы «цифра» к соответствующему этому символу однозначному числу (см. 6.1.1.2).

6.1.2.1 Приведение буквенного символа к верхнему регистру

- c.ToUpper – расширение возвращает буквенный символ c, приведенный к верхнему регистру, если он принадлежит к нижнему регистру. В противном случае символ возвращается без изменения.
- UpCase(c) – функция, делающая то же самое
- UpperCase(c) – функция, синоним UpCase.
- char.ToUpper(c) – статический метод класса, переводящий символ c на верхний регистр, если он принадлежит к нижнему регистру. В противном случае никаких действий не выполняется.

6.1.2.2 Приведение буквенного символа к нижнему регистру

- c.ToLower – расширение возвращает буквенный символ c, приведенный к нижнему регистру, если он принадлежит к верхнему регистру. В противном случае символ возвращается без изменения.
- LowCase(c) – функция, делающая то же самое
- LowerCase(c) – функция, синоним LowCase.
- char.ToLower(c) – статический метод класса, переводящий символ c на нижний регистр, если он принадлежит к верхнему регистру. В противном случае никаких действий не выполняется.

6.1.2.3 Преобразование символа в число

c.ToDigit – расширение возвращает буквенный символ c, преобразованный к изображаемому им целому неотрицательному однозначному числу типа **integer**.

Если символ не является цифрой, т.е. расширение .IsDigit возвращает для него **False**, при выполнении программы будет выдано сообщение «Ошибка времени выполнения: not a Digit» и программа завершится аварийно.

6.1.2.4 Символ, предшествующий указанному

- c.Pred – расширение возвращает буквенный символ, код которого предшествует коду символа c.
- Pred(c) – функция, делающая то же самое.

А какой код предшествует символу с кодом 0? Размышляя логически – никакой, потому что этим символом начинается кодовая таблица. Тогда что вернет вызов Chr(0).Pred? Увы, при выполнении будет сгенерировано исключение с сообщением «Ошибка времени выполнения: Значение было недопустимо малым или недопустимо большим для знака».

6.1.2.5 Символ, следующий за указанным

- c.Succ – расширение возвращает буквенный символ, код которого следует за кодом символа c.
- Succ(c) – функция, делающая то же самое.

Какой код следует за символом с кодом 65535 – последним кодом в таблице символов? Дальнейшие рассуждения схожи с приведенными в предыдущем подразделе. Да и результат попыток получить код такого символа будет таким же.

6.1.2.6 Смещение по кодовой таблице на указанное число символов

- Dec(c) – процедура, заменяющая значение переменной, содержащей символ с на символ, предшествующий ему. Вспомните декремент – операцию Dec - это она же, только для символов.
- Dec(c, n) – процедура, заменяющая значение переменной, содержащей символ с символом, находящимся в кодовой таблице на n позиций раньше. Тоже декремент.
- Inc(c) – процедура, заменяющая значение переменной, содержащей символ с на символ, следующий за ним. Это инкремент.
- Inc(c, n) – процедура, заменяющая значение переменной, содержащей символ с символом, находящимся в кодовой таблице на n позиций дальше. И тоже это инкремент.

6.1.3 Ввод символов

В языке Паскаль ввод с клавиатуры всегда завершается нажатием клавиши Enter, а вводимые значения (кстати, состоящие из одного или более символов) разделяются пробелами или нажатием все той же клавиши Enter. Нажатие клавиши Enter в операционной системе Windows посылает комбинацию символов #13#10 («перевод строки», «возврат каретки»), а в Unix-подобных системах – только #10. Но это тоже символы! Если не принять мер, они будут восприняты, как вводимые данные.

Вводимые данные сначала сохраняются в некоторой области памяти (буфере), а затем анализируются. Каждый раз, когда нужна очередная порция данных, проис-

ходит обращение сначала к буферу, а если там данных недостаточно – тогда уже к клавиатуре.

За прием данных с клавиатуры отвечают оператор Read(Элемент1, Элемент2, ...) или его разновидность Readln(Элемент1, Элемент2, ...). Readln (от английских слов Read Line – чтение строки) не воспринимает финальное нажатие клавиши Enter, как данные, более того, он полностью очищает буфер после ввода, даже если были прочитаны не все символы.

При вводе числовых или логических данных наличие или отсутствие служебных символов #13#10 в буфере непринципиально. Ведь в написании данных этих типов нет служебных символов, поэтому для парсера они лишь граница очередного введенного значения. Это позволяет использовать как Read, так и Readln. Но перед вводом символов буфер должен быть пуст, поэтому такому вводу должен предшествовать ввод данных с помощью Readln. Это надо понять или хотя бы очень хорошо запомнить.

```
begin // p06002
  var k := ReadInteger('Введите целое число:');
  Println('Вы ввели', k);
  var x := ReadlnReal('Введите вещественное число:');
  Println('Вы ввели', x);
  var c: char;
  Print('Введите звездочку:');
  Readln(c);
  Println('Вы ввели', c);
  Println(15 * '-');
  k := ReadInteger('Введите целое число:');
  Println('Вы ввели', k);
  x := ReadReal('Введите вещественное число:');
  Println('Вы ввели', x);
  Print('Введите звездочку:');
  Readln(c);
  Println('Вы ввели', c);
  Writeln('*** Работа завершена ***')
end.
```

```
Введите целое число: 256
Вы ввели 256
Введите вещественное число: -24.78
Вы ввели -24.78
Введите звездочку: *
Вы ввели *
---------------
Введите целое число: 34
Вы ввели 34
Введите вещественное число: 4.93
Вы ввели 4.93
Введите звездочку: Вы ввели

*** Работа завершена ***
```

Фигурной скобкой в приведенном диалоге, возникающем при работе с программой, отмечен «странный фрагмент». Конечно, эта странность заложена в программу с целью продемонстрировать «А что будет, если… ?». И это место в программе тоже помечено фигурной скобкой.

В первый раз звездочка была введена и успешно выведена. Во второй раз программа не делала паузы для ожидания ввода и не дала ничего ввести, а сразу вывела пустую строку. Не пробел, нет, именно целую пустую строку. Как это могло случиться, где место ошибки?

В первый раз вводу символа предшествовал оператор

```
var x := ReadlnReal('Введите вещественное число:');
```

Во второй раз выполнялся оператор

```
x := ReadReal('Введите вещественное число:');
```

Нашли различие? ReadlnReal и ReadReal. В первом случае использована разновидность «ln», очищающая буфер ввода, во втором случае буфер не очищается перед вводом символа. Именно поэтому во втором случае не делается обращение к клавиатуре, а символьное данное считывается из буфера. А там что? Там символ смены строки. Он и выводится: мы получили пустую строку.

> Ввод, предшествующий вводу символьных данных, должен обязательно выполняться по команде с очисткой буфера (Readln, ReadlnInteger и им подобных). Вводимые символьные данные **нельзя разделять** пробелами или чем-либо еще: любое нажатие клавиши воспринимается, как символ.

Имеются функции, осуществляющие ввод символьных данных с приглашением:

```
ReadlnChar('ТекстПриглашения'); // ввод одного символа
ReadlnChar2('ТекстПриглашения'); // ввод двух символов
ReadlnChar3('ТекстПриглашения'); // ввод трех символов
```

Приглашение может быть опущено и тогда используется формат ReadlnChar, ReadlnChar2, ReadlnChar3.

Если очищать буфер ввода не нужно, используются разновидности без «ln»: ReadChar, ReadChar2, ReadChar3; приглашение ко вводу также может присутствовать.

Рассмотрим простой пример со вводом символов.

```
begin
  var(a, b, c) := ReadChar3('-->');
  Writeln(a, b, c, ' --> ', c, b, a)
end.
```

```
--> сон
сон --> нос
```

Здесь ввод используется лишь один раз, поэтому можно использовать как ReadChar3, так и или ReadlnChar3. В протоколе работы с программой видно, что все три символа вводились без пробелов. Для вывода использован оператор Writeln, чтобы символы не разделялись пробелами. Программа вывела принятые символы вначале в порядке их ввода, а затем в обратном порядке.

Еще одна программа предлагает ввести буквы русского алфавита и проверяет правильность порядка их следования. Ее можно использовать в качестве небольшого теста на знание алфавита. В программе используется порядок следования кириллических символов в кодовой таблице.

```
begin // p06003
  Println('Проверьте, насколько хорошо Вы знаете русский алфавит!');
  Println('Введите подряд русские буквы в алфавитном порядке.');
  Println('Регистр букв неважен; буква Ё учитывается');
  var mc := Range('A', 'E') + 'Ё' + Range('Ж', 'Я');
  foreach var c in mc do
   if ReadChar.ToUpper <> c then
    begin
      Println('Вы ошиблись!');
      exit // завершить работу программы
    end;
  Println('Молодец!')
end.
```

```
Проверьте, насколько хорошо Вы знаете русский алфавит!
Введите подряд русские буквы в алфавитном порядке.
Регистр букв неважен; буква Ё учитывается
абвгдеёжзийклмнопрстуфхцчшщъыьэюя
Молодец!
```

Обратите внимание, что в теле цикла использован ввод посредством ReadChar, а не ReadlnChar. Программа запрашивает ввод символов подряд, но, как уже не раз писалось, анализ введенных данных (чтение буфера ввода) программа начнет выполнять только после нажатия клавиши Enter. К этому моменту в буфере должны накопиться 33 введенных символа с #13#10 в конце. В программе имеется цикл посимвольного ввода и анализа. Если использовать ReadlnChar, то после чтения первого символа буфер будет очищен и остальные введенные символы пропадут. Программа снова запросит ввод и так будет делать 32 раза (в русском алфавите 33 буквы). Тот самый редкий случай, когда нужно именно Read, а не Readln.

6.1.4 Пример: нахождение суммы в символьном виде

$$S_k = \sum_{i=1}^{10^k} i,\ k \in \mathbb{N}$$

Современные компьютеры имеют высокое быстродействие, но это не освобождает программиста от необходимости знания математики и умения думать. Рассмотрим пример задачи на вычисление суммы натуральных чисел, количество которых равно 10^k.

Пусть нам требуется найти сумму для k=30. Если считать сумму «в лоб», последовательно складывая натуральные числа от 1 до 10^{30}, то современному компьютеру, выполняющему порядка 10^9 сложений за секунду, потребуется 10^{21} секунд, что примерно равно 31,7 триллионов лет. С учетом того, что оценочный возраст нашей Вселенной примерно в 2300 раз меньше, понятно, что бессмысленно затевать такой процесс.

В математике числовая последовательность 1, 2, 3, … n называется арифметической прогрессией. Не вдаваясь в подробности, используем формулу для расчета суммы членов данной прогрессии в нашей задаче.

$$S_n = \frac{2+(n-1)}{2} \times n = \frac{1}{2} n(n+1);$$

$$n = 10^k \rightarrow S(k) = \frac{1}{2} 10^k (10^k + 1) = \frac{1}{2} \cdot 10 \cdot 10^{k-1} (10^k + 1);$$

$$S(k) = 5 \cdot 10^{k-1} (10^k + 1);$$

При k=30 мы будем иметь дело с числами порядка 10^{60}, имеющими 60 цифр. По диапазону данных тут подойдет только BigInteger. Но пока мы с ними не умеем работать. Точнее, не умеем задавать для них значения, превышающие величину int64.MaxValue. Поэтому идею с вычислениями суммы таким способом пока придется отложить.

Заметим, что величина 10^k – это единица с k нулями, поэтому 10^k+1 будет числом вида 1000…001, содержащим k-1 ноль. Число $5\times10^{k-1}$ можно записать как пятерку с k-1 нулем. Тогда S(k) будет иметь вид 5000…05000…00, где после пятерки следует k-1 ноль. Например, для k=1 получаем S(1)=55, для k=2 – S(k)=5050 и т.д.

Вычисление свелось к выписыванию пятерок и нулей. А работать с символами мы уже умеем. Оформим решение функцией.

```
function S(k: integer) := 2 * ('5' + '0' * (k - 1)); // p06004

begin
  S(30).Println
end.
```

Вот так выглядит наш результат:

500000000000000000000000000005000000000000000000000000000000

Сотые доли секунды вместо триллионов лет: стоило перед программированием задания немного подумать, не так ли?

6.2 Строковый тип данных

Как уже упоминалось, символы складываются в более крупные единицы текста – строки символов. В программировании под термином «строка» понимают совсем

не то, что в других областях, например, в литературе или в издательском деле. Строка – это последовательность символов, принадлежащих некоторому алфавиту.

В PascalABC.NET имеются два типа строк. Первый тип – строки, унаследованные от языка Turbo Pascal (в языке Паскаль, каким его описал Н.Вирт, строкового типа вообще не было). Длина таких строк ограничена 255 символами. В PascalABC.NET они именуются «***короткими (размерными) строками***» и относятся размерному типу. При описании короткой строки необходимо в квадратных скобках после ключевого слова **string** указывать ее максимальную длину – количество символов, которое строка может иметь в программе. Короткие строки оставлены в языке лишь для совместимости со старыми программами и для работы с типизированными файлами (см. часть 9).

Второй тип строк – это современные строки, базирующиеся на строках Microsoft .NET Framework и их длина ограничена примерно двумя гигабайтами. При описании таких строк их длина не указывается. Это основной тип строк PascalABC.NET, они относятся к данным ссылочного типа, а их тип так и называется - «строки». Они, строго говоря, не являются строками .NET хотя бы уже потому, что их можно изменять и копировать прямым присваиванием, а нумерация символов в строках ведется от единицы.

Каждый символ строки в памяти хранится в двух байтах таблицы Unicode.

Строковые данные в PascalABC.NET описываются с ключевым словом **string**. Литерал, представляющий строку, заключается в одинарные кавычки. Если внутри строки встречается символ одинарной кавычки, он удваивается.

```
var st, МояСтрока, p18: string; // три строки
var Строчка: string; // одна строка
var sos: sequence of string; // последовательность строк
var ar: array of string; // динамический массив строк
var sh1: string[27]; // короткая строка, максимум 27 символов
```

Описание строк можно соединять с инициализацией:

```
var s1: string := 'Это строка'; // тип указан явно
var s2 := '*** И это строка ***'; // автовыведение типа
var kt := ('Это', 'тоже', 'строка'); // кортеж из трех строк
```

Строка может состоять из одного символа, и даже может иметь и нулевую длину, т.е. не содержать ни одного символа. В последнем случае она записывается парой одиночных кавычек, следующих друг за другом.

Если указано **var** d := 'R', какого типа будет переменная d? Типа **char**, конечно же, – тут произойдет автовыведение типа. Если нужен тип string, в данной ситуации придется указать его явно или использовать явное приведение.

```
var s1 := 'Я'; // символ типа char
var s2: string := 'Я'; // строка типа string с длиной 1 – указан тип
var s3 := string('Я'); // строка типа string с длиной 1 – явное приведение типа
```

Короткие строки имеют размерный тип, а прочие строки – ссылочный, но при этом они могут вести себя, как объекты размерного типа. Строки можно рассматривать как последовательности символов и как массивы символов, так что для них применима большая часть функций, методов и расширений, рассмотренная в части 5.

На первый взгляд, с обработкой строк в PascalABC.NET дела обстоят совсем непросто. Имеется более ста различных процедур, функций, методов и расширений, предназначенных для работы со строками. Кажется невероятным, что можно запомнить, какие из них и когда следует использовать. Причина такого обилия кроется в требовании совместимости PascalABC.NET с базовым Паскалем. В Turbo Pascal (Borland Pascal) имелся некоторый набор процедур и функций для работы с короткими строками. В Borland Delphi появились и «длинные строки», принеся с собой большое количество добавочных функций, процедур и методов. Строки .NET Framework и новые парадигмы программирования PascalABC.NET существенно пополнили арсенал средств работы со строками.

Рекомендуемые при написании новых программ функции, процедуры, методы и расширения для работы со строками ниже будут помечаться значком ⍟

К любому одиночному символу строки можно обратиться, указав номер его позиции в строке, начиная от единицы, например a[5], str15[2 * i + 3]. Можно выполнить замену одиночного символа в строке при помощи оператора присваивания, например, a[8] := 'Z'.

6.2.1 Ввод строк

Строка при вводе с клавиатуры складывается из отдельных символов, поэтому на нее распространяются особенности ввода символов, рассмотренные в подразделе 6.1.3. Ввод строки завершается нажатием клавиши Enter, что не позволяет записать в нее символы смены строки #13 и/или #10 – их при необходимости нужно добавлять путем вставки в нужное место. Нажатие клавиши пробела не разделяет вводимые строки, а вставляет в строку пробел, так что за один раз можно ввести лишь одну строку. Об этой особенности нужно помнить при вводе массива строк – ввод каждой строки завершается только нажатием клавиши Enter. То же самое касается ввода в одном операторе строк и других типов данных: строку следует завершать нажатием Enter, в то время как другие данные можно разделять и пробелами.

Операторы Read и Readln осуществляют ввод, в том числе, строк, имена которых перечисляются в списке ввода. Использование оператора Read, не очищающего буфер ввода, требует большой осторожности, поэтому начинающим программистам проще пользоваться оператором Readln и не смешивать в одном операторе ввод строк и прочих данных. Работа с нижеследующей программой должна дать ключ к пониманию такого совета.

```
begin // p06005
  var i1, i2, i3: integer;
  var s1, s2: string;
  Readln(i1, s1, i2, s2, i3);
  Writeln(i1, '|', s1, '|', i2, '|', s2, '|', i3);
end.
```

Сеанс работы №1 (после ввода числа - пробел, после ввода строки – Enter):

```
15 строка1
-243 строка2
16
15| строка1|-243| строка2|16
```

Пробелы, введенные после чисел, стали частью введенных строк. Это плохо.

Сеанс работы №2 (после ввода любого данного – Enter):

```
15
строка1
Ошибка времени выполнения: System.FormatException: Входная строка имела
неверный формат.
```

Непонятно, что произошло. Предполагалось, что будет считано число 15, затем строка «строка1», далее снова число… Раскрыть тайну поможет ввод, при котором строки также будут изображать числа, а чтобы их выделить, строки сделаем пяти-символьными: 11111, 22222, 33333.

Сеанс работы №3 (после ввода любого данного – Enter):

```
15
11111
-243
15||11111||-243
```

«Тайна» генерации исключения в сеансе работы №2 раскрыта: ввод Enter делает строку пустой.

Сеанс работы №4 (после ввода числа - пробел, после ввода строки – Enter):

```
1 строка номер 1
2 строка номер 2
3
1| строка номер 1|2| строка номер 2|3
```

Можно вводить строки, содержащие, в том числе пробелы, но впоследствии первый символ каждой введенной строки нужно удалить (см. 6.2.10).

Может быть, использование оператора Read в этом случае будет более успешным? Удалим в тексте программы подстроку «ln» после «Read» и посмотрим результаты ее работы.

Сеанс работы №1 (после ввода числа - пробел, после ввода строки – Enter):

```
1 строка номер 1
2 строка номер 2
3
1| строка номер 1|2| строка номер 2|3
```

Результаты ничем не отличаются от полученных с использованием Readln.

Сеанс работы №2 (после ввода любого данного – Enter):

```
1
11111
2
1||11111||2
```

И в этом случае результаты аналогичны. Собственно, а чего другого следовало ожидать? Оператор Readln отличается от Read лишь тем, что опустошает буфер ввода **после окончания** своей работы. И только.

Помимо операторов Read/Readln, строки можно вводить при помощи функций ReadlnString и ReadlnString(p), где p – строка приглашения ко вводу. Эти функции возвращают значение введенной строки и могут быть использованы в выражениях вместо имени строк, что сокращает текст программы.

Существуют также разновидности функций ReadlnString2 и ReadlnString3, позволяющие ввести две или три строки.

Функции без «ln» - ReadSrting, ReadSrting2 и ReadSrting3 не очищают буфер ввода при окончании работы.

Все упомянутые функции поддерживают необязательный параметр, задающий строку приглашения ко вводу.

```
begin
  var (s1, s2, s3) := ReadlnString3('Введите три строки через Enter:');
  ('(' + s1 + ')').Println;
  ('(' + s2 + ')').Println;
  ('(' + s3 + ')').Println;
end.
```

Такой ввод позволяет корректно задать содержимое всех строк:

```
Введите три строки через Enter: первая строка
строка №2
третья строка
(первая строка)
(строка №2)
(третья строка)
```

Еще один пример, использующий результат ввода строки.

```
begin // p06006
  var s1 := ReadLnString('Привет, как тебя зовут?');
  var gr := ReadlnInteger(s1 + ', укажи свой год рождения:');
  case Sign(gr - 1971) of
    -1: Println('Просто мамонт! О языке Паскаль тогда еще не знали...');
    0: Println('Да вы с Паскалем ровесники!');
    1: Println('А язык Паскаль в это время уже существовал!')
  end
end.
```

Ниже приведен сеанс работы с программой.

```
Привет, как тебя зовут? Вася
Вася, укажи свой год рождения: 1964
Просто мамонт! О языке Паскаль тогда еще не знали...
```

6.2.2 Вывод строк

Выводить строки можно любыми средствами, которые использовались для вывода других типов данных: Print и Println, Write и Writeln, а также при помощи расширений .Print и .Println.

В части расширений следует отметить, что они рассматривают строку, как последовательность отдельных символов, но выводят эти символы не через пробел, как для других типов данных, а через пустой символ. Тем не менее, расширения .Print и .Println могут принимать в качестве параметра строку-разделитель, что иллюстрирует следующая простая программа

```
begin
  'Тестовая строка'.Println;
  'Тестовая строка'.Println(' ');
  'Тестовая строка'.Println('*');
  'Тестовая строка'.Println(' = ')
end.
```

Рассмотрите вывод этой программы, чтобы понять, как работает вывод строк по расширениям .Print и .Println

```
Тестовая строка
Т е с т о в а я   с т р о к а
Т*е*с*т*о*в*а*я* *с*т*р*о*к*а
Т = е = с = т = о = в = а = я =   = с = т = р = о = к = а
```

6.2.3 Длина строки

Длина строки – это текущее количество символов в ней.

- ⍟ s.Length – свойство, возвращает длину строки s;
- s.Count – расширение, делает то же самое;
- Length(s) – функция, делает то же самое.

В приведенной ниже программе видно, что строки s2[6] и s3[20] ведут себя одинаково.

```
begin // p06007
  var s1 := 'Строка';
  var s2: string[6] := 'Строка';
  var s3: string[20] := 'Строка';
  Writeln('|',s1, '|', s2, '|', s3,'|'); // |Строка|Строка|Строка|
  Println(s1.Length, s2.Length, s3.Length); // 6 6 6
  Println(s1.Count, s2.Count, s3.Count); // 6 6 6
  Println(Length(s1),Length(s2),Length(s3)); // 6 6 6
end.
```

Как получается, что описанная с длиной 20 символов строка, после инициализации литералом длиной 6 символов внешне выступает шестисимвольной? Понять происходящее позволит еще одна программа.

```
begin // p06008
  var s: string[20];
  s.Length.Println; // 0
  s := 'Строка';
  s.Length.Println; // 6
  s := 'Строка подлиннее';
  s.Length.Println; // 16
  s := '';
  s.Length.Println; // 0
  s := 'А это очень длинная строка'; // |А это очень длинная |
  Writeln('|', s, '|');
  s.Length.Println // 20
end.
```

Что происходит? Короткая строка описана, как имеющая максимальную длину 20 символов. Поскольку при первом определении длины строка не была инициализирована, начальное значение присвоилось автоматически и строка оказалась пустой. А длина пустой строки в символах равна нулю. Присваивание строке литерала длиной 6 символов изменило длину строки до 6. При попытке поместить в s строку длиной 26 символов, вместилось только 20. Обратите внимание: никаких сообщений о потере части данных не выдается. Получается, что мы всегда выводим фактическую длину строки.

Строку можно усечь или удлинить, изменяя ее длину. Это можно сделать при помощи процедуры SetLength. Удлиняемая строка дополняется справа пробелами до нужной длины.

⍟ SetLength(s, len) – усекает или удлиняет строку s до длины len.

```
begin // p06009
  var P: string->() := s -> Println('|' + s + '|'); // лямбда" для вывода
  var s := 'Маша ела кашу';
  P(s); // '|' - чтобы видеть пробелы в начале и конце
  SetLength(s, s.Length - 4);
  P(s);
  SetLength(s, s.Length + 6);
  P(s)
end.
```

```
|Маша ела кашу|
|Маша ела |
|Маша ела       |
```

Короткая строка с типом string[k] всегда имеет длину k+1 байт. В самом первом (левом) байте находится текущее количество символов, помещенных в строку. Поэтому Length и Count вычислениями не занимаются, а читают этот байт как переменную типа **byte**, но возвращают значение с типом **integer**.

6.2.4 Арифметические операции со строками (+ и *)

Операция сложения «+», примененная к строкам, соединяет их, «склеивает» в общую строку. Если же одним из операндов является значение числового выражения (за исключением типа BigInteger), а вторым – строка, то числовое значение предварительно преобразуется к строковому виду, а потом выполняется соединение.

В операции умножения «*» один операнд должен быть строкой, второй - целочисленным выражением. В результате строка соединяется сама с собой указанное значением выражения количество раз, давая новую строку.

В одном операторе можно записывать и больше одного сложения и/или умножения.

```
begin
  Println('--- 25 + 3 * 8 = '+(25 + 3 * 8) + ' ---')
end.
```

Вывод программы будет выглядеть следующим образом:

```
--- 25 + 3 * 8 = 49 ---
```

Обязательно ли было использовать скобки в выражении (25 + 3 * 8) ? Да, обязательно. Если этого не сделать, то будут выполнены следующие шаги вычислений:

```
'--- 25 + 3 * 8 = ' + 25 // получим '--- 25 + 3 * 8 = 25'
3 * 8 // получим 24
'--- 25 + 3 * 8 = 25' + 24 // получим '--- 25 + 3 * 8 = 2524'
'--- 25 + 3 * 8 = 2524' + ' ---' // получим '--- 25 + 3 * 8 = 2524 ---'
```

Не забывайте о скобках, когда будете «склеивать» выражения в строках.

Можно «склеить» в строку и повторенный несколько раз отдельный символ, например, **var** s := 30 * '+';

StringOfChar(c, k) – функция, возвращающая строку, состоящую из повторенного k раз символа c. Вряд ли кто-то будет ее использовать, поскольку эквивалентная запись k * c (или c * k) и короче, и понятнее.

6.2.5 Сравнение строк

Содержимое строк, даже если их длина различна, можно сравнивать между собой. Строки сравниваются посимвольно слева направо до первого несовпадения кодов символов, либо до окончания одной или обеих строк. Для строк определены результаты всех шести операций сравнения: <, <=, =, <>, >, >=. Большим считается тот символ, код которого больше. Если все символы более короткой строки совпали с символами более длинной, то большей строкой считается более длинная строка.

```
begin
  Println('Кошка' > 'Мышка'); // False, 'К' < 'М'
  Println('12345' > '1234');  // True, начало совпадает, строка 1 длиннее
  Println('ЛЕН' > 'ЛЁН'); // True, 'Ё' в Unicode идет перед 'А'
  Println('лен' > 'лён'); // False, 'ё' в Unicode после 'я'
  Println('Папа' <= 'мама'); // True, 'П' < 'м', вначале прописные буквы
  Println('Подвох' = 'Подвох') // False, самая первая 'о' - латинская
end.
```

При сравнении строк возможны три варианта: первая строка длиннее, строки равны, вторая строка длиннее. Для анализа ситуации придется делать одно или два последовательных сравнения, что может порождать не отличающиеся большой наглядностью конструкции из вложенных «if». В PascalABC.NET включены средства, позволяющие частично преодолеть это неудобство.

- CompareStr(s1, s2) – функция, выполняющее сравнение строк. Она возвращает значение типа integer. Когда первая строка длиннее, возвращается положительное, когда строки равны, возвращается 0 и когда первая строка короче, возвращается отрицательное значение. Пытаться искать какую-то гарантированную закономерность в значениях этих чисел не следует.
- ⍟ string.Compare(s1, s2) – статический метод, выполняющий сравнение строк. Он возвращает значение типа integer. Когда первая строка длиннее, возвращается 1, когда строки равны, возвращается 0 и когда первая строка короче, возвращается -1. Получается своеобразная функция Sign( ). Метод удобно использовать с оператором case для организации ветвления.
- ⍟ string.Compare(s1, s2, IgnoreCase) – статический метод, делающий то же самое и при этом игнорирующий регистр буквенных символов, если в качестве IgnoreCase указано True. Для запоминания не очень удачная мнемоника: запись string.Compare(s1, s2, True) ассоциируется с каким-то «настоящим», истинным сравнением, а тут наоборот – сравнение неточное, игнорирующее регистр.

```
begin // p06010
  var Эталон := 'Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))-Sin(D)';
  var a := Arr(
      'Sqrt(A/(2*pi*(B-Sqrt(b*b-C*C))))-Sin(d)',
      'Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))-Sin(D)',
      'Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C)))-Sin(D))',
      'Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))+Sin(D)');
  Println('Эталонная строка :', Эталон);
  foreach var s in a do
  begin
    var r := string.Compare(Эталон, s, True);
    case r of
      -1: Println(s, ' - эталон меньше');
      0: Println(s, ' - отлично!');
      1: Println(s, ' - эталон больше')
    end
  end
end.
```

Будет получен следующий результат

```
Эталонная строка : Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))-Sin(D)
Sqrt(A/(2*pi*(B-Sqrt(b*b-C*C))))-Sin(d)  - отлично!
Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))-Sin(D)  - эталон меньше
Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C)))-Sin(D))  - эталон меньше
Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))+Sin(D)  - эталон больше
```

6.2.6 Копирование строк

Строки копируются обыкновенным присваиванием вида s1 := s2.

```
begin // p06011
  var s1: string[20] := '12345';
  var s2: string[20] := 'abcde';
  Println(s1, s2); // 12345 abcde
  s1 := s2;
  Println(s1, s2); // abcde abcde
  s2 := '67890';
  Println(s1, s2); // abcde 67890
  s1 := 'абвгд';
  Println(s1, s2) // абвгд 67890
end.
```

Хорошо видно, что короткие строки независимы: изменения в одной строке никак не сказываются на другой. Но это короткие строки. А как с остальными?

Строки имеют ссылочный тип и при выполнении присваивания s1 := s2 в s1 копируется ссылка на строку s2. Но тогда любое изменение в одной из строк повлечет изменение в другой строке, а на практике этого не происходит. В этом легко убедиться, изменив описание строк в приведенной выше программе:

```
var s1 := '12345';
var s2 := 'abcde';
```

Причина кроется в том, что PascalABC.NET при модификации строк использует высокоэффективный механизм «Копирование при записи», иногда называемый на профессиональном жаргоне «коровьим» (COW – Copy-on-Write). Его суть в том, что непосредственно перед модификацией создается копия строки, с которой выполняется работа, а лишь затем ссылка обновляется.

6.2.7 Выделение подстроки

Подстрока – это часть строки s, полученная путем выборки некоторых ее символов, следующих подряд. Не забывайте, что символы в строке нумеруются от единицы.

- ⍟ s.Left(k) – расширение, возвращает k левых символов строки;
- LeftStr(s, k) – функция, делает то же самое;
- ⍟ s.Right(k) – расширение, возвращает k правых символов строки;
- RightStr(s, k) – функция, делает то же самое;
- Copy(s, from, k) – функция, возвращает k символов, начиная с позиции from;
- s.Substring(from, k) – метод, возвращает подстроку длиной k, выбирая из строки s символы, начиная с позиции from. Выход за пределы строки считается ошибкой. Нумерация символов ведется от нуля.

Для получения подстроки часто бывает удобным использовать срезы. Легче запоминается, короче запись.

6.2.8 Срезы строк

Со срезами вы уже знакомы по массивам (см. 5.3.5). Срезы строк реализуются аналогично, но возвращают они подстроку, а не последовательность или массив отдельных символов. Срез может использоваться в выражениях везде, где может использоваться строка.

```
begin // p06012
  var s := 'Параграф';
  s[:5].Println; // Пара
  s[5:].Println; // граф
  s[2:5].Println; // ара
  s[3::2].Println; // рга
  s[8::-2].Println; // фраа
end.
```

Имеется также расширение .Slice, позволяющее получать срезы. Нумерация символов здесь ведется **от нуля**.

- s.Slice(f, h) – возвращает срез строки s, начиная с позиции f и выбирая символы с шагом h;
- s.Slice(f, h, k) – возвращает срез строки s длины не более k, начиная с позиции f и выбирая символы с шагом h;

```
begin
  var s := 'Параграф';
  s.Slice(0, 1, 4).Println; // Пара
  s.Slice(4, 1).Println; // граф
  s.Substring(0, 4).Println; // Пара
  s.Substring(4).Println// граф
end.
```

Составим из букв слова «инфо***р***м***ат***и***ка***» слово «карта». Возьмем слог «ка», затем букву «р» и буквы «ат» в обратном порядке (выделены в слове полужирным курсивом). Подобные задачи часто предлагают школьникам при изучении строк.

```
begin // p06013
  var s := 'информатика';
  var w := s.Right(2) + s[5] + s[8:6:-1];
  w.Println
end.
```

Здесь использованы расширение s.Left( ) (см. 6.2.7), прямое обращение к символу строки по ее позиции s[ ] и срез s[: :]. Безусловно, это не единственное решение.

6.2.9 Смена регистра символов

Как и в случае с отдельными символами, в строке можно изменять регистр всех символов или их части (см. 6.1.2). Смена выполняется функциями и расширениями, поэтому исходная строка не меняется.

- LowerCase(s) – возвращает строку s, приведенную к нижнему регистру;
- ⍟ s.ToLower – то же самое;
- UpperCase(s) – возвращает строку s, приведенную к верхнему регистру;
- ⍟ s.ToUpper – то же самое.

6.2.10 Удаление символов в начале и в конце строки

Наиболее популярным при обработке строк является удаление пробелов перед первым непробельным символом (левые, или лидирующие пробелы) и после последнего непробельного символа(правые, или завершающие пробелы).

- TrimLeft(s) – функция, возвращающая исходную строку s с удаленными лидирующими пробелами;
- s.TrimStart – метод, возвращающий исходную строку s с удаленными лидирующими пробелами;
- TrimRight(s) – функция, возвращающая исходную строку s с удаленными завершающими пробелами;
- s.TrimEnd – метод, возвращающий исходную строку s с удаленными завершающими пробелами;
- Trim(s) – функция, возвращающая исходную строку s с удаленными лидирующими и завершающими пробелами;
- s.Trim – метод, возвращающий исходную строку s с удаленными лидирующими и завершающими пробелами.

```
begin // p06014
  var P: string->() := s -> Println('|' + s + '|'); // лямбда" для вывода
  var s := '   Маша    ела    кашу    ';
  P(s); // '|' - чтобы видеть пробелы в начале и конце
  P(TrimLeft(s)); // |Маша    ела    кашу    |
  P(s.TrimStart);
  P(TrimRight(s)); // |   Маша    ела    кашу|
  P(s.TrimEnd);
  P(Trim(s)); // |Маша    ела    кашу|
  P(s.Trim);
end.
```

Кроме пробела имеются и другие непечатаемые (пробельные) символы (подраздел 6.1.1.4), поэтому встречаются задачи и на удаление всех таких символов. Функции TrimLeft( ) и TrimRight( ) эти символы не различают, а функцией Trim( ) и тремя перечисленными выше методами можно пользоваться. Если удаляется не более 5-6 символов в начале или конце строки, предлагаемые методы работают быстрее функций, в противном случае функции оказываются проворнее.

Что делать, если нужная строка окружена, например, ненужными точками, запятыми и знаками «+» ? Можно ли удалить их? Описанные методы могут решить и это проблему. Для этого нужно указать в качестве аргумента удаляемые символы, разделяя их запятыми или использовать массив символов. Не забывайте, что в этом случае пробельные символы тоже должны быть указаны, если они могут быть нежелательными.

```
Пример1: s.TrimStart(' ','.',',','+');

Пример2: var p := Arr(' ','.',',','+');
         s.TrimEnd(p);

Пример3: var pa := Arr(' ','.',',', Chr(9), Chr(10), Chr(13));
         s.Trim(pa);
```

6.2.11 Удаление подстрок

- Удалить подстроку, например, в позициях с 8 по 24, можно при помощи срезов.

```
begin
  var s := 'У Вани машина, у Наташи - мяч';
  s:=s[:8]+s[25:];
  s.Println // У Вани - мяч
end.
```

- Delete(s, from, k) – процедура, удаляющая из строки s подстроку длиной k символов, начиная с позиции from. Нумерация позиций ведется от нуля.

```
begin
  var s := 'У Вани машина, у Наташи - мяч';
  Delete(s, 7, 17);
  s.Println // У Вани - мяч
end.
```

- ⍟ s.Remove(from) – расширение, удаляющее часть строки, начиная с позиции from и до конца. Нумерация позиций ведется от нуля.
- ⍟ s.Remove(from, k) – расширение, удаляющее подстроку длины k, начиная с позиции from. Нумерация позиций ведется от нуля. Если удалить k символов невозможно, генерируется исключение.

```
begin
  var s := 'У Вани машина, у Наташи - мяч';
  s.Remove(7, 17).Println // У Вани - мяч
end.
```

- Усечь строку можно также при помощи процедуры SetLength (см. 6.2.3).
- Расширение .Remove умеет делать еще один вид удаления – по содержимому подстроки. Если в качестве параметра задать массив подстрок, либо перечислить их через запятую, из строки будут удалены все вхождения каждой из подстрок.
- ⍟ s.Remove(ss) – удаление всех вхождений в строку s каждой из подстрок, заданных перечнем или массивом.

```
begin // p06015
  var s := 'Улишние Васловани машзасоряютина, у Наслоwаташи - мятекстч';
  s := s.Remove('лишние', 'слова', 'засоряют', 'текст'); // очистим текст
  s.Println; // У Вани машина, у Наташи - мяч
end.
```

6.2.12 Инверсия

Если в строке расположить все ее символы в обратном порядке, то полученную строку называют инвертированной относительно исходной или просто инверсией. Существуют также инверсии подстроки. PascalABC.NET предлагает несколько способов получения инверсии строки.

- s.Inverse – расширение, возвращающее инверсию строки s;
- ReverseString(s) – функция, возвращающее инверсию строки s;
- ReverseString(s, from, k) – функция, возвращающая строку s, в которой инвертирована подстрока длиной k, начиная с позиции from. Нумерация позиций производится от нуля.

Можно инвертировать строку с помощью среза и это самый короткий вариант записи инверсии. Строка может рассматриваться как последовательность символов, поэтому расширение .Reverse, также может выполнить инвертирование, но в результате будет получена не строка, а последовательность символов. Все эти варианты приведены в нижеследующем примере программы.

```
begin // p06016
  var s := 'А роза упала на лапу Азора';
  var s1 := s.Inverse;
  Println(s1); // арозА упал ан алапу азор А
  var s2 := s.Reverse;
  Println(s2); // [а,р,о,з,А, ,у,п,а,л, ,а,н, ,а,л,а,п,у, ,а,з,о,р, ,А]
  var s3 := ReverseString(s);
  Println(s3); // арозА упал ан алапу азор А
  var s4 := s[::-1]; // срез
  Println(s4); // арозА упал ан алапу азор А
  var s5 := ReverseString(s, 7, 13);
  Println(s5); // А роза упал ан алапу Азора
  var s6 := s[:8] + s[20:7:-1] + s[21:]; // подстрока с помощью срезов
  Println(s6) // А роза упал ан алапу Азора
end.
```

6.2.13 Вставка подстроки

Средств для вставки подстроки в строку всего два и функционально они несколько отличаются. Процедура Insert изменяет исходную строку, а метод Insert возвращает измененную строку в качестве результата.

- Insert(ss, s, from) – процедура, вставляющая в строку s подстроку ss, начиная с позиции from. Указание несуществующей позиции приводит ко вставке подстроки перед первым (при from <1) или после последнего символа строки;
- s.Insert(from, ss) – метод, возвращающий строку, полученную путем вставки подстроки ss в исходную строку s с позиции from. Нумерация позиций ведется от нуля. Указание несуществующей позиции может вызвать исключение с сообщением «Ошибка времени выполнения: Заданный аргумент находится вне диапазона допустимых значений».

```
begin // p06017
  var s := 'У меня есть конфета';
  s.Println; // У меня есть конфета
  var s1 := s.Insert(12, 'вкусная ');
  s1.Println; // У меня есть вкусная конфета
  Insert('вкусная ', s, 13);
  s.Println // У меня есть вкусная конфета
end.
```

6.2.14 Проверки в строке

В процессе работы со строкой бывает нужно выполнить некоторые проверки ее содержимого. PascalABC.NET предоставляет достаточное количество средств для таких проверок. Результат проверки всегда имеет тип **boolean** и равен True в случае ее успешности.

- StringIsEmpty(s, p) – функция. Проверяет, все ли символы строки s, начиная с позиции p, являются пробельными. О пробельных символах сказано в (6.1.1.4). Если все, значение p станет на единицу больше длины строки, в противном слу-

чае оно не измениться. В качестве p должна быть указана переменная целочисленного типа;
- s.StartWith(ss) – метод. Проверяет, начинается ли строка s с подстроки ss;
- s.EndWith(ss) – метод. Проверяет, заканчивается ли строка s подстрокой ss;
- s.Contains(ss) – метод. Проверяет, содержит ли строка s подстроку ss;
- s.Between(s1, s2) – расширение. Проверяет, находится ли строка s между строками s1 и s2 ($s1 \leqslant s \leqslant s2$);
- s.InRange(s1, s2) – то же самое;
- s.IsMatch(reg, opt) – расширение (см. 6.2.19.2). Проверяет, удовлетворяет ли строка s регулярному выражению reg (см. 6.2.19). С помощью opt можно задавать дополнительные опции.

6.2.15 Разбиение строки на слова

Операция разбиения строки на слова порождает массив строк, где каждый элемент является словом. Слово – это чаще всего последовательность непробельных символов, ограниченная не менее чем одним пробельным символом или концом строки. Безусловно, это не отвергает вариантов, когда слово должно удовлетворять более жестким требованиям, например, содержать только буквы, или только цифры, или не включать знаков препинания и т.д.
- s.ToWords(cc) – расширение, возвращающее массив слов, полученных разбиением строки s; при этом в качестве разделителей слов могут использоваться один или более символов, перечисленных в cc через запятую, либо содержащихся в массиве символов cc. Пустые слова, порождаемые наличием нескольких следующих подряд символов из cc, в массив не попадают. Параметр cc можно не указывать, тогда разделителями слов считаются пробельные символы;
- s.Split(cc) – метод, аналогичный расширению .ToWords. В отличие от него, в массив заносит и пустые строки.

```
begin // p06018
  var s := '   Карл   у   Клары украл    кораллы    ';
  s.ToWords.Println('.'); // элементы массива выводятся через точку
  s.Split.Println('.');
  s := '= =+Карл у  Клары+++украл = = =кораллы';
  s.ToWords('=', '+', ' ').Println('.');
  s.Split('=', '+', ' ').Println('.');
end.
```

При выводе был использован символ «точка», позволяющий увидеть места, на которых находятся пустые строки. Каждая пара следующих подряд точек – это и есть место вывода пустой строки.

```
Карл.у.Клары.украл.кораллы
...Карл...у...Клары.украл....кораллы....
Карл.у.Клары.украл.кораллы
....Карл.у..Клары...украл......кораллы
```

Если список символов – разделителей слов большой, конструкция получается достаточно громоздкой. Ее можно сократить, определив массив символов разделителей.

```
var cc := '+-.,?!;:" '.ToCharArray; // метод см. 6.2.20
s.Split(cc).Println('~');
```

Можно порекомендовать использовать расширение .ToWords, если слова разделены более чем одним символом и после обработки этих слов нет нужды сохранять точное количество разделителей. Для сохранения количества и положения разделителей нужно использовать метод .Split.

6.2.16 Замена подстроки в строке

Достаточно часто бывает необходимость заменить в строке один контекст на другой. Например, поменять имя переменной в выражении на более осмысленное.

- s.Replace(s1, s2) – метод, возвращающий строку, полученную из исходной строки s заменой всех подстрок s1 на подстроки s2.

```
begin
  var s := 'Sin(2*x-1)*Sqr(Cos(x+5))+0.4*x**3';
  s.Replace('x', 'УголНаклона').Println
end.
```

```
Sin(2*УголНаклона-1)*Sqr(Cos(УголНаклона+5))+0.4*УголНаклона**3
```

Более сложные замены можно осуществить на основе расширения .RegexReplace, использующего регулярные выражения (см. 6.2.19.3).

6.2.17 Сцепление (слияние) строк

Слияние строк – операция, обратная разбиению строки. Несколько строк сливаются в одну общую. При этом в месте слияния может находиться символ-разделитель, подстрока из нескольких разделителей или не находиться ничего.

- Concat(s1, s2, …) – функция, возвращающая строку, которая является сцеплением (конкатенацией) строк s1, s2, … без использования разделителей;
- string.Join(ss, del) – статический метод, возвращающий строку, полученную сцеплением подстрок, находящихся в массиве ss. Подстрока del используется в качестве разделителя;
- ss.JoinIntoString(del) – расширение, возвращающее строку, полученную сцеплением подстрок, находящихся в массиве или последовательности ss. Подстрока del используется в качестве разделителя; по умолчанию – символ пробела.

Функция Concat особого интереса не представляет, поскольку она может быть заменена операцией сложения «+». А остальное имеет смысл рассмотреть.

Результатом разбиения строки на слова является динамический массив строк **array of string** (см. 6.2.15). Сделав в словах необходимые изменения, можно снова получить строку. Это дает возможность создавать короткие, пусть быть может, и не самые эффективные с точки зрения расходуемой памяти и времени выполнения программы.

Традиционный путь обработки строки состоит в ее последовательном просмотре с целью выделения позиций, занимаемых очередным словом и некоторых действий над символами в этих позициях. Разбиение строки на слова и последующее сцепление обработанных строк позволяет легко написать программу обработки строк, при этом в минимальные сроки и без ошибок.

В журнале «Информатика» №8 (561), 16-30.04.2008 имеется статья: Т. Богомолова, Ирина Фалина , В. Шухардина «Ступенька мастерства: решаем задачи на обработку строк». Под номером 18 для самостоятельного решения предлагается следующая задача: «Слова в строке разделяются пробелами. Зашифровать текст таким образом, чтобы каждое слово текста было записано в обратном порядке». Дается также и решение, записанное в базовом Паскале. Алгоритм предполагает использование метода барьерных элементов, который в данной реализации сводится к дописыванию пробела в конец исходной строки. Ниже приводится авторский код программы.

Операции изменения строк в PascalABC.NET выполняются весьма неэффективно и вызовы их в цикле существенно увеличивают время выполнения программы. Особенно ресурсоемки операции множественной вставки подстрок. Строки в PascalABC.NET базируются на неизменяемых строках .NET Framework и любая модификация строки связана с операцией копирования. В определенной степени решает проблему использование класса StringBuilder, о котором имеет смысл подумать в случае, когда ваш алгоритм интенсивно модифицирует строки.

```
var
  s, s1: string;
  t: char;
  a, i, j, b, p: integer;

begin
  write('st ==> ');
  readln(s);
  s := s + ' '; {Добавили барьерный элемент}
  i := 1; a := length(s);
  repeat
    if s[i] <> ' ' then
    begin
      s1 := ''; p := i;
      while s[i] <> ' ' do
      begin
        s1 := s1 + s[i];
        i := i + 1;
      end;
      b := length(s1);
      for j := 1 to b div 2 do
      begin
        t := s1[j]; s1[j] := s1[b - j + 1];
        s1[b - j + 1] := t
      end;
      delete(s, p, b);
      insert(s1, s, p);
    end
    else i := i + 1
  until i >= length(s);
  writeln(s)
end.
```

32 строки текста на Паскале. Конечно, вы уже просмотрели код и поняли алгоритм. Нет? Хорошо, посмотрите еще пару минут. Снова не поняли? Ну что же, как говорится, «тяжело в ученье – будет тяжело и в работе». Давайте теперь напишем то же самое на PascalABC.NET.

```
begin // p06019
  ReadlnString('st ==>').Split.Select(w -> w.Inverse)
      .JoinIntoString.Println
end.
```

И это все? В один оператор??? Да, все. Кстати, эта программа в PascalABC.NET еще и работает вдвое быстрее – это к вопросу об «оптимальных по времени программах».

Разберем наше решение. ReadlnString('st ==>') считывает с клавиатуры строку, выдав приглашение ко вводу, взятое из приведенной в журнале программы и возвращает ее в качестве результата. Дадим полученной строке условное имя s. Далее выполняется метод s.Split, разбивающий строку по пробельным символам (параметр не задан) и возвращающий массив полученных слов, в том числе пустых. Назовем этот массив a. Проецирование a.Select(w -> w.Inverse) применяет к каждо-

му элементу массива a расширение Inverse, в результате чего получается инверсия находящихся в массиве слов. Результат проецирования - последовательность, которую обозначим p. Расширение p.JoinIntoString объединяет элементы в строку, используя в качестве разделителя принятый по умолчанию пробел. Получаем некоторую строку s, которую s.Println и выводит на монитор. Писать все это дольше, чем саму программу.

6.2.18 Поиск в строке

Под поиском понимается нахождение в строке номера позиции отдельного символа или подстроки. Такая позиция еще может называться индексом, если рассматривать строку как динамический массив символов; при этом индексирование производится от нуля.

- Pos(ss, s) – функция. Возвращает позицию первого вхождения подстроки ss в строку s. Возвращает ноль, если подстрока не найдена;
- PosEx(ss, s) – то же, для совместимости с Delphi;
- Pos(ss, s, from) – функция. Возвращает позицию первого вхождения подстроки ss в строку s, начиная поиск с позиции номер from. Возвращает ноль, если подстрока не найдена;
- PosEx(ss, s, from) – то же, для совместимости с Delphi;
- LastPos(ss, s) – функция. Возвращает позицию последнего вхождения подстроки ss в строку s. Возвращает ноль, если подстрока не найдена;
- LastPos(ss, s, from) – функция. Возвращает позицию последнего вхождения подстроки ss в строку s, начиная поиск в обратном направлении с позиции from. Фактически, значение from ограничивает поиск первыми from символами. Возвращает ноль, если подстрока не найдена;
- s.IndexOf(ss) – метод. Возвращает позицию первого вхождения подстроки ss в строку s. Нумерация позиций от нуля. Возвращает -1, если подстрока не найдена;
- s.IndexOf(ss, from, k) – метод. Возвращает позицию первого вхождения подстроки ss в строку s, начиная поиск с позиции from и распространяя его на k символов. Можно представить себе, что сначала делается срез строки, а затем поиск ведется в нем, но индекс при этом возвращается относительно начала исходной строки. Нумерация позиций от нуля. Возвращает -1, если подстрока не найдена;
- s.LastIndexOf(ss) – метод. Возвращает позицию полследнего вхождения подстроки ss в строку s. Нумерация позиций от нуля. Возвращает -1, если подстрока не найдена;
- s.LaseIndexOf(ss, from, k) – метод. Возвращает последнего первого вхождения подстроки ss в строку s, начиная поиск в обратном направлении с позиции from и распространяя его на k символов. Нумерация позиций от нуля. Возвращает -1, если подстрока не найдена;
- s.IndexOfAny(cc) – метод. Возвращает индекс первого вхождения в строку s любого символа из массива cc. Нумерация позиций от нуля. Возвращает -1, если ни один символ не найден;

- s.LastIndexOfAny(cc) – метод. Возвращает индекс последнего вхождения в строку s любого символа из массива cc. Нумерация позиций от нуля. Возвращает -1, если ни один символ не найден;
- s.MatchValue(reg, opt) – расширение. Возвращает первую подстроку в строке s, соответствующую регулярному выражению reg (см. 6.2.19.4). С помощью opt можно задавать дополнительные опции;
- s.MatchValues(reg, opt) – расширение. Возвращает последовательность подстрок в строке s, соответствующих регулярному выражению reg (см. 6.2.19.5). С помощью opt можно задавать дополнительные опции;
- s.Matches(reg, opt) – расширение. Возвращает последовательность элементов типа Match из строки s, соответствующих регулярному выражению reg (см. 6.2.19.6). С помощью opt можно задавать дополнительные опции.

Найдем позиции всех вхождений слова в тексте при помощи функции Pos. Обратите внимание на использование переменной i в левой и правой частях оператора присваивания с вызовом Pos. Еще одна особенность программы – «накопление» результатов в символьной строке r.

```
begin // p06020
  var s :=
      'Лениво дышит полдень мглистый,' + NewLine +
      'Лениво катится река.' + NewLine +
      'И в тверди пламенной и чистой' + NewLine +
      'Лениво тают облака.' + NewLine +
      '(Ф. И. Тютчев)';
  Writeln(s, NewLine, '-' * 45);
  var (i, k, r) := (1, 0, 'Слово "Лениво" найдено в позициях ');
  repeat
    i := Pos('Лениво', s, i);
    if i = 0 then break;
    r += i + ' ';
    i += 1;
    k += 1;
  until False;
  if k = 0 then Println('Слово "Лениво" в строке не обнаружено')
  else
  begin
    r.Println;
    Println('Количество вхождений:', k)
  end
end.
```

Программа находит позиции всех трех вхождений подстроки «Лениво».

```
Лениво дышит полдень мглистый,
Лениво катится река.
И в тверди пламенной и чистой
Лениво тают облака.
(Ф. И. Тютчев)
----------------------------------------
Слово "Лениво" найдено в позициях 1 33 86
Количество вхождений: 3
```

Чтобы показать, насколько регулярные выражения упрощают работу со строками, сравните приведенную выше программу со следующей, дающей точно такой же результат.

```
begin // p06021
  var s :=
      'Лениво дышит полдень мглистый,' + NewLine +
      'Лениво катится река.' + NewLine +
      'И в тверди пламенной и чистой' + NewLine +
      'Лениво тают облака.' + NewLine +
      '(Ф. И. Тютчев)';
  Writeln(s, NewLine, '-' * 45);
  var r := s.Matches('Лениво').Select(m -> m.Index + 1).JoinInToString;
  if r.Length > 0 then
  begin
    Println('Слово "Лениво" найдено в позициях', r);
    Println('Количество вхождений:', r.Where(c -> c = ' ').Count + 1)
  end
  else Println('Слово "Лениво" в строке не обнаружено')
end.
```

Каждый элемент Match хранит всю информацию относительно результатов поиска, в том числе, саму найденную подстроку, ее длину и позицию в исходной строке. Это позволило спроектировать элементы на номера позиций в строке. Получилась последовательность типа integer, элементы которой расширение JoinIntoString преобразовало в строку, разделив значения пробелами.

6.2.19 Использование регулярных выражений

Термин ***регулярное выражение*** (далее по тексту РВ) пришел из теории формальных языков и отражает свойство математических выражений, называемое ***регулярностью***. Желающие разобраться в том, что такое регулярность, могут обратиться к специальной литературе. Официального стандарта, определяющего, что является РВ, а что нет, в мире не существует, поэтому имеется множество диалектов РВ.

Платформа Microsoft .NET Framework обеспечивает поддержку диалекта в стиле языка Perl, так что при изучении РВ можно пользоваться литературой и по этому языку. За поддержку РВ отвечает пакет System.Text.RegularExpressions, называемый в обиходе RegEx.

Внешне РВ представляет собой некоторую строку-шаблон, с которой сопоставляется обрабатываемый текст. Результатом сопоставления будут подстроки, удовлетворяющие РВ. На основании содержимого этих подстрок делается вывод о наличии или отсутствии или чего-либо в тексте, осуществляется выборка подстрок или их замена.

РВ состоит из метасимволов – специальных символьных комбинаций и литералов – всех прочих символов. Проводя аналогию с Паскалем, метасимволы – это ключевые слова, а литералы – все остальное. Метасимволы объясняют, как интерпретировать литералы в РВ для выполнения поиска в заданной строке.

При работе с РВ учитывается набор значений параметров, определяющих специфику поиска, например, нужно ли учитывать регистр символов, останавливаться после первого успешного поиска или продолжать его и т.д. Эти параметры могут задаваться как в самом РВ, так и в операторе языка, использующего это РВ.

РВ посвящены целые книги. Понятно, что здесь невозможно сколь-нибудь подробно рассказать о РВ или научить профессиональной работе с ними, поэтому будут приведены лишь базовые сведения, достаточные для понимания примеров использования РВ.

Неверно думать, что РВ способны решать любые проблемы обработки текстовых строк. Их удел – поиск (и, возможно, замена) подстрок. Есть отличная фраза: «Если для решения проблемы вы не к месту попытаетесь использовать регулярное выражение, у вас станет две проблемы. Вторая – как применить это регулярное выражение».

Далее, когда речь будет идти о РВ, на основе которого производится поиск, наряду с термином «регулярное выражение» будет употребляться термин «шаблон».

6.2.19.1 Некоторые метасимволы

. (точка) – самый простой метасимвол. Он означает, что на данном месте в строке может встретиться любой символ. Например, слова клан, клен, клин и клон удовлетворяют шаблону 'кл.н', а шаблону вида '...' будет удовлетворять любое трехбуквенное сочетание символов. При этом не следует забывать, что пробел тоже является символом, поэтому и он удовлетворяет метасимволу «точка». Если нужно сослаться на символ «точка», то нужно примерить ***экранирование*** – указать перед точкой символ «\», получая комбинацию «\.».

\d – метасимвол «digit – цифра», означающий цифру от 0 до 9. Двухзначное число может быть изображено с помощью этого метасимвола в виде «\d\d». Если дана строка «У меня есть словарик на 23000 слов, изданный в 1974 году», то шаблон '\d\d' позволит найти подстроки «23», «00», «19» и «74», а по '\d\d\d' – подстроки «230» и «197».

\D – метасимвол, означающий не цифру. Вообще, в метасимволах РВ запись строчных букв означает наличие соответствующих символов, а прописных – их отсутствие.

\w – метасимвол «word – слово», обозначающий все то, из чего строится слово, т.е. буквы, цифры и знак подчеркивания. Во многих операционных системах и языках программирования под буквами понимается только латинский алфавит. В PascalABC.NET сюда включаются также буквы национальных алфавитов, поэтому не нужно беспокоиться по поводу кириллицы. Соответственно, все не \w обозначается, как \W. Во фразе «У меня есть словарик на 23000 слов, изданный в 1974 году» шаблону '\w\w\w\w' удовлетворяют подстроки «меня», «есть», «слов», «арик», «2300», «слов», «изда», «нный», «1974» и «году».

\s – метасимвол «space – пробел», обозначающий любой пробельный символ. \S означает любой непробельный символ.

^ – метасимвол, обозначающий начало строки, но только если он не входит в **символьный класс** (см. ниже).

$ – метасимвол, обозначающий конец строки.

Теперь вы можете потренироваться в нахождении подстрок. Для этого имеется достаточное количество онлайновых ресурсов. Один из наиболее подходящих для наших целей - //nregex.com

Почему именно этот ресурс? Он работает с РВ, предназначенными для Microsoft .NET Framework. Ресурс имеет англоязычный интерфейс, но это непринципиально. При входе на Nregex в середине окна имеется большое поле с текстом «If I just had $5.00 …». Удалите этот текст и введите в этом поле строку, в которой будет производиться поиск, например, приведенную выше «У меня есть словарик на 23000 слов, изданный в 1974 году». Шаблон вводится в строку под надписью «Regular Expression:», выполненную красным цветом, например, \d\d\d. Все найденные совпадения будут выделены.

Онлайновое тестирование РВ – отличный инструмент для начинающих. Намного удобнее, чем видеть сообщение о неверном шаблоне после запуска своей программы.

[…] – метасимвол, называемый «символьный класс». Внутри квадратных скобок перечисляются (без разделителей!) символы, которые могут представлять данный метасимвол, например [аи]. Шаблону кл[аи]н соответствуют подстроки «клан» и «клин».

[^…] – метасимвол, называемый «инвертированный символьный класс». Ему соответствуют символы, не перечисленные внутри скобок.

- (дефис) – метасимвол, но только в контексте символьного класса. Обозначает интервал. [A-Z] – все символы, коды которых находятся в интервале кодов от A до Z. Русская буква представляется в виде [А-Яа-яЁё]. Почему «Ёё» отдельно? Потому что в таблице Unicode эти две буквы стоят особняком.

(…|...|…) – метасимвол «конструкция выбора». Выбирается одна из альтернатив, разделенных «|» - символом «ИЛИ». Например (кот|кошка). Инверсии для этой конструкции не существует.

? - метасимвол, относящийся к ***квантификаторам*** (определяет количество повторов). Указывает, что символ, непосредственно предшествующий ему, является необязательным, т.е. может отсутствовать или присутствовать один раз. Например, шаблону «сон.?» удовлетворяет и слово «сон», и слово «соня».

+ - метасимвол, относящийся к квантификаторам. Указывает, что символ, непосредственно предшествующий ему, должен встретиться не менее одного раза. Шаблону «\d+» удовлетворяет последовательность цифр любой длины, а шаблон «[+-]?\d+» описывает произвольное целое число, которое может иметь знак.

* - метасимвол, относящийся к квантификаторам. Указывает, что символ, непосредственно предшествующий ему, может отсутствовать или встретиться сколько угодно раз. Шаблону «[+-]?\d+\.\d\d*» соответствуют любые числа с фиксированной точкой. Разберем его состав. «[+-]?» указывает, что символ может присутствовать знак «+» или «-». «\d+» сообщает, что ожидается не менее одной цифры. «\.» - это знак точки, потому что указано экранирование. Без экранирования это обозначало бы наличие одного произвольного символа. «\d\d*» - часть шаблона, указывающая, что на этом месте должна быть одна цифра, за которой могут следовать другие цифры. Целиком шаблон читается примерно так: «Необязательный знак плюс или минус, за которым следует одна или более цифр, далее следует точка, за которой находится не менее одной цифры». Это адекватно описывает запись любого числа с фиксированной точкой.

{m, n} – метасимвол, относящийся к квантификаторам. Указывает, что символ, непосредственно предшествующий ему, может встретиться не менее m и не более n раз. Если n=m, его можно опустить вместе с предшествующей запятой. Но если n не ограничено, его опускают, сохраняя запятую, и квантификатор {m,} означает «не менее m раз». Шаблон «\d{3}» определяет натуральное трехзначное число, а шаблон «\d{1,2}\.\d{1,2}.\d{4}» - дату в формате дд.мм.гггг.

6.2.19.2 IsMatch – наличие подстроки в строке

Расширение s.IsMatch(reg, opt) проверяет, удовлетворяет ли строка s регулярному выражению reg. С помощью opt можно задавать дополнительные опции.

В приводимом примере рассматривается строка «Роза увяла от мороза», в которой ищется подстрока «роза». Если учитывать регистр, то такая подстрока одна, без учета регистра их две.

```
begin // p06022
  var s := 'Роза увяла от мороза';
  s.IsMatch('роза').Println; // True
  s.IsMatch('роза\s').Println; // False
  s.IsMatch('Роза\s').Println; // True
  s.IsMatch('роза\s', RegexOptions.IgnoreCase).Println; // True
  s.IsMatch('(?i)роза').Println; // True
end.
```

В предпоследнем варианте проверки опция RegexOptions.IgnoreCase означает требование игнорировать регистр букв. В последнем варианте тот же эффект достигается при помощи конструкции (?i).

6.2.19.3 RegexReplace – замена подстрок в строке

Расширение s.RegexReplace(reg, ss, opt) возвращает строку, полученную заменами в строке s подстроки, удовлетворяющей регулярному выражению, подстрокой ss. С помощью opt можно задавать дополнительные опции. Делаются замены всех найденных подстрок.

```
begin // p06023
  var s := 'Мама мыла раму, Маша ела кашу';
  s.RegexReplace('М','Д').Println; // Дама мыла раму, Даша ела кашу
  s:='23*x-Sin(x)';
  s.RegexReplace('x','(x+1.5)').Println; // 23*(x+1.5)-Sin((x+1.5))
  s.RegexReplace('\-Sin\(x\)','').Println; // 23*x
end.
```

В последней замене, показывающей удаление подстроки, обратите внимание на экранирование. Начинающим полезно вначале убедиться при помощи IsMatch в правильности поиска, а затем делать замену.

Вторая форма расширения s.RegexReplace(reg, Match -> string, opt) отличается тем, что найденные подстроки заменяются их преобразованием, заданным лямбда-выражением. Match – это объект одноименного класса, являющийся результатом каждого удачного сопоставления шаблона reg со строкой s.

```
begin // p06024
  var s := 'Мама мыла раму';
  // смена регистра
  s.RegexReplace('.a', m -> UpperCase(m.Value)).Println; // МАМА мыЛА РАму
  // номер позиции вхождения подстроки
  s.RegexReplace('.a', m -> m.Index.ToString).Println; // 02 мы7 10му
  // длины слов
  s := 'А роза упала на лапу Азора';
  s.RegexReplace('\w+', m -> m.Length.ToString).Println; // 1 4 5 2 4 5
  // заменить пробелы их количеством
  s := '       тестовая  строчка   ';
  s.RegexReplace('\s+', m -> m.Length.ToString).Println; // 7тестовая2строчка3
end.
```

6.2.19.4 MatchValue – поиск первого вхождения подстроки

Расширение s.MatchValue(reg, opt) возвращает первую из подстрок в строке s, соответствующую регулярному выражению reg. С помощью opt можно задавать дополнительные опции. В отличии от расширения IsMatch (см. 6.2.19.2) возвращает подстроку, а не логическое значение, показывающее успешность поиска. В качестве примера возьмем тот же самый, что для IsMatch.

```
begin // p06025
  var s := 'Роза увяла от мороза';
  s.MatchValue('роза').Println; // роза
  s.MatchValue('роза\s').Println; // пустая строка
  s.MatchValue('Роза\s').Println; // Роза
  s.MatchValue('роза\s', RegexOptions.IgnoreCase).Println; // Роза
  s.MatchValue('(?i)роза').Println; // Роза
end.
```

6.2.19.5 MatchValues – поиск всех вхождений подстроки

Расширение s.MatchValues(reg, opt) возвращает последовательность подстрок в строке s, соответствующую регулярному выражению reg. С помощью opt можно задавать дополнительные опции.

```
begin // p06026
  var s := 'Роза увяла от мороза';
  s.MatchValues('роза').Println; // роза
  s.MatchValues('роза\s').Println; // пустая строка
  s.MatchValues('Роза\s').Println; // Роза
  s.MatchValues('роза\s', RegexOptions.IgnoreCase).Println; // Роза
  s.MatchValues('(?i)роза').Println; // Роза роза
end.
```

В последнем обращении к MatchValues было найдено две подстроки. Первая получена как слово «Роза» с игнорированием регистра, вторая – как подстрока слова «мо**роза**».

6.2.19.6 Matches – результаты поиска в элементах типа Match

Расширение s.Matches(reg, opt) возвращает последовательность элементов типа Match из строки s, соответствующих регулярному выражению reg (см. 6.2.19). С помощью opt можно задавать дополнительные опции.

Тип Match объявлен в пакете System.Text.RegularExpressions и подробная справка по нему может быть найдена в описании этого пакета. Каждый элемент Match хранит всю информацию относительно результатов поиска, в том числе, саму найденную подстроку, ее длину и позицию в исходной строке. Некоторые детали описаны ниже (см. 6.2.19.8).

Пример работы с расширением Matches уже приводился (см. 6.2.18).

6.2.19.7 Опции регулярного выражения

Опции (необязательные возможности) RegexOprions регулярных выражений обозначались выше, как opt. Приведем лишь наиболее употребительные четыре опции из десяти имеющихся них. В скобках указаны коды опций.

- None (0) – указывает на отсутствие опций;
- IgnoreCase (1) – игнорирование регистра буквенных символов;
- Multiline (2) – многострочный режим. Если текст содержит символы перевода строки, они рассматриваются, как обычные символы и поиск производится во всей строке;
- Singleline (16) – однострочный режим. Если текст содержит символы перевода строки, строка рассматривается только до первого такого символа.

Опции задаются в виде параметра RegexOptions.Опция. Вместо имен опций можно писать их коды в виде RegexOptions(Код). Если нужно указать несколько опций, коды складываются, как целые числа.

Некоторый недостаток использования опций состоит в том, что они действуют в пределах всего регулярного выражения. В отдельных случаях может помочь использование опций непосредственно в выражении. С этой целью в нужных местах включаются конструкции вида (?⍟) или (?-⍟), где ⍟ - так называемый ***инлайновый символ***, обозначающий опцию. В пакете System.Text.RegularExpressions имеются пять инлайновых символов, но мы отметим три из них:

- i – IgnoreCase (1);
- m – Multiline (2);
- s – Singleline (16).

Знак «минус» перед инлайновым символом означает отмену действия соответствующей опции.

По умолчанию действует многострочный режим с учетом регистра.

```
begin // p06027
  var s := 'Карл у Клары украл кораллы';
  // Карл Клары крал кораллы
  s.MatchValues('(?i)к\w*(?-i)(ар|ра)\w*').Println;
  s.MatchValues('[Кк]\w*(ар|ра)\w*').Println;
end.
```

Первый шаблон приведен лишь для того, чтобы показать, как используются инлайновые символы. Второй шаблон показывает, что в данном случае можно превосходно обойтись и без них.

6.2.19.8 Элементы Matches

Как уже упоминалось, тип Match объявлен в пакете System.Text.RegularExpressions. Каждая подстрока, соответствующая успешному поиску в соответствии с шаблоном, сохраняется в объекте типа Match. Объект имеет достаточно сложную структуру, но на практике чаще всего используются только три его поля:

- Match.Value – сохраненная подстрока;
- Match.Index – позиция в исходной строке, в которой найден первый символ сохраненной подстроки (отсчет с единицы);
- Match.Length – длина сохраненной подстроки.

Практически, расширение MatchValue возвращает в качестве результата значение поля Value первого из элементов Matches, а MatchValues – последовательность значений поля Value из всех элементов Matches.

6.2.20 Преобразование строки в динамический массив

Строку логически можно рассматривать, как динамический массив символов. Но можно действительно преобразовать строку в динамический массив символов.

- s.ToCharArray – метод, возвращающий динамический массив типа char, полученный путем посимвольного разбиения исходной строки s. Обратное преобразование из массива или последовательности символов (массив превращается в последовательность в подавляющем количестве операций с ним – это вы уже знаете) в строку можно выполнить при помощи расширения .JoinIntoString.

Пусть требуется выбрать из заданной строки все символы по одному разу, отсортировать их по возрастанию и заменить полученной строкой исходную.

```
begin // p06028
  var s := 'наша тестовая строка';
  s := s.ToCharArray.Distinct.Sorted.JoinIntoString;
  s.Println //  авекнорстшя
end.
```

6.2.21 Преобразование целых чисел к строке

Целые числа может потребоваться преобразовать к строковому представлению для формирования строк вывода, для анализа цифр в числе (число цифр, одинаковые цифры, перестановки цифр и т.д.), для передачи куда-либо данных в символьном виде и во многих других случаях.

- Str(n, s) – процедура. Преобразует значение n любого целочисленного типа, кроме BigInteger, к строке s;
- IntToStr(n) – функция. Возвращает преобразованное к строке значение n типов integer и int64;
- ⍟ n.ToString – метод. Возвращает преобразованное к строке значение n.

Метод .ToString является единственным из перечисленных, позволяющим преобразовать к строке тип **BigInteger**.

6.2.22 Преобразование вещественных чисел к строке

Вещественные числа преобразуют к строке в тех же случаях, что и целые.

- Str(n, s) – процедура. Преобразует значение n типа real или single к строке s;
- FloatToStr(n) – функция. Возвращает преобразованное к строке значение n типа real;
- n.ToString – метод. Возвращает преобразованное к строке значение n вещественного типа.

6.2.23 Преобразование строки к числу

С преобразованием числа к строке все достаточно просто. Обратное преобразование намного сложнее.

Строка может целиком представлять одно число. В этом случае говорят о преобразовании строки к числу. Строка может содержать представление нескольких чисел одного типа, отделенных друг от друга каким-то разделителем или группой разделителей (чаще всего – пробелом, знаком табуляции #09 или запятой). Такая строка может быть преобразована к массиву чисел. Наконец, во всех случаях строка может содержать некорректное представление числа или одного из ряда чисел.

6.2.23.1 Преобразование строки к целому числу

- s.ToInteger – расширение. Возвращает содержимое сроки s, преобразованное к значению типа integer. В случае ошибке генерируется исключение;
- StrToInt(s) – функция. Возвращает содержимое сроки s, преобразованное к значению типа integer. В случае ошибке генерируется исключение;
- s.ToInteger(k) – расширение. Возвращает содержимое сроки s, преобразованное к значению типа integer. В случае ошибки возвращается значение k типа integer;
- Val(s, n, err) – процедура. Преобразует содержимое строки s к значению n типа integer. В случае успешного преобразования err=0, в противном случае err>0;
- TryStrToInt(s, n) – функция. Преобразует содержимое строки s к значению n типа integer. В случае успешного преобразования возвращает True. Если преобразование не удалось, возвращает False;
- s.TryToInteger(n) – расширение. Преобразует содержимое строки s к значению n типа integer. В случае успешного преобразования возвращает True. Если преобразование не удалось, возвращает False;
- integer.TryParse(s) – статический метод. Возвращает содержимое сроки s, преобразованное к значению типа integer. В случае успешного преобразования возвращает True. Если преобразование не удалось, возвращает False;
- StrToInt64(s) – функция. Возвращает содержимое сроки s, преобразованное к значению типа int64. В случае ошибке генерируется исключение;
- TryStrToInt(s, n) – функция. Преобразует содержимое строки s к значению n типа int64. В случае успешного преобразования возвращает True. Если преобразование не удалось, возвращает False;

- int64.TryParse(s) – статический метод. Возвращает содержимое сроки s, преобразованное к значению типа int64. В случае успешного преобразования возвращает True. Если преобразование не удалось, возвращает False;
- s.ToBigInteger – Возвращает содержимое сроки s, преобразованное к значению типа BigInteger. В случае ошибке генерируется исключение;.

Статический метод .TryParse также имеется в целочисленных классах **shortint**, **byte**, **smallint**, **word**, **cardinal** и **BigInteger**.

6.2.23.2 Преобразование строки к вещественному числу

- s.ToReal – расширение. Возвращает содержимое сроки s, преобразованное к значению типа real. В случае ошибке генерируется исключение;
- StrToReal(s) – функция. Возвращает содержимое сроки s, преобразованное к значению типа real. В случае ошибке генерируется исключение;
- StrToFloat(s) – функция. Возвращает содержимое сроки s, преобразованное к значению типа real или single. В случае ошибке генерируется исключение;
- s.ToReal(r) – расширение. Возвращает содержимое сроки s, преобразованное к значению типа real. В случае ошибки возвращается значение r типа real;
- TryStrToReal(s, r) – функция. Преобразует содержимое строки s к значению r типа real. В случае успешного преобразования возвращает True. Если преобразование не удалось, возвращает False;
- TryStrToFloat(s, r) – функция. Преобразует содержимое строки s к значению r типа real или single. В случае успешного преобразования возвращает True. Если преобразование не удалось, возвращает False;
- TryStrToSingle(s, r) – функция. Преобразует содержимое строки s к значению r типа single. В случае успешного преобразования возвращает True. Если преобразование не удалось, возвращает False
- s.TryToReal(r) – расширение. Преобразует содержимое строки s к значению r типа real. В случае успешного преобразования возвращает True. Если преобразование не удалось, возвращает False;
- real.TryParse(s) – статический метод. Возвращает содержимое сроки s, преобразованное к значению типа real. В случае успешного преобразования возвращает True. Если преобразование не удалось, возвращает False.

Статический метод .TryParse также имеется в классе **single**.

6.2.23.3 Преобразование строки к массиву чисел

- s.ToIntegers – расширение. Возвращает содержимое сроки s, преобразованное к динамическому массиву значений типа integer. В случае ошибке генерируется исключение;
- s.ToReals – расширение. Возвращает содержимое сроки s, преобразованное к динамическому массиву значений типа real. В случае ошибке генерируется исключение.

Разделителями могут выступать только пробельные символы.

6.2.24 Пример с преобразованиями строк и чисел

Пусть дана некоторая строка. Внутри строки содержатся простые выражения вида «(Число ЗнакОперации Число)», где числа только целые без знака, знак операции – это «+», «-», «*» и «/». Числа и знаки операции зашумлены прочими символами. Требуется выделить выражения и каждое записать на отдельной строке в виде Выражение = ВычисленноеЗначение.

```
function Calc(v: string): string; // p06029
begin
  var m := v.MatchValues('(\d+|[+\-*/])').ToArray;
  var a := m[0].ToInteger;
  var op := m[1][1];
  var b := m[2].ToInteger;
  case op of
    '+': Result := ' = ' + (a + b);
    '-': Result := ' = ' + (a - b);
    '*': Result := ' = ' + (a * b);
    '/': Result := ' = ' + (a / b);
  end;
  Result := v + Result
end;

begin
  var s := 'ar(f?3yh535v^+et14!f)re(@2T6\s/1we1r)(y14 2p*23)gr(18-253)';
  s.RegexReplace('[^()+\-*/0-9]', '').MatchValues('\([^)]+\)')
      .Select(v -> Calc(v)).PrintLines
end.
```

Будут получены следующие результаты:

```
(3535+14) = 3549
(26/11) = 2.36363636363636
(142*23) = 3266
(18-253) = -235
```

Замена с помощью RegexReplace удалит из строки весь «мусор». Использован шаблон '[^()+\-*/0-9]', который заменяет пустой строкой (а по факту просто удаляет) подстроки-символы, не являющиеся круглыми скобками, знаками арифметических операций и цифрами. Как обычно, символ «минус» экранирован, поскольку его наличие в символьном классе создает диапазон. Полученная строка передается MatchValues с шаблоном, формирующим последовательность подстрок, состоящих из элементов, заключенных в круглые скобки. Подстроки проецируются на функцию Calc, делающую вычисления и формирующую окончательный вид подстроки. Полученная последовательность подстрок отображается на мониторе посредством PrintLines.

Функция Calc получает из входной строки вида «(Число1 ЗнакОперации Число2)» помощью шаблона в MatchValues последовательность из трех элементов типа **string**. Поскольку последовательность не хранится, а с каждым элементом нужно работать отдельно, последовательность сохраняется в массиве m. Теперь подстрока представлена элементом m[0] для Число1, m[1] для знака операции и m[2] для

Число2. Поместим числовые значения для Число1 и Число2 в переменные a b и соответственно.

Операция – это строка длиной в один символ. И вот здесь есть неудобство, заложенное идеологией языка Паскаль. Тип односимвольного литерала нельзя интерпретировать однозначно. В операторе case используются как раз односимвольные литералы. И компилятор интерпретирует их тип, как **char**. Массив m имеет тип **array of string**, поэтому m[1] тоже имеет тип **string**. При попытке использовать m[1] в case компилятор зафиксирует несоответствие типа. В Паскале нет отдельных функций или методов для преобразования типа **string** -> **char** и наоборот. Поэтому приходится прибегать к шаманству. Если s – строка типа **string**, то ее отдельный символ будет иметь тип **char**. Поэтому мы можем написать s[1] и получить значение нужного нам типа. Преобразовать символ «c» к строке проще: можно его присвоить переменной типа **string** или сложить с пустой строкой: c + ''.

Вычисленное значение автоматически преобразуется к строке, поскольку таковы правила преобразования в операции «+» соединения строки с числом (см. 6.2.4).

6.2.25 Чтение данных из строки

Операция чтения данных из строки s состоит в получении подстрок и преобразовании их в значения требуемого типа. Указывается начальная позиция чтения from, а после успешного чтения from увеличивается на длину прочитанной подстроки, позволяя считывать очередное значение.

Чтение данных может быть полезно при получении строк из текстовых файлов (см. часть 9). Предполагается, что числа в строке могут быть разделены произвольным количеством любых пробельных символов, в том числе, признаками смены строки. Это позволяет прочитать в одном цикле многострочный файл.

6.2.25.1 Чтение из строки данных типа integer

- s.ReadInteger(from) – расширение. Начиная с позиции from, считывается символьное представление целого числа, ограниченного пробельным символом или концом строки. Затем оно преобразуется и возвращается как значение типа integer. Значение from увеличивается на длину прочитанной подстроки и будет содержать позицию первого после окончания числа пробельного символа. Если преобразование не удалось или from указано за пределами строки s, генерируется исключение;
- ReadIntegerFromString(s, from) – функция. Делает то же самое;
- TryReadIntegerFromString(s, from, n) – функция. Начиная с позиции from, считывается символьное представление целого числа и делается попытка преобразовать его к целому числу. Если попытка успешна, значение сохраняется в n и функция возвращает True. В противном случае функция возвращает False.

```
begin // p06030
  var s := '123'#9#13#10'56 00001'; // так тоже можно строить строки
  var from := 1;
  loop 3 do
    s.ReadInteger(from).Print // 123 56 1
end.
```

6.2.25.2 Чтение из строки данных типа real

- s.ReadReal(from) – расширение. Начиная с позиции from, считывается символьное представление вещественного числа, ограниченного пробельным символом или концом строки. Затем оно преобразуется и возвращается как значение типа real. Значение from увеличивается на длину прочитанной подстроки и будет содержать позицию первого после окончания числа пробельного символа. Если преобразование не удалось или from указано за пределами строки s, генерируется исключение;
- ReadRealFromString(s, from) – функция. Делает то же самое;
- TryReadRealFromString(s, from, r) – функция. Начиная с позиции from, считывается символьное представление вещественного числа и делается попытка преобразовать его к вещественному числу. Если попытка успешна, значение сохраняется в r и функция возвращает True. В противном случае функция возвращает False.

```
begin // p06031
  var s := 'Значение 3.1416 приблизительно равно числу "пи"';
  var from := Pos('3', s);
  Println('Считали число', s.ReadReal(from)) // Считали число 3.1416
end.
```

6.2.25.3 Чтение из строки символьных данных

- s.ReadWord(from) – расширение. Начиная с позиции from, считывается слово, ограниченное пробельным символом или концом строки. Возвращаемое значение имеет тип string. Значение from увеличивается на длину прочитанного слова будет содержать позицию первого после окончания слова пробельного символа. При попытке читать за пределами строки генерируется исключение;
- ReadWordFromString(s, from) – функция. Делает то же самое.

Вот так можно вывести строку, удалив из нее лишние пробелы:

```
begin // p06032
  var s := '   Карл   у   Клары украл   кораллы';
  var (l, from) := (s.Length, 1);
  while from <= l do
    Print(s.ReadWord(from)); // Карл у Клары украл кораллы
  Println
end.
```

6.2.26 Форматирование данных для вывода

Для оформления выводимых данных в языке Паскаль традиционно используется процедуры Write и Writeln, в которых для выводимого элемента данных может указываться минимальная ширина поля вывода w, а для вещественных значений – еще и количество цифр в дробной части t. Разделителем служит символ «двоеточие» (:). Если параметр w не указан или имеет недостаточную для отображения выводимого значения величину, значение выводится со своей фактической длиной. Если для вещественного числа параметр t не указан, в дробной части выводится фактическое количество цифр, но при этом общее количество цифр не может превышать 15. Если в дробной части больше цифр, чем указано в t, выводится t цифр и при этом делается округление. При указании значения t, требующего в выводимом числе отобразить более 15 цифр, все цифры после пятнадцатой будут представлены нулями.

```
begin // p06033
  var (a, b) := (-23.4723423290834, 16554);
  Writeln(a, ' ', b, ' ', a * b);
  Writeln(a, ' ', b, ' ', a * b:0:5);
  Writeln(a, ' ', b, ' ', a * b:20);
  Writeln(a, ' ', b, ' ', a * b:25:5);
  Writeln(a, ' ', b, ' ', a * b:0:20);
  Writeln('Тестовая строка', ' ', '*');
  Writeln('Тестовая строка':10, ' ', '*');
  Writeln('Тестовая строка':30, ' ', '*');
end.
```

Обратите внимание на частое использование нуля при задании ширины поля вывода w. Это типичный прием вывода вещественных значений в Паскале, когда неизвестно количество цифр в целой части числа.

```
-23.4723423290834 16554 -388561.154915647
-23.4723423290834 16554 -388561.15492
-23.4723423290834 16554    -388561.154915647
-23.4723423290834 16554             -388561.15492
-23.4723423290834 16554 -388561.15491564700000000000
Тестовая строка *
Тестовая строка *
              Тестовая строка *
```

В случает, когда значение вещественного числа слишком мало или велико, будет использован формат вывода с плавающей точкой вида x.x…xE∓nn ($x.x…x \cdot 10^{\mp nn}$).

Выводимые числовые значения прижимаются **к правой границе** отведенного для вывода поля шириной w за счет добавления слева нужного количества пробелов. Это общепринято. Точно так же, принято прижимать строки **к левой границе** поля, поэтому они дополняются пробелами справа. Но тут Паскаль ведет себя странно: пробелы все равно добавляются **слева** и об этом надо помнить.

Существуют и другие способы вывода данных. Их общая идея состоит в том, чтобы формировать строки такими, какими они должны отображаться и затем выводить без изменения.

6.2.26.1 Выравнивание строки пробелами

Чтобы строка s длиной L имела нужную ширину w, можно дополнить ее w-L пробелами слева или справа. Например, вот так:

```
s := ' ' * (w - s.Length) + s;
s := s + ' ' * (w - s.Length);
```

Но так писать для каждого выводимого элемента неудобно. Поэтому в классе **string** были введены два метода:

- s.PadRight(w) – возвращает строку s, дополненную пробелами справа до длины w;
- s.PadLeft(w) – возвращает строку s, дополненную пробелами слева до длины w.

6.2.26.2 Составное форматирование

Выравнивание строки пробелами решает проблему с выводом символьных данных. Для любого типа данных есть общее решение – использование ***составного форматирования*** (composite format в терминологии Microsoft .NET Framework).

Составное форматирование использует в качестве исходных данных строку составного формата (шаблон, задающий правила преобразования данных) и список объектов, подлежащих форматированию. Этот список следует за строкой составного формата и отделяется от нее запятой. Элементы в списке нумеруются от нуля и разделяются запятыми.

Строка составного формата может содержать произвольную комбинацию некоторого количества неизменяемого текста, представленного литералами, и элементов форматирования, каждый из которых заключается в фигурные скобки.

Элемент форматирования является ***местозаполнителем*** (поле, в котором впоследствии будет размещено значение) и в простейшем случае содержит только индекс – порядковый номер элемента, подлежащего форматированию, в списке.

```
begin
  Format('Длина окружности диаметра {0} равна {1}', 2.16, 2.16 * Pi).Println
end.
```

```
Длина окружности диаметра 2.16 равна 6.78584013175395
```

В общем случае элемент форматирования имеет вид

{*индекс*, *выравнивание* : *формат*},

где *выравнивание* – целое значение со знаком, указывающее общую ширину отформатированного поля, *формат* – строка формата для элемента.

Если выравнивание имеет отрицательное значение, результат прижимается влево путем добавления справа пробелов до нужной ширины.

В Microsoft .NET Framework определен обширный набор форматов и здесь будет рассмотрена только небольшая их часть. В частности, не рассматриваются настраиваемые числовые форматы.

§ 1. Для целочисленных данных используется формат d или D (decimal), за которым может быть указано желаемое количество цифр.

```
begin
  Format('*{0,20:d}*{0,-15:D6}*{0:d1}*', -152).Println
end.
```

```
*                -152*-000152        *-152*
```

Здесь трижды указан индекс 0, что позволило три раза использовать одно и то же значение -152 из списка. Числовое значение после d (D) определяет минимальное желаемое количество выводимых цифр, значение выравнивания перед двоеточием – общее количество позиций. Обратите внимание на появление незначащих нулей.

§ 2. Формат вещественных данных f или F определяет преобразование для отображения с фиксированной точкой (fixed), т.е. в виде целой и дробной части, разделенных точкой. По умолчанию в дробной части будут указаны две цифры с округлением значения.

```
begin
  Format('*{0,20:f}*{0,-15:f6}*{0:f1}*', -152.327).Println
end.
```

```
*             -152.33*-152.327000    *-152.3*
```

§ 3. Формат вещественных данных e или E определяет преобразование для отображения с плавающей точкой (exponential), т.е. в виде одной цифры в целой части, нужного количества цифр в дробной части и показателя степени числа 10, отделенного буквой E. Этот формат обычно используется для величин, имеющих очень малые или очень большие значения. По умолчанию число знаков в дробной части равно шести.

```
begin
  Format('*{0,20:e}*{0,-15:E6}*{0:e1}*', -152.327e-31).Println
end.
```

```
*      -1.523270e-029*-1.523270E-029 *-1.5e-029*
```

§ 4. Общий формат числовых данных g или G определяет преобразование для данных любого числового типа. Наиболее подходящий формат (d, f, e) компилятор выбирает самостоятельно. Для отображения значений **real** с максимальной точностью используется формат g17, для **single** – g9.

```
begin
  Format('*{0,20:g}*{0,-15:G6}*{0:g1}*', -152).Println;
  Format('*{0,20:g}*{0,-15:g6}*{0:g1}*', -152.327).Println;
  Format('*{0,20:g}*{0,-15:G6}*{0:g1}*', -152.327e-31).Println
end.
```

```
*                -152*-152           *-2e+02*
*            -152.327*-152.327       *-2e+02*
*        -1.52327e-29*-1.52327E-29   *-2e-29*
```

§ 5. Формат для преобразования целочисленных данных к их шестнадцатеричному представлению x или X. По умолчанию отображается минимально-необходимое количество цифр.

```
begin
  Format('*{0,20:x}*{0,-15:X6}*{0:x1}*', 1023).Println;
  Format('*{0,20:x}*{0,-15:X6}*{0:x1}*', -1023).Println
end.
```

```
*                 3ff*0003FF         *3ff*
*            fffffc01*FFFFFC01       *fffffc01*
```

6.2.26.3 Функция и статический метод Format

Составное форматирование является основой преобразования строк при помощи функции и статического метода Format.

- Format(cf, params) – функция. Возвращает строку, полученную форматированием списка параметров params при помощи строки составного формата cf;
- **string**.Format(cf, params) – статический метод. Делает то же самое.

Примеры работы с Format приведены в предыдущем подразделе.

6.2.26.4 Вывод с использованием составного форматирования

- WriteFormat(cf, params) – процедура. Форматирует список параметров params при помощи строки составного формата cf и выводит полученную строку;
- WritelnFormat(cf, params) – процедура. Делает то же самое, осуществляя затем переход на новую строку вывода.

6.2.26.5 Интерполированные строки

Интерполированная строка – это объединение строки составного формата и списка объектов, подлежащих преобразованию в строковое представление. Получается как бы «два в одном» в сравнении с компонентами Format.

Интерполированная строка представляет собой литерал, перед которым записан символ $. Литерал содержит интерполированные выражения. Интерполированное выражение очень похоже на элемент форматирования строки составного формата: оно так же заключается в фигурные скобки и содержит описание формата данных, только вместо индекса подлежащего форматированию выражения указывается само это выражение. Проще всего это продемонстрировать в сравнении с примерами, приведенными в 6.2.26.2.

```
begin // p06034
  Format('Длина окружности диаметра {0} равна {1}', 2.16, 2.16 * Pi).Println;
  $'Длина окружности диаметра {2.16} равна {2.16 * Pi}'.Println;
  Format('*{0,20:d}*{0,-15:D6}*{0:d1}*', -152).Println;
  $'*{-152,20:d}*{-152,-15:D6}*{-152:d1}*'.Println;
  Format('*{0,20:f}*{0,-15:f6}*{0:f1}*', -152.327).Println;
  $'*{-152.327,20:f}*{-152.327,-15:f6}*{-152.327:f1}*'.Println
end.
```

Обычно интерполированные строки получаются короче и нагляднее, чем строки составного форматирования.

```
begin
  var (a, b) := (5.2, 4.17);
  Format('Сумма квадратов катетов a={0} и b={1} равна {2}',
      a, b, a * a + b * b).Println;

$'Сумма квадратов катетов a={a} и b={b} равна {a * a + b * b}'.Println
end.
```

```
Сумма квадратов катетов a=5.2 и b=4.17 равна 44.4289
Сумма квадратов катетов a=5.2 и b=4.17 равна 44.4289
```

Интерполированная строка – это строка типа **string** и с ней можно работать обычным образом.

```
begin // p06035
  var (a, b) := (5.2, 4.17);
  var s1 := $'Сумма квадратов катетов a={a} и b={b} равна {a*a+b*b}';
  var s2 := $'{a*a} + {b*b} = {a*a+b*b}';
  s2.Println;
  s1.Println
end.
```

```
27.04 + 17.3889 = 44.4289
Сумма квадратов катетов a=5.2 и b=4.17 равна 44.4289
```

6.3 Для самостоятельного решения

Т6.1. Внутри произвольной строки, вводимой с клавиатуры, удалить пробелы, заменив каждую их группу ровно одним пробелом. Удалить также все пробелы, окаймляющие строку (т.е. находящиеся до первого и после последнего символа, отличающегося от пробела).

Т6.2. Разновидность задачи Т6.1, в которой строка представлена литералом и требуется сохранить ее исходное разбиение на строчки. Исходная строка:

```
var s := '  Да    охранюся я     от    мушек,  ' + NewLine +
    '  От    дев, не     знающих   любви,   ' + NewLine +
    '    От дружбы слишком    нежной и    –   ' + NewLine +
    'От   романтических    старушек.  ' + NewLine +
    ' ' * 20 + 'М.  Ю.  Лермонтов   ';
```

Т6.3. В задаче Т.6.2 сохраните имеющийся пробельный отступ в последней строчке, выделяющей автора стихотворения. Недопустимо использовать знание о том, что пробелов было двадцать, но можно получить это значение программным путем.

Т6.4. Определим слово, как произвольную последовательность непробельных символов, ограниченных границами строки и/или пробельными символами. Из произвольной строки удалите три наиболее длинных слова, сохранив все прочие символы. В качестве теста предлагается строка

```
var s := '    Однажды  Лебедь,    Рак, да    Щука' + NewLine +
      '  Везти с  ' + Chr(9) + '  поклажей    воз взялись   ';
```

Т6.5. Подсчитать количество цифр 7 в факториале числа 10000. Число 10000! = 1×2×3×…×10000.

Часть 7

Типы данных

Свободное от смысла и раздумья
Геройски-вдохновенное безумье.
Джон Драйден.
«Авессалом и Ахитофель»

Как все настоящие программисты знают, единственной полезной структурой данных является массив. Строки, списки, структуры и наборы - это все разновидности массивов и их можно рассматривать как массивы без усложнения вашего языка программирования.
Эд Пост, «Настоящие программисты не используют Паскаль»,
DATAMATION, July 1983

Типов данных в PascalABC.NET довольно много. Но так уж устроен человек, что ему всегда хочется большего. Например, определить данные собственного типа (он называется ***пользовательским типом данных***), чтобы с ними было удобно работать в решаемой задаче. В реальных задачах пользовательские типы данных вводятся практически всегда.

Пользовательский тип данных строится на основе уже имеющихся (или ранее определенных) типов данных. Он может быть подмножеством какого-то типа данных, некоторой совокупностью элементов определенного типа (например, последовательностью или массивом) и даже достаточно сложной конструкцией, называемой ***записью*** (**record**). На базе пользовательского типа можно создавать другой пользовательский тип и это очень часто используется.

Тип данных описывается в разделе типов **type**, который должен предшествовать первой программной единице, использующей его. Само описание выглядит достаточно просто:

```
имя типа = тип;
```

Иногда пользовательский тип вводят для того, чтобы повысить уровень читаемости программы. Например, в программе обрабатывающей результаты контрольных работ, можно ввести тип Оценки, представляющий динамический массив целого типа.

```
type
  Оценки = array of integer;
begin
  var Информатика, Математика, РусскийЯзык, Физика: Оценки;
  SetLength(Информатика, 27); // количество оценок
  SetLength(Математика, 30);
  SetLength(РусскийЯзык, 30);
  SetLength(Физика, 28);
  // остальная часть программы
end.
```

Если пользовательский тип заменяет одно имя другим, такие типы называют ***синонимами***, например

```
Целое = integer;
Логическое = boolean;
```

PascalABC.NET позволяет использовать ***обобщенный тип***, для которого конкретный тип определяется позднее (см. 7.6). Операция замещения конкретным типом обобщенного называется ***инстанцированием***.

> В этой части тип данных будет условно обозначаться T, если явно не указано иное. Исключение составляет обобщенный тип данных (см. 7.6), в котором действительно указывается тип <T>.

Пользовательские типы данных в PascalABC.NET предоставляют программисту намного более широкие возможности, чем в базовом Паскале. Записи очень схожи с классом, но имеют некоторые отличия, о которых будет сказано при рассмотрении классов. Записи могут содержать так называемый ***конструктор***, позволяющий выполнять инициализацию данных, а также иметь свойства, методы и многое другое, свойственное классам. С появлением в Паскале классов и кортежей роль записей стала более скромной. Тем не менее, при работе с типизированными файлами (см. часть 9) записи оказываются вне конкуренции.

7.1 Записи

Записью называется набор элементов, в котором каждый элемент имеет имя и называется ***полем записи***.

Пусть требуется написать программу для работы с простыми дробями. Дробь состоит из числителя и знаменателя, причем оба они имеют целочисленный тип. В этом случае можно описать пользовательский тип Дробь, представляющий собой запись с полями Числитель и Знаменатель, имеющими тип **integer**.

```
type
  Дробь = record
    Числитель, Знаменатель: integer
  end;

begin
  var a, b, c: Дробь; // три дроби
  var ma: array of Дробь; // массив дробей
  // ...
end.
```

Как видно из приведенного примера, запись имеет заголовок, в котором указывается имя записи. Заголовок содержит ключевое слово **record**, объявляющий запись (можно сказать, что запись имеет фиксированный в языке тип **record**). Далее следуют описания полей без использования ключевого слова **var**. Заголовок не отделяется от описания полей привычной точкой с запятой. Описание записи

завершается ключевым словом **end**, перед которым можно не указывать точку с запятой.

После того, как тип объявлен, можно описывать переменные этого типа.

7.1.1 Обращение к полям записи

Для обращения к полям записи используется точечная нотация. Указывается имя переменной, идентифицирующее запись, а затем через точку – имя поля. Если поле, в свою очередь, имеет собственные поля, после имени поля ставится еще одна точка и т.д.

```
type
  ТипФИО = record
    Фамилия: string;
    Имя: string;
    Отчество: string
  end;

  Ученик = record
    ФИО: ТипФИО;
    ДатаРождения: DateTime;
    Класс: integer;
  end;

  Класс = array of Ученик;

begin
  var Класс10в: Класс;
  SetLength(Класс10в, 26); // класс из 26 учеников
  for var i := 0 to Класс10в.High do // инициализация
  begin
    Класс10в[i].ФИО.Фамилия := ReadlnString('Фамилия:');
    Класс10в[i].ФИО.Имя := ReadlnString('Имя:');
    Класс10в[i].ФИО.Отчество := ReadlnString('Отчество:');
    Класс10в[i].Класс := 10;
    // ....

  end;

  // ...
end.
```

В разделе описания типов объявлены три пользовательских типа данных, два из которых – записи. Обратите внимание на порядок описания типов: сначала описан тип ТипФИО, а затем тип Ученик, имеющий поле типа ТипФИО. Последним описан тип, являющийся массивом элементов типа Ученик.

В основной программе создается переменная типа Класс – динамический массив элементов типа Ученик. С помощью SetLength массиву выделяется память для

размещения 26 элементов – именно столько учеников будет в классе. Затем нужно занести исходные данные, для чего организуется цикл с перебором по всем ученикам. Построение обращения к конкретному полю понятно из приведенного фрагмента программы.

Для вывода значений полей записи приходится строить обращение к каждому полю. Необходимость записывать для каждого поля конструкцию вида a.b.c.d.e.f… наводит на мысль о том, что не следует увлекаться созданием записей со сложной иерархической структурой. В отладочных целях можно пользоваться процедурой Write, выводящей запись полностью и со всеми полями. Вывод производится в круглых скобках, поля перечисляются через запятую.

Несколько упростить обращение к подобным полям позволяет оператор **with**, но он объявлен устаревшим и рекомендован к использованию только в целях совместимости.

7.1.2 Конструктор записи

Внутри записи можно объявлять процедуры и функции, которые в этом случае называются ***методами***. Это терминология объектно-ориентированного программирования и подробности будут рассмотрены при изучении классов. На самом деле тип **record** является классом, поэтому на него распространяется соответствующая терминология.

Один из методов мы выделим особо. Он представляет собой функцию с фиксированным именем Create, называемую ***конструктором***, причем это имя настолько фиксировано, что его даже можно не писать! В описании конструктора вместо ключевого слова **function** указывается **constructor**, а тип возвращаемого значения не указывается.

Сейчас наиболее важной для нас является способность конструктора инициализировать поля записи, а остальные его функции мы пока не будем рассматривать.

Конструктор можно не описывать, и тогда среда Microsoft .NET Framework сама создает конструктор без параметров, который инициализирует все числовые поля нулями, строковые – пустой строкой, логические – значением False. Если пользователь описал собственный конструктор (и даже несколько конструкторов с разными параметрами), будет вызван тот конструктор, параметры которого совпадут по типу и количеству с указанными при создании записи.

Для вызова конструктора можно использовать два способа. Первый и основной – это вызов в стиле языка C# с использованием ключевого слова **new**, за которым указывается тип записи и далее в круглых скобках следует перечень фактических параметров для передачи конструктору. Второй способ оставлен для совместимости с Object Pascal и Delphi. Указывается тип записи, за ним через точку слово Create

и далее в круглых скобках следует перечень фактических параметров для передачи конструктору.

Параметры, переданные конструктору, используются для инициализации полей записи, поэтому конструктор в записи обычно описывается в случае, когда такая инициализация нужна.

На стр. 255 приводился пример записи для работы с простыми дробями. Целые числа также могут участвовать в этой работе, а любое целое число можно представить дробью со знаменателем, равным единице. Поэтому полезно заранее инициализировать знаменатель единицей. Добавим в запись конструкторы с тем, чтобы ее можно было инициализировать при объявлении.

```
// p07001
type
  Дробь = record
    Числитель, Знаменатель: integer;

    constructor(a, b: integer);
    begin
      (Числитель, Знаменатель) := (a, b)
    end;

    constructor(a: integer);
    begin
      (Числитель, Знаменатель) := (a, 1)
    end;

  end;

begin
  var a := new Дробь(3, 11); // дробь, стиль вызова конструктора C#
  var b := new Дробь(13); // целое, стиль вызова конструктора C#
  var c := Дробь.Create(-1143, 65434); // дробь, стиль вызова Object Pascal
  var d: Дробь; // описание без вызова конструктора
  Writeln(a, NewLine, b, NewLine, c, NewLine, d);
end.
```

```
(3,11)
(13,1)
(-1143,65434)
(0,0)
```

Объявлено два конструктора. Первый для дробей, имеющих числитель и знаменатель. Второй – для целых чисел и в нем в знаменатель заносится единица. Показаны оба способа вызова конструктора. При обычном описании, как это сделано для переменной d, конструктор не вызывается. Но ничто не мешает впоследствии при надобности написать d := **new** Дробь(m, n) и выполнить инициализацию.

7.1.3 Инициализаторы полей

Поля в записи можно инициализировать и без конструктора, совмещая их описание с присваиванием значения. Значение может быть константой или выражением, но в последнем случае компилятор должен иметь возможность вычислить его значение. Поменять значение поля, заданное инициализатором, можно при вызове конструктора или после создания записи путем присваивания, либо чтения данных в это поле.

```
// p07002
type
  Дробь1 = record
    Числитель := 0;
    Знаменатель := 1
  end;

  Дробь2 = record
    Числитель, Знаменатель: integer;

    constructor(a: integer; b: integer := 1);
    begin
      (Числитель, Знаменатель) := (a, b)
    end;

  end;

begin
  var a1, b1, c1: Дробь1;
  a1.Числитель := 3; a1.Знаменатель := 11; // дробь 3/13
  b1.Числитель := 13; // целое число 13
  Writeln(a1, NewLine, b1, NewLine, c1, NewLine);
  var a2 := new Дробь2(3, 11); // дробь, стиль вызова конструктора C#
  var b2 := new Дробь2(13); // целое, стиль вызова конструктора C#
  var c2 := Дробь2.Create(-1143, 65434); // стиль вызова Object Pascal
  var d2: Дробь2; // описание без вызова конструктора
  Writeln(a2, NewLine, b2, NewLine, c2, NewLine, d2);
end.
```

```
(3,11)
(13,1)
(0,1)

(3,11)
(13,1)
(-1143,65434)
(0,0)
```

7.1.4 Инициализация записи

Инициализации записи уже упоминалась выше (см. 7.1.2). Для полноты материала здесь эти сведения будут приведены повторно.

В случае, когда описывается константа или переменная, инициализацию записи можно совмещать с описанием в весьма неуклюжем инициализаторе записи, как это делалось в Delphi.

Пусть имеется описание записи:

```
type
  Дробь = record
    Числитель, Знаменатель: integer;
  end;
```

- Инициализация «в стиле Delphi» может быть проведена в следующем виде

```
const a: Дробь = (Числитель: 13; Знаменатель: 137);
var b: Дробь := (Числитель: 5; Знаменатель: 19);
```

Непонятно, чем руководствовались разработчики Object Pascal, придумывая необходимость указывать имена полей в инициализаторе. Ведь поменять местами эти поля нельзя, так к чему указывать имена? Если полей много, инициализатор становится очень громоздким. Такая инициализация может найти применение для записей, не содержащих конструктора, а также при инициализации констант.

- При наличии конструктора его можно вызвать, как в Delphi, считая статическим (классовым) методом. С этой целью при инициализации переменной после знака операции присваивания записывается конструкция вида Тип.Create(список значений).

```
var b := Дробь.Create(5,19);
```

Здесь подразумевается, что в описании записи имеется конструктор, принимающий два параметра.

- Конструктор можно также вызывать, как в языке C#, для чего используется ключевое слово new. При инициализации переменной после знака операции присваивания записывается конструкция вида new Тип(список значений). Это современный и наиболее предпочтительный стиль программирования.

```
var b := new Дробь(5, 19);
```

- Вместо конструктора можно в описании записи определить собственный метод инициализации и затем вызвать его. Преимуществом является возможность вызывать метод несколько раз, инициализируя лишь поля записи, упоминающиеся в нем. Но если инициализация производится однократно, то такой способ оказывается лишь более громоздким, чем два предыдущих, поскольку требует в одном операторе описать переменную, а в другом – инициализировать ее.

```
type
  Дробь = record
    Числитель, Знаменатель: integer;

    procedure Init(a, b: integer);
    begin
      (Числитель, Знаменатель) := (a, b)
    end;
  end;

begin
  var b: Дробь;
  b.Init(5, 19);
  Writeln(b)
end.
```

7.1.5 Вывод переменной типа запись

Вывести все поля записи позволяет процедура Write/Writeln. Если в списке вывода несколько элементов и пробел в качестве разделителя устраивает, можно пользоваться Print/Println. Весь вывод заключается в круглые скобки, а значения полей отделяются запятой. Все это хорошо для отладки, но в остальных случаях никуда не годится. В самом деле, вряд ли пользователя обрадует отображение числа 6 в виде (6, 1) при работе с простыми дробями.

Вывод можно переопределить, для чего в описании записи следует задать собственный метод-функцию с именем ToString, возвращающую строку и имеющую в заголовке описатель **override**. Это также даст возможность преобразовывать поля записи в строку.

```
// p07003
type
  Дробь = record
    Числитель, Знаменатель: integer;

    constructor(a, b: integer);
    begin
      (Числитель, Знаменатель) := (a, b)
    end;

    function ToString: string; override;
    begin
      Result:= $'{Числитель}/{Знаменатель}';
    end;
  end;

begin
  var b := new Дробь(5, 19);
  Println(b) // выводится 5/19
end.
```

Конечно, теперь и b.ToString.Println; даст аналогичный результат. Если в записи типа Дробь определить методы, позволяющие проводить арифметические операции, получим полноценный тип для работы с дробями.

7.1.6 Операция присваивания для записей

Записи относятся к данным размерного типа. Это означает, в частности, что присваивание выполняет копирование значений всех полей. Присваивание разрешено, если правая и левая части имеют один и тот же тип, либо один из типов является поддиапазоном другого.

```
// p07004
type
  Дробь = record
    Числитель, Знаменатель: integer;

    constructor(a, b: integer);
    begin
      (Числитель, Знаменатель) := (a, b)
    end;

    function ToString: string; override;
    begin
      Result:= $'{Числитель}/{Знаменатель}';
    end;
  end;

begin
  var b := new Дробь(5, 19);
  var c := b;
  c.Знаменатель := 21;
  b.ToString.Println; // вывод 5/19
  c.ToString.Println // вывод 5/21
end.
```

В приведенном примере имеется присваивание записи, после которого изменяется значение одного из полей. Второе поле не изменяется с тем, чтобы показать факт присваивания. Как следует из примера, при присваивании возможно автовыведение типа.

7.1.7 Сравнение записей

Вариантов здесь немного. Записи могут быть равны друг другу или не равны. Равенство записей означает, что для каждой пары их одноименных полей выполняется равенство значений. Рассматривая пример p07004 можно утверждать, что после присваивания **var** c := b выражение b = c будет иметь значение True, а после выполнения оператора c.Знаменатель := 21; равенства уже не будет.

Равны ли записи a и b, если для них имеется описание var a, b: T и для них не выполнялось ни инициализации, ни присваиваний? Равны, поскольку по умолчанию все поля в типе T будут инициализированы одинаково.

7.1.8 Передача записей в качестве параметров

Записи, содержащие более одного поля, во избежание ненужного копирования нужно передавать в подпрограммы по ссылке. Если содержимое полей записи в подпрограмме менять не планируется, ссылку передают как константу (**const**), в противном случае – как переменную (**var**). Будет ли ошибкой, если забыть указать **const**? Нет, не будет, программа останется работоспособной. Но если запись содержит поля с данными размерного типа и эти поля имеют большой размер (например, статический массив из тысячи элементов), скорость выполнения программы замедлится на несколько порядков.

```
// p07005
type
  МойТип = record
    Поле := 100
  end;

procedure Вывод(const k: МойТип);// не меняем
begin
  Writeln(k)
end;

procedure Правка(var k: МойТип; Значение: integer);// меняем
begin
  k.Поле := Значение
end;

begin
  var r := new МойТип;
  Вывод(r); // (100)
  Правка(r, 29);
  Вывод(r) // (29)
end.
```

7.2 Перечислимый тип

Конструктор перечислимого типа имеет вид заключенного в круглые скобки списка неповторяющихся идентификаторов (имен), разделенных запятыми. Имена, как обычно, строятся по правилам языка PascalABC.NET. И тут возможность использовать в них символы национальных алфавитов оказывается, как нельзя кстати. Перечислимый тип является порядковым, каждый его элемент занимает в памяти 4 байта. Значения перечислимого типа можно использовать в качестве параметра цикла со счетчиком, меток оператора case, индексов в статических массивах и так далее, что повышает уровень наглядности программы.

```
type
  ДниНедели = (Пн, Вт, Ср, Чт, Пт, Сб, Вс);
```

Имея такое описание, можно объявить переменную с типом ДниНедели и присвоить ей любое из возможных значений. Автовыведение типа работает и здесь, поэтому оператор **var** d := Чт; вполне корректен. Поскольку Чт является именем, а регистр букв в тексте программы на языке Паскаль игнорируется везде, кроме символьных и строковых литералов, записанных в одинарных кавычках, можно было указать также ЧТ, чТ или чт – все это будет воспринято, как Чт.

Вместо описания перечислимого типа в разделе типов можно указать его конструктор при описании переменной, например var Пол: (М,Ж), но это скорее экзотика, потому что завести вторую переменную такого же типа можно только присваиванием копии имеющейся переменной, т.е. var ВашПол := Пол.

Функция Ord(t) для переменной перечислимого типа t возвращает порядковый номер значения t в списке типа, начиная от нуля. Функция Pred(t) для переменной перечислимого типа t возвращает предшествующее значение, Succ(t) – последующее. Для приведенного примера с днями недели значению Чт предшествует Ср, а следующим будет Пт.

Операция приведения типа позволяет получить значение элемента по порядковому номеру в списке типа, т.е. совершает действие, обратное Ord. Для приведенного примера ДниНедели(1) вернет Вт.

Процедуры инкремента Inc(t, n) и декремента Dec(t, n) служат для изменения значения переменной t путем смещения по списку вперед (Inc) или назад (Dec) на n элементов. Если t=Чт, то Inc(t,3) установит t в Вс, а Dec(t,2) – во Вт. В случае, когда n=1, можно использовать укороченную запись Inc(t), Dec(t).

Данные перечислимого типа можно сравнивать при помощи всех шести операций сравнения. Сравниваются, конечно же, порядковые номера значений в списке типа.

> Внимание! Перечислимый тип реализуется при помощи целочисленной арифметики и контроль выхода значений за отведенные границы **отсутствует**. Если в описании типа список содержит n элементов и происходит переход к несуществующему элементу с номером k, лежащим вне интервала [0; n-1], в качестве имени возвращается значение k с учетом его знака.

В нашем примере при t = Чт вызов Inc(t,20) установит t в значение 3 + 20 = 23, а Dec(t, 8) установит t в значение 3 – 8 = -5. Функция Succ(Вс) вернет 7, а Pred(Пн) вернет -1.

Перечислимый тип – пример типа, у которого недостатки в большинстве случаев перевешивают достоинства, поэтому используется он сравнительно редко.

7.3 Диапазонный тип

Это тоже порядковый тип, представляющий собой подмножество данных целого, символьного или перечислимого типа. Описывается в виде a .. b, где a и b – границы диапазона значений, которые могут принимать данные, причем a < b. Тип, на основе которого строится диапазонный тип, называется ***базовым типом***. Длина, отводимая в памяти элементу диапазонного типа, совпадаете с длиной элемента базового типа. Совпадает также набор допустимых операций и операторов.

```
type
  ДвухзначноеЦелое = 10..99;
  ДниНедели = (Пн, Вт, Ср, Чт, Пт, Сб, Вс);
  ВыходныеДни = Сб..Вс;
  МалыеЛатинскиеБуквы = 'a'..'z';
```

7.4 Эквивалентность и совместимость типов

Рассмотрим простую программу, которая выведет единственное число 10.

```
type
  Яблоки = byte;
  Груши = integer;

begin
  var k1: Яблоки := 10;
  var k2: Груши := 12;
  k2 := k1;
  Print(k2)
end.
```

В программы были определены два пользовательских типа: Яблоки и Груши. Типы разные, но имеющие общий базовый тип – числовой. Мы знаем, что если в выражении встречаются два разных типа, компилятор пытается выполнить приведение типа k1 к типу k2. В данном случае это сделать можно, поэтому ошибки не возникает, и мы получаем возможность поместить количество яблок в количество груш. Вполне логично: и то, и другое – фрукты (целочисленные данные). В этом случае говорят, что ***типы совместимы***.

Изменится ли картина, если объявить тип Груши синонимом типа **real**? Нет, потому что тип Яблоки приводим и к этому типу, т.е. совместимость тут тоже есть.

Будет ли такая совместимость сохраняться для более сложных типов данных, например, динамических массивов, при условии сохранения совместимости типов их элементов?

```
type
  v1 = array of integer;
  v2 = array of byte;

begin
  var a1: v1 := (1, 2, 3);
  var a2: v2 := (4, 5, 6);
  a2 := a1; // динамические массивы
  Print(a2[1]);
end.
```

Получаем сообщение об ошибке: «Нельзя преобразовать тип array of integer к array of byte». Если оба массива будут одного типа, например **integer**, программа отработает корректно и выведет результат 2.

А теперь посмотрим, что происходит при использовании статических массивов.

```
type
  v1 = array[1..3] of integer;
  v2 = array[1..3] of integer;

begin
  var a1: v1 := (1, 2, 3);
  var a2: v2 := (4, 5, 6);
  a2 := a1;
  Print(a2[1]);
end.
```

Удивительно, но здесь мы получим сообщение об ошибке «Нельзя преобразовать тип array [1..3] of integer к array [1..3] of integer». Запрет на присваивание при неотличимых описаниях типов? Но ведь динамические массивы присваивать можно! Увы, так решил в свое время Н. Вирт, создавая язык Паскаль. Типы v1 и v2 **разные**, поскольку они **описаны отдельно**. Динамическим массивам просто повезло, что Н. Вирт их в язык не ввел. Кто-то может подумать: «Велика ли беда, ну нельзя присвоить один статический массив другому!». А ведь велика! Раз нельзя присвоить, нельзя передать массив в подпрограмму, если фактический и формальный параметры имеют разные имена типов. И вернуть массив нельзя без выполнения того же условия. Поэтому писать в данном случае нужно так:

```
type
  v1, v2 = array[1..3] of integer;
```

7.4.1 Совпадение типов

Типы T1 и T2 считают ***совпадающими***, если они имеют одно имя или определены в секции **type** как синонимы (7). Пусть даны описания

```
type
  v1: array [-2..3] of integer;
  v2: v1;

var
  a1: v1;
  a2: v2;
  b, c: array [1..3] of integer;
  d: array [1..3] of integer;
```

Здесь у переменные a1 и a2 типы совпадают, поскольку типы v1 и v2 - синонимы. Переменные b и c имеют один и тот же тип по описанию. Переменная d имеет другой тип, который не совпадает ни с одним из прочих типов, приведенных в данном описании.

7.4.2 Эквивалентность типов

Типы T1 и T2 считаются ***эквивалентными***, если выполняется одно из условий:

- T1 и T2 совпадают;
- T1 и T2 – динамические массивы, типы элементов которых совпадают;
- T1 и T2 – множества или указатели с совпадающими базовыми типами (см. 7.7);
- T1 и T2 – процедурные типы с совпадающим списком формальных параметров (для фунций – еще и с совпадающим типом возвращаемого значения).

Эквивалентность считается быть ***именной***, если она достигается совпадением имен типов. В противном случае эквивалентность считается ***структурной***, поскольку она является результатом совпадения структуры в описании типов. Структурная эквивалентность имеет место для динамических массивов, множеств, типизированных указателей и процедурных типов. Для всех прочих типов эквивалентность может быть только именной.

7.4.3 Совместимость типов

Типы T1 и T2 считаются ***совместимыми***, если выполняется одно из условий:

- T1 и T2 эквивалентны;
- T1 и T2 принадлежат к целочисленным типам;
- T1 и T2 принадлежат к вещественным типам;
- T1 и T2 являются поддиапазонами некоторого типа или один из типов – поддиапазон другого;
- T1 и T2 являются множествами с совместимыми базовыми типами.

Типы могут быть ***совместимыми по присваиванию***. Значение типа T1 можно присвоить переменной типа T2, если выполняется одно из условий:

- T1 и T2 совместимы;
- T1 имеет вещественный тип, T2 – целочисленный;
- T1 имеет строковый тип, T2 – символьный;

- T1 – бестиповый указатель **pointer** , T2 – типизированный указатель;
- T1 – указатель или процедурная переменная, T2 = **nil**;
- T1 – процедурная переменная, T2 – имя процедуры или функции со структурно эквивалентными параметрами
- T1 и T2 имеют классовый тип (см. часть 13), причем один из типов унаследован от другого. В PascalABC.NET все типы, кроме указателей, наследуются от класса Object, поэтому значение любого из таких типов можно присвоить переменной типа Object;
- T1 имеет тип интерфейса (см. часть 11), T2 – тип класса, реализующего этот интерфейс.

Если тип T2 совместим по присваиванию с типом T1, говорят, что тип T2 ***неявно приводится*** к типу T1.

7.5 Пример: работа с таблицей

Имеется таблица, отображающая результаты успеваемости класса численностью 26 школьников за год по трем изучаемым предметам.

Ученик	Информатика			Математика			Физика		
	1 полугодие	2 полугодие	За год	1 полугодие	2 полугодие	За год	1 полугодие	2 полугодие	За год
Иванов Иван Иванович	4	4	4	5	4	4	5	3	4
Петров Петр Петрович	4	5	5	4	4	4	4	5	5
Валентинова Валентина Валентиновна	5	5	5	4	5	5	5	5	5
Сидоров Сидор Сидорович	4	4	4	4	5	5	4	4	4

Требуется создать пользовательский тип данных, пригодный для хранения представленной информации и создать набор процедур и функций, позволяющих вводить данные, выводить их и получать средний балл за каждый из периодов обучения в разрезе предметов и в целом – всего 12 средних баллов.

В структуре таблицы можно выделить следующие группы «заголовков»: по строкам - Ученик (по условию их 26 и все разные), по колонкам - Предмет (их три) и Период (их тоже три), поэтому колонок 3 × 3 = 9. На пересечении 26 строк с 9 колонками записаны 26 × 9 = 234 оценки и по каждой колонке будет нужно находить среднее. Если ориентироваться на строки, запись будет иметь 10 полей. Если на колонки – 26 полей. Выбор очевиден – единицей наших данных будет строка и мы назовем ее Ученик. Таблица будет представляться массивом из 26 таких строк.

Предмет и Период можно пронумеровать, как колонки, но давайте попробуем использовать перечислимые типы данных. Фамилию, имя и отчество ученика представим строкой string. Если понадобится, мы всегда сможем разбить строку на слова.

С иллюстративной целью в примере будет введено только четыре строки данных и будет создаваться динамический массив Класс из четырех элементов. Типы данных будут указаны с префиксом t (например, tПредмет), чтобы можно было пользоваться переменными Предмет, Период и Ученик.

Здесь мы впервые сталкиваемся с понятием «массив массивов». Имеются три предмета, каждый из которых содержит три оценки. Если ввести массив Оценка[ ] из трех элементов, содержащий оценки по Предмету и массив Предмет[ ] из трех элементов, задающий Период, каждый элемент Оценка[i] будет являться массивом Предмет[Период]. Получается массив Оценка[Предмет][Период].

```
// p07006
type
  tПериод = (Полугодие1, Полугодие2, Год);
  tПредмет = (Информатика, Математика, Физика);
  tУченик = record
    ФИО: string;
    Оценка: array[tПредмет] of array[tПериод] of integer;
  end;

procedure Ввод(var Ученик: tУченик);
begin
  Ученик.ФИО := ReadlnString('Фамилия, И.О. ученика:');
  Write('Введите 9 оценок через пробел: ');
  for var Предмет := Информатика to Физика do
    for var Период := Полугодие1 to Год do
      Read(Ученик.Оценка[Предмет][Период]);
  Readln; // очистить буфер перед последующим вводом строки
end;

function СреднийБаллДетально(const Класс: array of tУченик): array of real;
begin
  var n := Класс.Length;
  Result := ArrFill(9,0.0); // создание с обнулением
  for var Ученик := 0 to Класс.High do
    for var Предмет := Информатика to Физика do
      for var Период := Полугодие1 to Год do
        Result[3 * Ord(Предмет) + Ord(Период)] +=
            Класс[Ученик].Оценка[Предмет][Период];
  for var i := 0 to Result.High do
    Result[i] /= n;
end;
```

```
procedure ВыводСреднегоБаллаДетально(Баллы: array of real);
begin
  Println('Средние баллы');
  for var Предмет := Информатика to Физика do
    for var Период := Полугодие1 to Год do
      $'{Предмет} за {Период}: {Баллы[3 * Ord(Предмет) + Ord(Период)]}'.Println
end;

function СреднийБалл(const Класс: array of tУченик): real;
begin
  var n := Класс.Length;
  Result := 0;
  for var Ученик := 0 to Класс.High do
    for var Предмет := Информатика to Физика do
      for var Период := Полугодие1 to Год do
        Result += Класс[Ученик].Оценка[Предмет][Период];
  Result /= n * 9
end;

begin
  var n := ReadlnInteger('Число учеников:');
  var НашКласс := new tУченик[n];
  for var Ученик := 0 to n - 1 do
    Ввод(НашКласс[Ученик]);
  ВыводСреднегоБаллаДетально(СреднийБаллДетально(НашКласс));
  $'Средний балл по классу за год {СреднийБалл(НашКласс):f2}'.Println;
end.
```

Ниже представлен пример диалога с программой.

```
Число учеников: 4
Фамилия, И.О. ученика: Иванов Иван Иванович
Введите 9 оценок через пробел: 4 4 4 5 4 4 5 3 4
Фамилия, И.О. ученика: Петров Петр Петрович
Введите 9 оценок через пробел: 4 5 5 4 4 4 4 5 5
Фамилия, И.О. ученика: Валентинова Валентина Валентиновна
Введите 9 оценок через пробел: 5 5 5 4 5 5 5 5 5
Фамилия, И.О. ученика: Сидоров Сидор Сидорович
Введите 9 оценок через пробел: 4 4 4 4 5 5 4 4 4
Средние баллы
Информатика за Полугодие1: 4.25
Информатика за Полугодие2: 4.5
Информатика за Год: 4.5
Математика за Полугодие1: 4.25
Математика за Полугодие2: 4.5
Математика за Год: 4.5
Физика за Полугодие1: 4.5
Физика за Полугодие2: 4.25
Физика за Год: 4.5
Средний балл по классу за год 4.42
```

Сложно выглядит, не правда ли? И это с учетом возможности использования содержательных имен на кириллице и привлечением, где это получалось, современ-

ных средств программирования языка PascalABC.NET. В утешение можно только отметить, что на базовом Паскале программа с использованием перечислимого типа данных выглядит еще более громоздкой. Обратим внимание на некоторые детали в программе.

В процедуре Ввод( ) имеются три оператора ввода. Первый читает символьную строку и в нем использован Readln с последующей очисткой буфера ввода. Второй в цикле читает 9 целых чисел, которые предлагается ввести (для компактности) в строку через пробел. Здесь используется Read без очистки буфера ввода, поскольку Паскаль читает данные по нажатию Enter и очистка привела бы к тому, что было бы воспринято только значение одного числа (до первого пробела). Но это порождает проблему при следующем вызове процедуры. Неочищенный буфер ввода при чтении строки – это плохо и подробное объяснение уже приводилось в части 6. Поэтому в конце процедуры стоит оператор Readln без списка данных. Он тихо и незаметно очищает буфер ввода.

Как отмечалось выше, детальных средних баллов получается девять. Логично написать функцию, возвращающую массив из девяти элементов. Конечно, можно было придумать массив массивов с типом **real**, хранящих эти средние, подобно полю Оценка в записи типа tУченик, но смысла в этом нет, поскольку данные используются однократно. Для обращения к элементам массива пришлось составить функцию преобразования комбинации «Предмет» – «Период» в индекс от 0 до 8. Функция Ord( ) для перечислимого типа из трех элементов возвращает значения от 0 до 2, чти и дает для выражения 3 * Ord(Предмет) + Ord(Период) диапазон изменения от 0 до 8.

Интерполированные строки для вывода использованы для удобства. Неизбежное при использовании обычного списка вывода обилие кавычек и запятых удлинило бы строки программного кода, а разрыв строк вывода из-за переноса не добавляет им наглядности.

Можно ли написать программу короче? Можно, если … отказаться от перечислимого типа данных! Не зря в последнем абзаце предыдущей главы упоминалось, что данные этого типа используются сравнительно редко.

```
// p07007
type
  tУченик = record
    ФИО: string;
    Оценка: array of integer;
  end;

procedure Ввод(var Ученик: tУченик);
begin
  Ученик.ФИО := ReadlnString('Фамилия, И.О. ученика:');
  Ученик.Оценка := ReadArrInteger('Введите 9 оценок через пробел:', 9);
  Readln
end;
```

```
function СреднийБаллДетально(const Класс: array of tУченик): array of real;
begin
  var n := Класс.Length;
  Result := ArrFill(9, 0.0);
  for var Ученик := 0 to n - 1 do
    for var j := 0 to 8 do
      Result[j] += Класс[Ученик].Оценка[j];
  Result.Transform(t -> t / n);
end;

function СреднийБалл(const Класс: array of tУченик):=
    Класс.SelectMany(t->t.Оценка).Average;

procedure ВыводСреднегоБаллаДетально(Баллы: array of real);
begin
  Println('Средние баллы');
  var аПредмет := 'Информатика Математика Физика'.ToWords;
  var аПериод := 'Полугодие1 Полугодие2 Год'.ToWords;
  for var Предмет := 0 to 2 do
    for var Период := 0 to 2 do
      $'{аПредмет[Предмет]} за {аПериод[Период]}: {Баллы[3 * Предмет + Период]}'
        .Println
end;

begin
  var n := ReadlnInteger('Число учеников:');
  var НашКласс := new tУченик[n];
  for var Ученик := 0 to n - 1 do
    Ввод(НашКласс[Ученик]);
  ВыводСреднегоБаллаДетально(СреднийБаллДетально(НашКласс));
  $'Средний балл по классу за год {СреднийБалл(НашКласс):f2}'.Println;
end.
```

45 строк вместо 59. Достаточно существенное сокращение текста программы, почти на четверть. И текст понимать стало проще. Отметки по строке теперь представлены просто массивом из 9 элементов. А на группы «три по три» мы его разбиваем мысленно.

Отметим еще один прием работы с перечислимым типом данных, когда на его основе создается множество.

```
procedure Ввод(var Ученик: tУченик);
begin
  Ученик.ФИО := ReadlnString('Фамилия, И.О. ученика:');
  Write('Введите 9 оценок через пробел: ');
  foreach var Предмет in [Информатика..Физика] do
    foreach var Период in [Полугодие1..Год] do
      Read(Ученик.Оценка[Предмет][Период]);
  Readln;
end;
```

Здесь вместо цикла **for** использован цикл **foreach** по созданному «на лету» множеству. Разницы вроде бы никакой, но это только на первый взгляд. Вспоминайте: множество неупорядоченно. А это значит, что его элементы вовсе не обязательно будут перебираться в том порядке, который подразумевает перечислимый тип. Использовать в этой процедуре множество – заложить себе «бомбочку» с часовым механизмом, поставленным на неизвестное время. Отладка программ с подобными закладками – занятие не для слабонервных.

Использовать перечислимый тип данных или нет, и если использовать, то в каких случаях, решать вам.

7.6 Обобщенный тип

Обобщенным (generic) типом называется шаблон, на основе которого создается класс, запись или подпрограмма, имеющий в качестве параметров один или более типов данных. Подстановка конкретных типов на место параметров называется ***инстанцированием***. Параметры указываются после имени обобщенного типа в угловых скобках. При выведении требуется точное соответствие типов, **приведение типов не допускается**. По умолчанию с переменными, имеющими тип параметра обобщенного класса или подпрограммы, внутри методов обобщённых классов и обобщенных подпрограмм можно делать лишь ограниченный набор действий: **присваивать и проверять на равенство**. Это достаточно скромно. Ниже будут рассмотрены приемы, позволяющие делать с такими переменными некоторые дополнительные операции.

Рассмотрим пример. Пусть требуется написать функцию, принимающую массив и возвращающую в виде массива результат перестановки каждой пары соседних элементов, т.е. $a_1, a_0, a_3, a_2, a_5, a_4 \ldots a_k, a_{k-1} \ldots$

```
function Perm<T>(a: array of T): array of T; // p07008
begin
  Result := Copy(a);
  for var i := 0 to a.Length div 2 - 1 do
    Swap(Result[2 * i], Result[2 * i + 1]);
end;

begin
  var a := ArrRandom(9, 10, 99);
  a.Println; // 45 99 38 67 72 87 83 12 56
  var b := Perm(a);
  b.Println; // 99 45 67 38 87 72 12 83 56
  var c := SeqRandomReal(8, 10, 99).Select(p -> Round(p, 1)).ToArray;
  c.Println;       // 97.3 96.8 98.3 29.5 44.9 12 43.7 37.9
  Perm(c).Println; // 96.8 97.3 29.5 98.3 12 44.9 37.9 43.7
end.
```

Здесь функция Perm имеет один обобщенный параметр T. При первом вызове происходит инстанцирование с заменой T на **integer**, при втором – с заменой T на **real**.

Если при вызове требуется явно привести параметры к определенному типу, этот тип указывается в угловых скобках вместо T, а перед открывающей скобкой ставится экранирующий символ &, иначе знак «<» будет воспринят, как операция отношения. Например, запись вида MyFunction &<**real**>(1) означает приведение константы 1 к 1.0. Конечно, проще было написать MyFunction(1.0).

В случае использования нескольких параметров разного типа, можно указывать в описании MyProc<T, T1 ,T2, …>(…)

Внутри методов обобщённых классов и обобщенных подпрограмм параметры типа T можно лишь присваивать и проверять на равенство. Можно также использовать присваивание значения по умолчанию, используя конструкцию default(T) – значение по умолчанию для типа T. Переменная ссылочного типа получит значение nil, размерного – нулевое значение.

Разрешить использование некоторых действий с переменными, имеющими тип параметра обобщенного класса или подпрограммы, позволяет задание ограничений на обобщенные параметры, задаваемые в секции **where** после заголовка подпрограммы или класса. Подробный разбор таких ограничений выходит за рамки данной книги. Отметим лишь способ организации сравнений на неравенство.

$$F(a,b)=\begin{cases}1, & a>b\\ 0, & a=b\\ -1, & a<b\end{cases}$$

Пусть требуется написать функцию, Comp(a, b), возвращающую результат сравнения параметров типа T как значение типа **integer**. Проблема состоит в том, что по умолчанию мы не можем сравнивать параметры типа T на «больше» и «меньше» и такое сравнение нужно разрешить.

```
// p07009
function Comp<T>(a, b: T): integer;
  where T: IComparable<T>;
begin
  if a = b then Result := 0
  else
  if a.CompareTo(b) > 0 then Result := 1
  else Result := -1
end;

begin
  Comp(3, 5).Println;               // -1
  Comp(4.2, 2.7).Println;           //  1
  Comp&<real>(4, 5.2).Println;      // -1
  Comp('x', 'X').Println;           //  1
  Comp(False, True).Println;        // -1
  Comp('Корова', 'Собака').Println; // -1
  Comp(1.0, 1 + 1e-16).Println      //  0
end.
```

Если у вас возникли сомнения в полезности этой функции, подумайте о реализации, например, процедуры сортировки последовательности или массива элемен-

тов типа T – там без сравнения не обойтись. Кроме того, такая функция дает возможность организовать ветвление **case** по трем направлениям, поскольку в метках **case** могут быть только константы. Иногда **case** смотрится лучше вложенных **if**.

Несколько слов об IComparable<T>. Это один из стандартных ***интерфейсов***, реализуемых Microsoft .NET Framework. Об интерфейсах будет сказано в части 10. Для того, чтобы можно было воспользоваться функцией CompareTo, ее аргументы должны реализовывать обобщенный интерфейс IComparable<T>. Этот интерфейс реализован для числовых типов, а также **boolean**, **char** и **string**. Можно реализовать IComparable<T> в пользовательском типе и после этого данные такого типа можно будет сравнивать между собой. Это могут быть записи, массивы или что-то еще.

7.7 Указатели

Очень специфичная вещь. Новички в программировании часто испытывают трудности при понимании указателей и особенности работы с ними. В отличие от языков C/C++, где без указателей программисты даже мыслить не умеют, в отличие от базового Паскаля, в котором без указателей нельзя строить динамические структуры данных, в языке PascalABC.NET роль указателей, как типа, весьма незначительна. В базовом Паскале указатели имеют тип, называемый ссылочным. В .NET-языках этот тип зарезервирован для другого понятия, поэтому указатели в PascalABC.NET имеют тип «указатели».

Внешне в указателях нет ничего загадочного – это всего лишь область памяти, хранящая адрес. В PascalABC.NET указатель может быть ***типизированным*** или ***бестиповым***. Типизированный указатель содержит адрес области памяти, предназначенной для размещения данных определенного типа, а для бестипового указателя тип данных заранее не определен. Типизированный и бестиповый указатель можно присваивать друг другу.

Для объявления типизированного указателя используется символ caret («крышечка»):

```
var
  p1: ^integer; // указатель на тип integer
  ptr: ^МойТип; // указатель на тип МойТип
```

Бестиповый указатель объявляется с использованием ключевого слова **pointer**:

```
var p: pointer;
```

В этой книге мы пока еще ни разу не опускались до «низменной возни с адресами памяти», поэтому может возникнуть вопрос о том, как получить этот самый адрес. Получить его позволяет операция, называемая «взятие адреса». Для нее зарезервирован символ @ с неудобным для русского языка названием «коммерческое эт». Он же – «собака», «обезьяна», … и даже какой-то «кракозяблик».

```
begin
  var i := 3;
  var p := @i; // автовыведение типа ^integer
  Writeln(p);  // $23ED24 - как пример
end.
```

В приведенном примере адрес переменной i теперь находится в указателе p. Можно говорить, что p теперь указывает на i. Хорошо, указали, но какой в этом смысл?

Смысл использованию указателей придает операция ***разыменования***, позволяющая обращаться к содержимому памяти по адресу, который хранится в типизированном указателе. Операция разыменования использует тот же знак, что и описание типизированного указателя, только записывается он после имени указателя. Разыменовываемый указатель может появиться и в левой части оператора присваивания.

```
begin
  var i := 3;
  var p := @i;  // автовыведение типа ^integer
  p^ := p^ * 2 + 1;
  Writeln(i);   // теперь i = 7
end.
```

Рассмотрим, как получилось значение 7. Указатель p хранит адрес переменной i, поэтому теперь разыменованный p является просто еще одним именем для области памяти, названной i. Исходя из этого можно оператор p^ := p^ * 2 + 1 переписать как i := i * 2 + 1, что при i, равном трем, дает новое значение i, равное семи.

Так что же нам дал указатель? Всего лишь, достаточно низкоуровневую возможность создать для переменной некий синоним имени. Да, но у нас для этой цели уже есть ссылки! Именно поэтому роль указателей в PascalABC.NET весьма незначительна. Фактически, они оставлены лишь в целях совместимости с базовым Паскалем. Весьма экзотический для нынешних времен способ писать программу.

Указатели позволяют в базовом Паскале создавать структуры данных, такие как стеки, очереди, списки, деревья и им подобные. Особенность таких структур состоит в том, что в них каждый элемент связан с одним или более таких же элементов. Эта связь организуется при помощи указателей. В PascalABC.NET для подобных структур данных создаются классы, в которых организация связей легко реализуется посредством ссылок.

> В языке Паскаль запрещается определять один тип данных посредством другого, который описан ниже. Но для указателей сделано исключение: они могут ссылаться на еще не описанный тип.

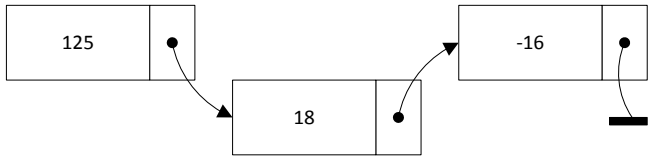
Рассмотрим пример подобного описания.

```
type
  УказательНаУзел = ^Узел;
  Узел = record
    Значение: integer;
    Связь: УказательНаУзел
  end;
```

Здесь тип УказательНаУзел – это указатель на запись типа Узел, а тип Узел содержит поле Связь, которое имеет тип УказательНаУзел. Теперь можно создавать переменные типа Узел и связывать их друг с другом в различного рода цепочки посредством поля Связь.

> Ввиду особенностей платформы .NET типизированный указатель не должен быть ссылочного типа или содержать ссылочные типы на каком-либо подуровне (например, поля записи не должно иметь ссылочный тип). Исключение из этого правила сделано для динамических массивов и строк.

В качестве примера рассмотрим реализацию с помощью указателей так называемого односвязного списка из трех элементов. Односвязным он называется потому, что между элементами списка устанавливается одна связь, а именно, со следующим элементом.



```
type // p07010
  УказательНаУзел = ^Узел;
  Узел = record
    Значение: integer;
    Связь: УказательНаУзел
  end;

begin
  var Начало, ТекущийУказатель, ПредыдущийУказатель: УказательНаУзел;
  new(ТекущийУказатель); // создаст запись и инициализирует указатель
  ТекущийУказатель^.Значение := 125;
  ТекущийУказатель^.Связь := nil; // пока связи нет.
  ПредыдущийУказатель := ТекущийУказатель; // запомнили
  Начало := ТекущийУказатель;
  foreach var v in Seq(18, -16) do
  begin
    new(ТекущийУказатель);
    ТекущийУказатель^.Значение := v;
    ТекущийУказатель^.Связь := nil;
    ПредыдущийУказатель^.Связь := ТекущийУказатель; // а вот и связь!
    ПредыдущийУказатель := ТекущийУказатель
  end;
  // выведем значение всех элементов (125 18 -16)
  var Элемент := Начало^; // первый
  repeat
    Элемент.Значение.Print;
    Элемент := Элемент.Связь^
  until Элемент.Связь = nil;
  Элемент.Значение.Println
end.
```

Здесь важно понимать, что переменная типа *УказательНаУзел*, за которой следует «крышечка», является синонимом переменной типа *Узел*. Это хорошо видно на примере оператора **var** Элемент := Начало^.

Пример дан исключительно в демонстрационных целях. При создании списка использовались указатели, а при работе со списком – ссылка на запись.

7.8 Тип данных BigInteger

Этот целочисленный тип данных уже использовался в предыдущих частях книги. Работа с данными типа **BigInteger** достаточно специфична, поэтому рассматривается отдельно.

Тип данных **BigInteger** отображается на тип System.Numerics.BigInteger библиотеки Microsoft .NET Framework, позволяющий записывать и обрабатывать целые числа практически неограниченной длины.

Для данных типа **BigInteger** реализованы арифметические операции сложения, вычитания, умножения и деления. Операция деления, в отличие от других цело-

численных типов данных, возвращает не вещественное значение, а результат целочисленного деления, имеющий тип **BigInteger**. Остаток целочисленного деления a на b можно получить с помощью традиционной для языка Паскаль операции a mod b.

С операцией возведения в степень дела обстоят немного сложнее. Функция Power, а также ее синоним, операция **, не работают с типом **BigInteger**. Возвести значение типа **BigInteger** можно только в степень с показателем, приводящимся к типу **integer**, для чего используется вызов статической функции Pow из библиотеки .NET:

```
BigInteger.Pow(Основание, ПоказательСтепени)
```

В этой библиотеке есть немало полезных функций, поэтому при необходимости серьезной работы с «длинной арифметикой», имеет смысл их изучить. В частности, обратите внимание на:

- `BigInteger.GreatestCommonDivisor(a, b)` – НОД чисел a и b;
- `BigInteger.ModPow(p, q, k)` – остаток от целочисленного деления на k значения p в степени q;
- `BigInteger.Log(a)` – натуральный логарифм a.

7.8.1 Инициализация данных типа BigInteger

Казалось бы, какие тут могут быть проблемы: в операторе присваивания слева указываем имя переменной типа **BigInteger**, справа – литерал, изображающий нужное значение. Пока значение не превышает **int64**.MaxValue, т.е. в нем не больше 19 цифр – действительно, никаких проблем. А вот если цифр больше, то компилятор такое значение забракует, поскольку не сможет создать целочисленную константу необходимого размера. Выход – использовать строковое представление числа. Но тогда придется строку приводить к типу **BigInteger**.

```
var a := BigInteger.Parse('12345678901234567890123456789O');
```

Метод .Parse предполагает, что строка содержит корректное изображение целого числа, возможно со знаком. В случае, если это не так, будет сгенерировано исключение и вы получите сообщение об ошибке, причем произойдет это не при компиляции, а во время выполнения программы: «Ошибка времени выполнения: Не удалось выполнить синтаксический анализ значения».

Конечно, в реальной ситуации написать ерунду в литерале можно разве что себе назло. А вот получить некорректное значение при клавиатурном вводе вполне возможно.

```
var a := BigInteger.Parse(ReadlnString);
```

Для обработки подобной ситуации следует использовать другой метод:

```
var a: BigInteger;
if not BigInteger.TryParse(ReadlnString, a) then
begin
  Print('Неверный ввод');
  exit
end;
```

Преобразовать целочисленное значение или выражение в типу **BigInteger** можно при помощи явного приведения, например **BigInteger**(0). В случае вещественного типа нужно сделать предварительное приведение к целочисленному типу.

7.8.2 Приведение BigInteger к другому типу

Попытка явно привести значение типа **BigInteger** к другому целочисленному типу может привести к возникновению исключения на этапе выполнения программы. Действительно, количество цифр может оказаться чрезмерным и тогда приведение приведет к потере точности, что очень нежелательно в целочисленной арифметике. В то же время, приведение к вещественному типу исключения не вызовет, поскольку сам факт использования данных такого типа свидетельствует о том, что программист заранее готов в какой-то степени жертвовать точностью.

```
begin // p07011
  var a := BigInteger.MinusOne; // это -1
  var b := integer(a);
  a := BigInteger.Parse('123456789012345678901234 5');
  Print(b, real(a)) // -1 1.23456789012346E+24
  var s := a.ToString; // к строке приводим обычным методом
  Println(s) // 123456789012345678901234 5
end.
```

7.9 Тип данных decimal

Еще один числовой данных, базирующийся на библиотеке Microsoft .NET Framework. Отображается на вещественный тип данных System.Decimal. Точность представления 28-29 цифр. Запись значений производится в формате с фиксированной точкой. Используется, если число нецелое, а точности в 16 знаков, которые дает тип **real**, недостаточно.

Для типа **decimal** определены четыре стандартные операции арифметики: сложение, вычитание, умножение и деление. Операнды целочисленных типов, за исключением **BigInteger**, в арифметических выражениях автоматически приводятся к типу **decimal**. Операнды вещественного типа нужно приводить явно.

```
begin // p07012
  var a, b, c, d: decimal;
  a := 10;
  b := a + 3;
  // c := 2.3; ошибка преобразования при компиляции
  c := decimal(2.3);
  d := c + decimal(1.5);
  Println(a, b, c, d) // 10 13 2.3 3.8
end.
```

Как и в случае с **BigInteger**, значения с большим количеством цифр обеспечиваются парсингом строк: a := **decimal**.Parse('1234567890123456.78901').

Из множества библиотечных функций отметим наиболее употребительные (как обычно, полный перечень вы можете получить в среде PascalABC.NET, введя точку после decimal):

- **decimal**.Ceiling(a) – ближайшее целое значение в сторону +∞;
- **decimal**.Floor(a) – ближайшее целое значение в сторону -∞;
- **decimal**.Reminder(a, b) – остаток деления a на b (может быть и нецелым);
- **decimal**.Round(a) – арифметическое округление значения a к целому;
- **decimal**.ToDouble(a) – преобразование значения a к типу **real**;
- **decimal**.ToInt32(a) – преобразование значения a к типу **integer**;
- **decimal**.ToInt64(a) – преобразование значения a к типу **int64**;
- **decimal**.Truncate(a) – целая часть значения a.

7.10 Тип данных DateTime

Введенный в Microsoft .NET Framework тип данных для работы с датой и временем. Отображается на тип System.DateTime, во внутреннем формате которого данные представляются количеством «тиков» – интервалов времени длиной в 0.1 микросекунду, прошедших после полуночи 1 января 1 года. Тип данных являются записью. Однажды созданную дату изменить нельзя, но можно присвоить переменной другое значение.

Для начала попробуем вывести текущие дату и время:

```
begin
  Println(DateTime.Now) // 12/21/2018 5:40:51 PM
end.
```

В целом понятно, но хорошо бы получить результат в более привычном формате 21.12.2018 17:40:51. Ну что же, «Подарок от фирмы Microsoft – в студию!»

```
begin // p07013
  Println(DateTime.Now.ToString('',
     System.Globalization.CultureInfo.GetCultureInfo('ru-RU')))
end.
```

Если так обстоят дела с выводом готового значения, - подумаете вы, - можно представить, какие чудеса ждут нас при создании данных этого типа! И не сильно

погрешите против истины. Безусловно, в Microsoft об Америке побеспокоились и дату-время американцы могут вводить в привычном формате.

```
begin
  var dt1 := DateTime.Parse('12/21/2018 7:02 PM'); // 21.12.2017 19:02:00
  var dt2 := new DateTime(2018, 12, 21, 19, 3, 0); // 21.12.2017 19:03:00
  Println(dt1, dt2); // 12/21/2018 7:02:00 PM 12/21/2018 7:03:00 PM
end.
```

Понятно, что если организовать запрос ввода строкой, то придется вводить именно по типу 12/21/2018 7:02 PM. С высокой степенью вероятности это породит при вводе массовые ошибки, даже если ввод сопровождать разъясняющим приглашением. На помощь снова приходит «культура» (CultureInfo):

```
var dt3 := DateTime.ParseExact(ReadlnString('дд.мм.гггг='),
   'd', System.Globalization.CultureInfo.GetCultureInfo('ru-RU'));
```

Дату вводим в привычном формате: 21.12.2018. Но представьте себе школьника, который на ЕГЭ по памяти пишет такой оператор…

Вопреки ожиданиям, работать с этим типом данных довольно просто. Получить день, месяц, год, часы, минуты, секунды и миллисекунды для переменной dt типа **DateTime** можно при помощи обращения к соответствующим свойствам dt.Year, dt.Month, dt.Day, dt.Hour, dt.Minute, dt.Second, dt.Millisecond. Имеются и другие свойства, которые вы можете обнаружить «по точке» в среде PascalABC.NET. Получить на основе имеющейся даты (времени) необходимое смещение во времени можно при помощи функций, добавляющих некоторую величину. Например, dt.AddDays(3) увеличит дату 3 дня, dt.AddYears(-2) – уменьшит дату на 2 года.

Если из одной отметки времени вычесть другую, должен получиться временной интервал. Когда мы это делаем вручную, то приводим временные отметки к некоторой единице, например, к дням, а потом находим разность. В .NET для подобной разности существует специальный тип System.TimeSpan. Свойства объекта этого типа позволяют получать интервал в необходимых единицах не крупнее дней. Например, .TotalDays возвращает интервал в днях, .TotalHours – в часах. Это позволяет находить временные интервалы в виде разности двух переменных типа **DateTime**.

Чтобы узнать, какая дата будет через 157 дней, нужно всего лишь увеличить количество дней в текущей дате на 157.

```
begin // p07014
  var d2 := DateTime.Today.AddDays(157);
  Println($'{d2.Day}.{d2.Month}.{d2.Year}, {d2.DayOfWeek}');
end.
```

Для текущей даты 07.01.2019 на выводе будет получена строка 13.6.2019, Thursday.

Как видите, если предполагается большой объем работ с датой и временем, есть смысл подумать о написании для этой цели собственного класса на базе **DateTime**.

7.11 Для самостоятельного решения

T7.1. Опишите запись, пригодную для инициализации массива среднесуточных температур с точностью 0,1 в диапазоне [Tmin; Tmax] за указываемое число суток. Значения температуры формируются при помощи датчика случайных чисел. Создайте запись описанного типа для 30 суток с диапазоном [13,7; 24], выведите значения полученного набора температур и значение средней температуры за указанный период.

T7.2. (Сложная) Реализуйте функцию сложения положительных десятичных чисел A и B, имеющих целую (ЦЧ) и дробную (ДЧ) часть с произвольным количеством цифр. Можно, например, найти отдельно значения Счц = Ацч + Вцч и Сдч = Адч + Вдч, а затем при необходимости выполнить перенос старшего разряда из Сдч в Сцч. Необходимо учесть, что дробная часть может начинаться с одного и более нулей, что создаст определенные проблемы с выравниванием слагаемых в дробной части. Результат должен выводиться в привычном виде.

С помощью этой функции выполните следующие примеры:

1) `64828163049832654452517130787222 81,202479108450305936595 + 9583703050575718123003,719290875762843674731031067743 2;`
2) `24353915677471938471515052140237420849035,352023 + 5962999145,6967858008690949913771158672 62;`
3) `62773040926651449548308363519209375020834809 3 + 0,795713648375157400281354681;`
4) `0.65 + 0.59;`
5) `1,9999 + 0,001;`
6) `1,2 + 0.00015;`
7) `875762843674731031067743 + 3265445251713078722;`
8) `0,0000001 + 0,00003;`
9) `0,999999999 + 0,000000001`

Приведенные данные можно взять из файла T7.2.txt.

T7.3. Используя тип данных DateTime, напишите функцию, возвращающую краткое наименование дня недели (Пн, Вт, … Вс) для указанной даты. Способ и формат задания даты выберите самостоятельно.

Примечание: Необходимую информацию можно, например, найти в Интернет при поиске по контексту «System.DateTime получить день недели».

T7.4. Воспользовавшись материалом из предыдущего задания, напечатайте для указанного года список дат, приходящихся на пятницы. Возможно, потребуется дополнительное обращение к Интернет.

Часть 8

Многомерные массивы

— У Константина Константиновича девяносто четыре родителя пяти различных полов, девяносто шесть собрачников четырех различных полов, двести семь детей пяти различных полов и триста девяносто шесть соутробцев пяти различных полов.

А. Стругацкий, Б. Стругацкий.
«Сказка о тройке»

В части 5 были рассмотрены одномерные массивы. Но массив может иметь и большее число измерений.

Если M(x,y) - точка на плоскости, в декартовой системе координат она имеет две координаты. Точка M(x,y,z) имеет три координаты. Если рассмотреть положение точки в трехмерном пространстве, оно может оказаться изменяющимся во времени t, и точка M(x,y,z,t) будет описываться четырьмя координатами. Плоская или объемная фигура – это совокупность точек, так что она в рассмотренных случаях будет описываться двухмерным, трехмерным или четырехмерным массивом. Можно придумать варианты, когда понадобится массив и с еще большим числом измерений. Из многомерных массивов наиболее популярны двухмерные (можно также писать двумерные) массивы. В PascalABC.NET двухмерные динамические массивы называются ***матрицами***.

Матрица имеет два измерения, называемые строками и столбцами по аналогии с матрицами в математике. Матрица всегда имеет прямоугольную форму: количество элементов в каждой строке постоянно и количество элементов в каждом столбце тоже постоянно. Внешне матрицу всегда можно представить в виде таблицы.

Пусть имеется матрица, состоящая из m строк и n столбцов. В этом случае говорят, что она имеет размер m×n. Не путайте размер с размерностью: в матрице размерность (число измерений массива) равна двум. Если элемент матрицы A находится на пересечении строки i и столбца j, его записывают как $A_{i,j}$ или A[i, j].

8.1 Динамические двухмерные массивы (матрицы)

Как вы уже знаете, массивы бывают статические и динамические. Размеры (т.е. количество элементов) по каждому измерению статического массива должны быть известны компилятору, иначе он не сможет зарезервировать нужный объем памяти компьютера. В случае динамического массива мы сообщаем компилятору лишь количество измерений массива. Конечно, в любом случае, мы сообщаем еще и тип элементов массива. Динамический массив создается во время работы программы, и тогда же определяются его размеры. Процедура SetLength, с помощью которой можно в процессе выполнения программы задать или поменять размер одномерного массива, работает и для двухмерных массивов. Но если с первоначальным заданием размера массива все просто, то с его переопределением есть проблема. Мы разберемся с этим позже.

8.1.1 Описание и создание матриц

Пусть у нас имеется двухмерный массив размером 4×3. В PascalABC.NET динамические массивы нумеруются с нуля, поэтому получаются четыре строки с номерами от 0 до 3 и три столбца, нумерованные от 0 до 2.

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \end{pmatrix}$$

Каждую строку двухмерного массива можно представить, как обычный массив. Массив из таких строк образует массив массивов. Построить его несложно.

```
begin
  var A: array of array of integer;
  var (m, n) := (4, 3); // число строк и столбцов
  SetLength(A, m); // распределим память под m строк
  for var i := 0 to m - 1 do
    SetLength(A[i], n); // в каждой строке создадим массив из n элементов
  A[3][2] := 43; // строка с индексом 3, в ней элемент индексом 2
  A[1, 0] := 21; // строка с индексом 1, в ней элемент индексом 0
  Writeln(A) // [[0,0,0],[21,0,0],[0,0,0],[0,0,43]]
end.
```

При обращении к элементу такого «двухмерного массива» допускается запись как двух индексов по отдельности, так и совместно. Можно обойтись без этих нагромождений, описав «нормальный» двухмерный массив.

```
begin
  var A: array [,] of integer; // обратите внимание на запятую
  var (m, n) := (4, 3); // число строк и колонок
  SetLength(A, m, n);
  A[3, 2] := 43; // писать A[3][2] тут не допускается
  A[1, 0] := 21;
  Writeln(A) // [[0,0,0],[21,0,0],[0,0,0],[0,0,43]]
end.
```

Для двухмерных массивов, подобно одномерным, можно совмещать описание с выделением памяти. Для этого массив создается с использованием ключевого слова **new**.

```
var a: array [,] of integer := new integer[4, 3]; // с описанием
var b := new real[4, 3]; // с автовыведением типа
```

Можно также выполнить инициализацию, добавив ***конструктор массива***.

```
var a := new integer[4, 3] ((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));
var b := new real[2, 3] ((2.1, 3.7, 5), (1, 2, 3));
var d: array of array of integer := ((1, 2, 3), (4, 5), (6, 7, 8));
```

После такой инициализации массив *a* будет содержать следующие значения

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

Конструктор двухмерного массива имеет два уровня скобок. Первый (внешний) уровень ограничивает сам конструктор, второй (внутренний) – ограничивает каждую строку. В приведенном примере массив a имеет четыре строки и три колонки. Соответственно, конструктор содержит четыре группы по три значения в каждой. Пример с инициализацией массива *b* напоминает нам о том, что заданные в конструкторе значения автоматически приводятся к нужному типу.

Массив *d* инициализируется как массив массивов. Отметьте, что в данном случае конструктор позволяет создать и инициализировать непрямоугольный массив: во второй строке два элемента, а в остальных по три. К элементу массива *d* можно также обращаться в виде d[i][j] или d[i, j]. Тем не менее, у массивов *d* и *b* разный тип данных. А это означает что их, например, нельзя передавать в подпрограммы один вместо другого.

8.1.2 Генерация матриц

Генерация матриц подразумевает присваивание элементам матрицы значений, задаваемых генератором псевдослучайных чисел или некоторым выражением. В случае, когда нужно присвоить всем элементам матрицы одинаковые значения, говорят о ***заливке*** матрицы. Генерация матриц в PascalABC.NET подразумевает вызов некоторой функции, возвращающей двухмерный динамический массив необходимых размеров. Это позволяет заранее не описывать матрицы, а пользоваться автоматическим выведением типа.

8.1.2.1 Заполнение случайными значениями

Имеются два генератора, заполняющие матрицу целыми, либо вещественными значениями.

- MatrRandom(m, n, a, b) – заполнение матрицы размера m×n целыми числами из интервала [a; b]. Имеется синоним MatrRandomInteger ;
- MatrRandomReal(m, n, a, b) – заполнение матрицы размера m×n вещественными числами из интервала [a; b].

Имеются следующие значения параметров, принимаемые по умолчанию: m=5, n=5, a=0, b=100.

Следующий пример иллюстрирует создание матриц, а также удобный способ организации их вывода при помощи расширения .Print. Традиционно, это расши-

рение «пропускает через себя» данные, сохраняя их тип. Поэтому, если у вас «синдром одной строки», можете смело встраивать .Print в цепочку.

```
begin // p08001
  var a := MatrRandom(6, 9, -50, 50);
  a.Println;
  Println;
  var b := MatrRandomReal(4, 3, -5, 5);
  b.Println
end.
```

Отметим аккуратный вывод, который дает .Print. По умолчанию под вывод целочисленного значения отводятся четыре позиции, под элемент вещественного типа – семь позиций, в двух из которых размещается дробная часть.

```
  39   8  50 -21  15 -42  12  25  19
 -38 -35   7 -37  27   4 -12  44 -12
   0 -49  45  13  11 -12 -43  46  39
  -7  19  18  46 -46  -8  40 -49 -13

  -4.30  -1.49  -4.66
   0.62   3.09  -0.52
  -4.76   3.59   1.23
   1.63   1.03   4.25
```

Если нужна иная разметка вывода, можно указать количество позиций явно.

```
begin // p08002
  MatrRandom(6, 9, -50, 50).Println(6);
  Println;
  MatrRandomReal(4, 3, -5, 5).Println(11, 7)
end.
```

```
19   -32     6    27    35    40   -22    10    30
23   -18   -35    32    16   -41    34    32    48
33   -31   -16    25    -1    34    42   -11    18
-1    11   -23    -5   -17    -9    43    29    47
 2   -16   -48   -23     2     7    18   -47   -39
26    -2   -43    -2   -28   -20    -7    36    47

 -3.5468194  4.4170922  2.0108772
  2.2497611  3.4569828 -0.4248126
 -2.7983750  2.9217856  4.9491709
  0.2331057 -2.2594955  4.9472349
```

8.1.2.2 Заполнение фиксированным значением

MatrFill(m, n, x) возвращает матрицу размера m×n, заполненную значением выражения x. Тип элементов матрицы будет совпадать с типом значения x. Например, вызов MatrFill(5, 5, '*') вернет матрицу размером 5×5 типа char, каждый элемент которой будет содержать звездочку.

8.1.2.3 Заполнение значениями, зависящими от индексов

MatrGen(m, n, (i, j) -> f(i, j)) возвращает матрицу размера m×n, каждый элемент которой заполняется значением некоторой функции от своих индексов. Например, квадратная матрица с единичными элементами на диагонали от $A_{0,0}$ до $A_{5,5}$ и нулями в остальных элементах строится вызовом

```
var a := MatrGen(6, 6, (i, j) -> i = j ? 1 : 0);
```

А вот так получается таблица умножения (привет, Турбо Паскаль!):

```
begin // p08003
  MatrGen(10, 10, (i, j) -> (i + 1) * (j + 1)).Println
end.
```

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

8.1.2.4 Заполнение на основе одномерного массива

- Matr(m, n, a) – заполнение матрицы размера m×n значениями элементов одномерного массива a. Предполагается, что этот массив имеет длину m×n. Заполнение матрицы производится в порядке по строкам;
- Matr(m, n, $a_{0,0}$, $a_{0,1}$, … $a_{m-1,n-1}$) – заполнение матрицы размера m×n перечисленными значениями элементов.

При несовпадении количества предлагаемых для заполнения элементов с размером матрицы генерируется исключение с выдачей сообщения «Ошибка времени выполнения: Количество инициализирующих элементов не совпадает с количеством элементов матрицы»

Пример. Заполнение матрицы размером 3×8 первыми 24 числами Фибоначчи.

```
begin // p08004
  var a := Matr(3, 8, ArrGen(24, 1, 1, (i, j)-> i + j)).Println(6)
end.
```

```
     1      1      2      3      5      8     13     21
    34     55     89    144    233    377    610    987
  1597   2584   4181   6765 10946 17711 28657 46368
```

Подобный прием удобно использовать для оперативного формирования и вывода различных таблиц.

8.1.3 Клавиатурный ввод значений элементов матриц

В базовом Паскале имеется единственный способ ввести значения элементов матрицы: организовать перебор всех элементов во вложенных циклах с последовательным присваиванием введенного значения очередному элементу. Порядок ввода можно задать как по строкам (второй индекс меняется быстрее первого), так и по столбцам (первый индекс меняется быстрее).

```
begin
  var (m, n) := (3, 2);
  var a := new real[m, n];
  for var i := 0 to m - 1 do
    for var j := 0 to n - 1 do
      Read(a[i, j]);
  a.Println
end.
```

```
2.5 -1.4 3.9 0.15 2 0.03
   2.50  -1.40
   3.90   0.15
   2.00   0.03
```

Чтобы каждый раз не писать эти типовые вложенные циклы, в PascalABC.NET введены функции ReadMatrInteger(m, n) и ReadMatrReal(m, n), возвращающие матрицу размера m×n типа **integer** или **real** соответственно, заполненную принятыми с клавиатуры значениями. Для прочих типов данных придется пользоваться приведенным выше решением на базе вложенных циклов.

```
begin
  var (m, n) := (3, 2);
  var a := ReadMatrReal(m, n);
  a.Println
end.
```

Поскольку массив динамический, компилятор заранее не распределяет под него память. Это позволяет не фиксировать в программе значения m и n, а вводить их с клавиатуры или получать каким-либо иным путем.

8.1.4 Вывод матриц (.Print)

Расширение .Print (и его разновидность .Println) для матриц имеют некоторые особенности. Во-первых, они возвращают не последовательность, а матрицу тех же размеров. Во-вторых, необязательным параметром здесь является не строка-разделитель, а количество позиций, отводимых под длину поля для выводимого элемента. И главное – вывод осуществляется построчно с сохранением формы матрицы.

- .Print(n) – для всех типов элементов матриц, кроме вещественных, выводит каждый элемент в n позициях. По умолчанию n=4 ;
- .Print(n, k) – для вещественных элементов отводит под вывод значения n позиций, сохраняя k знаков в дробной части (с округлением). По умолчанию n=7, k=2.

Недокументированная способность расширения .Print возвращать после вывода исходную матрицу дает возможность компактно записывать алгоритмы, встраивая .Println в цепочку, но в этом есть некоторое «шаманство». Оно отчасти допустимо для учебных задачек, а в практику программирования входить не должно.

В отличие от базового Паскаля, не допускающего указывать имя массива в операторе вывода Write, PascalABC.NET позволяет указывать в качестве параметров Write/Print любые массивы. Следует заметить, что получаемый в этом случае вывод пригоден в основном для целей отладки.

8.1.5 Переопределение размеров матрицы

Переопределить размеры матрицы можно двумя способами. Первый состоит в определении новой матрицы такого же типа в правой части оператора присваивания. Поскольку матрица – это динамический массив, левая и правая части оператора присваивания будут совместимы, что позволит передать ссылку на вновь созданную матрицу. Второй способ состоит в использовании процедуры SetLength.

Рассмотрим использование процедуры SetLength изменяющей размер матрицы с 3×4 на 2×7. Первоначально в матрице содержится 12 элементов, а после изменения размеров их станет 14, поэтому значения двух элементов не будут определены. Но мы помним, что все данные числового типа, не получившие значений, в .NET будут инициализированы нулями.

Казалось бы, достаточно просто задать новые размеры матрицы при помощи SetLength (мы так уже поступали с одномерными массивами) – и задача решена. Посмотрим, так ли это:

```
begin // p08005
  var a := MatrGen(3, 4, (i, j)-> 10 * (i + 1) + j + 1);
  a.Println;
  Println;
  SetLength(a, 2, 7);
  a.Println
end.
```

Результат оказывается вовсе не таким, как обычно ожидается.

```
 11  12  13  14
 21  22  23  24
 31  32  33  34

 11  12  13  14   0   0   0
 21  22  23  24   0   0   0
```

Если визуально наложить полученную матрицу на исходную, станет хорошо видно, что сохранились значения элементов, принадлежащие общей части обеих матриц. Прочие элементы оказались обнулены.

Секрета тут нет. Матрицы в памяти компьютера хранятся построчно, т.е. второй индекс у элементов увеличивается быстрее первого. Для исходной матрицы в памяти были значения (11 12 13 14) (21 22 23 24) (31 32 33 34). После того, как SetLength перераспределила память, в каждой строке стало семь элементов вместо четырех. Первая строка превратилась в (11 12 13 14 0 0 0), вторая – в (21 22 23 24 0 0 0), а третьей строки в новой матрице просто нет.

Но все же, как быть, если нам надо создать новую матрицу без потери данных? Для этого нужно воспользоваться способом, который был упомянут первым. Превратить матрицу в одномерный массив, а потом с помощью Matr (см. 8.1.2.4) сформировать из него двумерный массив. И выполнить присваивание.

```
begin // p08006
  var a := MatrGen(3, 4, (i, j)-> 10 * (i + 1) + j + 1);
  a.Println;
  Println;
  a := Matr(2, 7, a.ElementsByRow.ToArray + Arr(0, 0));
  a.Println
end.
```

```
11  12  13  14
21  22  23  24
31  32  33  34

11  12  13  14  21  22  23
24  31  32  33  34   0   0
```

Здесь пришлось немного забежать вперед, и использовать расширение .ElementsByRow. Отметим пока, что оно позволяет получить из матрицы последовательность значений элементов в порядке по строкам.

8.1.6 Получение сведений о текуших размерах матрицы

- a.RowCount – расширение, возвращающее количество строк в матрице;
- a.ColCount – расширение, возвращающее количество столбцов в матрице.

Очень полезная вещь в процедурах и функциях. Позволяет не передавать в них лишние параметры. Да и в цикле перебрать элементы строки от 0 до a.ColCount-1 тоже вполне удобно.

8.1.7 Выборка элементов матрицы

Выбирать можно строку, столбец и все элементы в порядке прохода матрицы по строкам или столбцам. Результат может быть одномерным массивом или последовательностью. Все эти операции обеспечивает набор расширений.

- a.Col(k) – возвращает в виде массива колонку матрицы a с номером k (отсчет номеров ведется от нуля);
- a.ColSeq(k) – возвращает в виде последовательности колонку матрицы a с номером k;

- a.Cols – возвращает последовательность колонок матрицы, в которой каждая колонка, в свою очередь, является последовательностью;
- a.Row(k) – возвращает в виде массива строку матрицы a с номером k (отсчет номеров ведется от нуля);
- a.RowSeq(k) – возвращает в виде последовательности строку матрицы a с номером k;
- a.Rows – возвращает последовательность строк матрицы, в которой каждая строка, в свою очередь, является последовательностью;
- a.ElementsByCol – возвращает последовательность элементов матрицы, выбирая их по колонкам;
- a.ElementsByRow – возвращает последовательность элементов матрицы, выбирая их по строкам;
- a.ElementsWithIndices – возвращает последовательность трехэлементных кортежей, в которой каждый элемент формируется на основе Aij и имеет структуру вида (значение, индекс i, индекс j);
- a.MatrSlice(ar, ac) – возвращает новую матрицу, строки которой выбираются из исходной матрицы по индексам строки, находящимся в массиве ar, а столбцы – по индексам столбцов, находящимся в массиве ac;
- a.MatrSlice(fr, tr, fc, tc) – возвращает новую матрицу, строки которой выбираются из исходной матрицы по индексам от fr до tr, а колонки – индексам от fc до tc (срез матрицы).

Ниже в демонстрационных целях приводится программа, делающая различные операции с матрицами.

```
function NewMatrix(m, n: integer): array[,] of integer; // p08007
begin
  Result := MatrRandom(m, n, -50, 50);
  Result.Println(4);
  Println('-' * 4 * n)
end;

begin
  var a := NewMatrix(3, 5);
  var b := NewMatrix(2, 7);
  Println('Min(A) =', a.ElementsByRow.Min);
  Println('Среднее по B:', b.ElementsByRow.Average);
  Print('Сумма по колонкам в A:');
  a.Cols.Select(t -> t.Sum).Println;
  Print('Сумма кубов модулей элементов последней строки A:');
  a.RowSeq(a.RowCount - 1).Select(t -> Abs(t ** 3)).Sum.Println;
  Print('Максимальный элемент в B:');
  var max := b.ElementsWithIndices.MaxBy(t -> t[0]);
  $'B[{max[1]+1},{max[2]+1}] = {max[0]}'.Println;
  Writeln('Выборка строк и столбцов матрицы A по индексам');
  a.MatrSlice(Arr(0, 2), ArrGen(2, i -> 2 * i + 1)).Println(4);
  Writeln('Срез матрицы B');
  b.MatrSlice(0, 1, 3, 5).Println(4)
end.
```

```
   8 -39 -26  -1 -39
  -41   9  40  33 -13
 -12 -47 -10  33 -11
  50  24 -44 -28 -37
--------------------
  31  32 -17  -9  26 -30 -12
   7  25   6 -30 -31 -36 -40
---------------------------
Min(A) = -47
Среднее по B: -5.57142857142857
Сумма по колонкам в A: -3 -14 -14 38 -61
Сумма кубов модулей элементов последней строки A: 296613
Максимальный элемент в B: B[1,2] = 32
Выборка строк и столбцов матрицы A по индексам
   9  33
  24 -28
Срез матрицы B
  -9  26 -30
 -30 -31 -36
```

8.1.8 Перебор всех элементов матрицы (.foreach)

Процедура .foreach является расширением, позволяющим последовательно перебрать все элементы матрицы в порядке «по строкам». Имеются две разновидности, отличающиеся набором параметров лямбда-процедур:

- a.ForEach(T-> ()) применяет указанное лямбда-процедурой действие к каждому элементу матрицы a;
- a.ForEach((T,i,j)-> ()) применяет указанное лямбда-процедурой действие к каждому элементу матрицы a, предоставляя доступ к его индексам i и j.

Например, организовать вывод элементов массива в желаемом оформлении можно следующим образом.

```
begin
  var a := MatrRandom(3, 5, -9999, 9999);
  a.ForEach((v, i, j)->
    begin $'{v,10:# ##0}'.Print; if j = a.ColCount-1 then Println end)
end.
```

```
   322      4 799      9 543     -1 905     -3 759
 4 847      6 600     -9 545      8 367      8 560
-1 762      5 268     -3 794     -6 507     -4 447
```

8.1.9 Модификация строк и столбцов

Помимо выборки отдельных строк и столбцов, в PascalABC.NET включены расширения для их замены, обмена местами выбранной пары строк/столбцов и преобразования всех элементов матрицы.

- a.ConvertAll(T -> T1) – функция, возвращающая матрицу типа T1, в которой каждый элемент матрицы a типа T преобразован в соответствии с заданным лямбда-выражением;

- a.ConvertAll((T, i ,j) -> T1) – функция, возвращающая матрицу, в которой каждый элемент матрицы a преобразован в соответствии с заданным лямбда-выражением, в котором участвуют индексы строки и столбца;
- a.Fill((i, j) -> T) – процедура, заполняющая каждый элемент матрицы a в соответствии с заданным лямбда-выражением, в котором участвуют индексы строки и столбца;
- a.SetCol(k, v) – процедура, заменяющая в матрице a элементы столбца с индексом k элементами массива или последовательности v того же типа;
- a.SetRow(k, v) – процедура, заменяющая в матрице a элементы строки с индексом k элементами массива или последовательности v того же типа;
- a.SwapCols(k1, k2) – процедура, обменивающая местами колонки с индексами k1 и k2;
- a.SwapRows(k1, k2) – процедура, обменивающая местами строки с индексами k1 и k2;
- a.Transform(T -> T) – процедура, преобразующая каждый элемент матрицы a в соответствии с заданным лямбда-выражением;
- a. Transform((T, i ,j) -> T) – процедура, преобразующая каждый элемент матрицы a в соответствии с заданным лямбда-выражением, в котором участвуют индексы строки и столбца.

В качестве примера создадим целочисленную матрицу A размером 7×7 из случайных чисел на интервале [-50;50] и найдем среднее значение m среди всех ее элементов. Далее, заменим все элементы, отличающиеся по абсолютной величине от m больше чем на 20, суммой индекса строки и столбца такого элемента, взятой со случайным знаком. В полученной матрице заполним нулями все строки, в которых положительных элементов будет больше, чем прочих. Выведем исходную матрицу, найденное значение m, матрицу, полученную после выборочной замены элементов и результирующую матрицу.

```
begin // p08008
  var a := MatrRandom(7, 7, -50, 50);
  a.Println;
  Println('-' * 7 * 4);
  var m := a.ElementsByRow.Average;
  Println('m =', m);
  var rs: () -> integer := () -> Random(1, 2) = 1 ? -1 : 1;
  a.Transform((t, i, j) -> Abs(t - m) > 20 ? rs * (i + j) : t);
  a.Println;
  Println('-' * 7 * 4);
  var b := a.Rows.Select(row -> row.Where(t -> t > 0).Count).ToArray;
  var nCols := a.ColCount;
  var c := ArrFill(nCols, 0);
  for var i := 0 to a.RowCount - 1 do
    if 2 * b[i] > nCols then
      a.SetRow(i, c);
  a.Println
end.
```

```
 -45   5  47   0  10 -22  36
  -4  -2 -23 -36 -16   1  31
 -10 -21  49  20 -40  13  20
 -34 -20  50 -22  47 -45 -50
  -5  39  45  24   7   0  26
   5  -9  -9   6 -13  38 -19
 -24 -33 -50 -21 -16 -23  37
----------------------------
m = -1.14285714285714
   0   5   2   0  10  -5   6
  -4  -2   3   4 -16   1   7
 -10 -21  -4   5  -6  13   8
  -3 -20   5  -6   7  -8  -9
  -5   5  -6  -7   7   0  10
   5  -9  -9   6 -13  10 -19
   6  -7   8 -21 -16  11 -12
----------------------------
   0   0   0   0   0   0   0
   0   0   0   0   0   0   0
 -10 -21  -4   5  -6  13   8
  -3 -20   5  -6   7  -8  -9
  -5   5  -6  -7   7   0  10
   5  -9  -9   6 -13  10 -19
   6  -7   8 -21 -16  11 -12
```

Остановимся на некоторых операторах приведенной программы. Переменная rs – это имя лямбда-функции без параметров, возвращающей случайное значение типа integer, равное 1 или -1. Процедура Transform преобразует матрицу по указанному в условии задачи правилу: если для некоторого элемента t матрицы Abs(t - m) > 20, то значение элемента заменяется суммой его индексов, умноженной на значение функции rs. Значения элемента массива b[i] – это количество положительных элементов в строке и индексом i. Массив с содержит нули, его размер равен количеству столбцов в матрице. Строки матрицы последовательно просматриваются в цикле **for**, поскольку нам понадобятся индексы строк в случае замены. В теле этого цикла проверяется заданное условие и при его выполнении делается замена строки посредством SetRow. В самом деле, если b[i] – количество положительных элементов, а nCols – их общее количество, исходное условие выглядит как b[i] > nCols – b[i] или 2*b[i] > nCols.

8.1.10 Транспонирование матрицы

Транспонированием матрицы в математике называется замена ее строк столбцами. Исходная матрица размером m×n превращается в матрицу размером n×m.

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \end{pmatrix} \rightarrow \begin{pmatrix} a_{00} & a_{10} & a_{20} & a_{30} \\ a_{01} & a_{11} & a_{21} & a_{31} \\ a_{02} & a_{12} & a_{22} & a_{32} \end{pmatrix}$$

Операция транспонирования матрицы a может быть выполнена посредством функции Transpose(a), возвращающей транспонированную матрицу.

8.1.11 О матричных операциях

Как отмечалось ранее, матрицы в PascalABC.NET – это всего лишь название двухмерных динамических массивов. В математике под матрицами понимают совсем иные объекты, для которых определен набор соответствующих операций и операторов. Некоторая их часть реализована для класса Matrix в библиотеке численных методов NumLibABC (см. часть 11), входящей в состав стандартной поставки PascalABC.NET.

8.2 Статические двухмерные массивы

Как и в случае с одномерными массивами, статический двухмерный массив описывается с указанием границ индексов. Границы должны быть заданы и для строк, и для столбцов. Память под статический массив распределяется на этапе компиляции, выделяется при загрузке программы, и в дальнейшем не может быть перераспределена.

Статический двухмерный массив описывается в виде

```
var ИмяМассива: array[m1..n1, m2..n2] of Тип; или
var ИмяМассива: array[t1, t2] of Тип;
```

Конструкция вида m..n описывает минимальное и максимальное значение, которое может принимать индекс массива, причем допускаются и отрицательные значения. Эта конструкция задается константой порядкового типа. Количество элементов в массиве можно вычислить по формуле (n1-m1+1)×(n2-m2+1).

```
var a: array[0..12, 1,,4] of byte;
var b, c: array[-5..8, 0..6] of real;
```

Описание массива можно совместить с инициализацией его элементов. Как и для динамических массивов, поскольку тип массива задан, можно при инициализации массива вещественного типа указывать в списке значения целочисленных типов.

```
begin // p08009
  var a: array[1..3, 1..4] of integer := ((1, 2, 3, 4),
      (5, 6, 7, 8), (9, 10, 11, 12));
  Println(a);
  var b: array[0..2, 1..3] of real := ((1.2, 5, -3.05),
      (-4, -7, 1), (15, 7, 7));
  Println(b);
  var c: array [0..2, 1..2] of char := (('a', 'b'),
      ('c', 'd'), ('e', 'f'));
  Println(c)
end.
```

```
[[1,2,3,4],[5,6,7,8],[9,10,11,12]]
[[1.2,5,-3.05],[-4,-7,1],[15,7,7]]
[[a,b],[c,d],[e,f]]
```

Двухмерные статические массивы в PascalABC.NET оставлены в целях совместимости с базовым Паскалем.

8.3 Массивы размерности выше двух

Общепринятое название массивов, имеющих размерность выше двух, – многомерные массивы. PascalABC.NET не имеет специальных средств работы с многомерными массивами – реализованы лишь их описание, создание, инициализация и вывод. Также, многомерные массивы можно присваивать и передавать в качестве параметров.

При описании и инициализации многомерных массивов нужно задавать границы их индексов по каждому измерению и соответствующий им конструктор массива.

```
begin // p08010
  var a: array[1..2, 1..3, 1..2] of integer := (((1, 2), (3, 4), (5, 6)),
      ((7, 8), (9, 10), (11, 12)));
  Println(a);
  a[1, 2, 2] := 0;
  Println(a);
end.
```

```
[[[1,2],[3,4],[5,6]],[[7,8],[9,10],[11,12]]]
[[[1,2],[3,0],[5,6]],[[7,8],[9,10],[11,12]]]
```

8.4 Примеры решения задач с матрицами

Задача 1. Переставить строки матрицы C размера m×n: первую с последней, вторую с предпоследней и т.д. В полученной матрице умножить каждый элемент третьей сверху строки на сумму элементов предпоследнего столбца. Значения m и n ввести с клавиатуры, элементы матрицы получить по формуле $C_{ij} = \sin(2*i-j)$ и округлить до трех знаков после запятой.

Здесь единственный подвох – случай с нечетным количеством строк m. Нужно сделать m/2 перестановок строк. «Третья сверху» строка будет иметь индекс 2, а предпоследний столбец – индекс n-2.

```
begin // p08011
  var (m, n) := ReadInteger2;
  var c := MatrGen(m, n, (i, j)-> Round(Sin(2 * i - j), 3));
  c.Println(7, 3);
  Println('-' * 7 * n); // разделитель
  for var i := 0 to (m - 1) div 2 do
    c.SwapRows(i, m - i - 1);
  c.Println(7, 3);
  Println('-' * 7 * n);
  var s := c.Col(n - 2).Sum;
  c.SetRow(2, c.Row(2).Select(t -> t * s));
  c.Println(7, 3)
end.
```

```
5 8
  0.000 -0.841 -0.909 -0.141  0.757  0.959  0.279 -0.657
  0.909  0.841  0.000 -0.841 -0.909 -0.141  0.757  0.959
 -0.757  0.141  0.909  0.841  0.000 -0.841 -0.909 -0.141
 -0.279 -0.959 -0.757  0.141  0.909  0.841  0.000 -0.841
  0.989  0.657 -0.279 -0.959 -0.757  0.141  0.909  0.841
-------------------------------------------------------
  0.989  0.657 -0.279 -0.959 -0.757  0.141  0.909  0.841
 -0.279 -0.959 -0.757  0.141  0.909  0.841  0.000 -0.841
 -0.757  0.141  0.909  0.841  0.000 -0.841 -0.909 -0.141
  0.909  0.841  0.000 -0.841 -0.909 -0.141  0.757  0.959
  0.000 -0.841 -0.909 -0.141  0.757  0.959  0.279 -0.657
-------------------------------------------------------
  0.989  0.657 -0.279 -0.959 -0.757  0.141  0.909  0.841
 -0.279 -0.959 -0.757  0.141  0.909  0.841  0.000 -0.841
 -0.784  0.146  0.942  0.871  0.000 -0.871 -0.942 -0.146
  0.909  0.841  0.000 -0.841 -0.909 -0.141  0.757  0.959
  0.000 -0.841 -0.909 -0.141  0.757  0.959  0.279 -0.657
```

Задача 2. Упорядочить столбцы матрицы A размера m×n таким образом, чтобы элементы второй сверху строки образовали невозрастающую лексикографическую последовательность. Значения m и n ввести с клавиатуры, элементы матрицы заполнить случайными строчными буквами латинского алфавита.

В латинском алфавите 26 букв и в кодовой таблице строчные буквы следуют от «a» до «z» непосредственно друг за другом в алфавитном порядке. Поэтому достаточно сгенерировать случайное число r в диапазоне [0..25] и выбрать символ с кодом буквы «a», увеличенным на r. Далее следует принять решение, производить ли перестановку столбцов в существующей матрице или определить новую, а затем поместить в нее столбцы исходной матрицы в нужном порядке. В первом случае придется реализовать алгоритм обменной сортировки, но менять местами не элементы, а целиком столбцы. Во втором случае можно создать на базе элементов второй строки последовательность кортежей, в которые войдет значение элемента и его индекс в строке, отсортировать элементы этой последовательности по невозрастанию значений, а затем скопировать в новую матрицу колонки в порядке, указанном индексами из состава кортежей. Любое решение обладает как достоин-

ствами, так и недостатками. Здесь будут приведены оба варианта, а выбор останется читателю.

Вариант А. Сортировка матрицы на месте.

```
begin // p08012
  var (m, n) := ReadInteger2;
  var a := MatrGen(m, n, (i, j)-> Chr(Ord('a') + Random(0, 25)));
  a.Println(2);
  Println;
  for var i := n - 2 downto 0 do
    for var j := 0 to i do
      if a[1, j] < a[1, j + 1] then a.SwapCols(j, j + 1);
  a.Println(2)
end.
```

```
5 8
 e z z x k k m g
 r d t l e x b s
 d x q j e e q k
 o o q t l f l s
 m d m v q m k k

 k z g e x k z m
 x t s r l e d b
 e q k d j e x q
 f q s o t l o l
 m m k m v q d k
```

Вариант Б. Сортировка с использованием вектора индексов.

```
begin // p08013
  var (m, n) := ReadInteger2;
  var a := MatrGen(m, n, (i, j)-> Chr(Ord('a') + Random(0, 25)));
  a.Println(2);
  Println;
  var v := a.Row(1).Select((v, i)-> (v, i))
      .OrderByDescending(t -> t[0]).ToArray;
  var b := new char[m, n];
  for var i := 0 to n - 1 do
    b.SetCol(i, a.Col(v[i][1]));
  a := b;
  a.Println(2)
end.
```

Это вариант для тех, кто не помнит, как реализовать алгоритм обменной сортировки.

Задача 3. Найдена «на просторах Интернет», но не зря же говорят, что все пути ведут в Рим – это задача Matrix27 из задачника М.Э. Абрамяна. На //youtube.com выложен ролик, где в течение тринадцати (!) минут автор старательно пыталась изложить решение этой задачи в среде PascalABC.NET, но… средствами базового Паскаля. Буду честным: я добросовестно пытался заставить себя выслушать пред-

лагаемый ход решения задачи методом проб и ошибок, но это оказалось сверхзадачей. Замечу лишь, что итоговое решение содержало 30 строк кода.

Итак, несложное условие: Дана матрица размером M×N. Найти максимальный среди минимальных элементов ее строк.

Как в предложенном решении, матрицу будем заполнять целочисленными значениями на интервале [0; 20] с помощью датчика случайных чисел. Выведем исходную матрицу. Далее построим последовательность, содержащую минимальные элементы каждой строки и найдем значение максимального элемента этой последовательности.

```
begin // p08014
  var (m, n) := ReadInteger2('Введите M и N:');
  var a := MatrRandom(m, n, 0, 20);
  a.Println(3);
  Println(3 * n * '-'); // разделитель
  a.Rows.Select(row -> row.Min).Max.Println
end.
```

```
Введите M и N: 7 12
 20  6 10 14 15 18  8 10 20  5  1 11
  3  1  9  1 10 11  3 18  6  0 12  5
 18 16  5 18  8 15 20 17  9 18 19  4
  2 12 11 15  8 16  3  6 14  0  6  3
 14  4 11  1 20 15  2 20 15  4 17 19
 12 10  1  4  5  5  2  5  3  5 20 18
 20  8 10 10 17  5 11 17 10  2  4  3
------------------------------------
4
```

Семь строчек. А никак не тридцать. Из них одна – предпоследняя – решает поставленную задачу, а прочие выполняют вспомогательные функции. Программа пишется за пару минут и не требует запутанных объяснений. Это – PascalABC.NET.

8.5 Для самостоятельного решения

T8.1. Заполните квадратную матрицу размера n×n (n – нечетное число) в соответствии со следующей схемой:

```
* 1 1 1 1 1 *
4 * 1 1 1 * 2
4 4 * 1 * 2 2
4 4 4 * 2 2 2
4 4 * 3 * 2 2
4 * 3 3 3 * 2
* 3 3 3 3 3 *
```

T8.2. Напишите программу, выводящую календарь на указанный год и месяц. Используйте тип данных System.DateTime по примеру заданий T7.3 и T7.4. Возможно, понадобится дополнительная информация из Интернет.

Рекомендуемая форма вывода:

```
Пн      2   9  16  23  30
Вт      3  10  17  24
Ср      4  11  18  25
Чт      5  12  19  26
Пт      6  13  20  27
Сб      7  14  21  28
Вс  1   8  15  22  29
```

Т8.3. Задан массив размером m×n со случайными целочисленными элементами из диапазона [-99; 99]. В каждой строке заменить нулями элементы, отличающиеся от среднего по строке значения более чем на 75. После этого вычеркнуть из массива строки и столбцы, на пересечении которых находятся нули. Поиск нулевых элементов вести по строкам.

Часть 9

Файлы

Самое непостижимое в этом мире — то, что он постижим.

Альберт Эйнштейн

Все ранее рассмотренные массивы и множества обладали одним общим свойством: перед обработкой они полностью располагались в оперативной памяти. Отмечалось также, что последовательности sequence в памяти не хранятся, а загружаются по одному элементу в момент обращения к нему. Количество элементов в таких последовательностях было либо заранее известно, либо последовательность считалась бесконечной и обрабатывалась до выполнения некоторого условия.

В случае, когда программа должна принимать в обработку и/или выдавать большой объем данных, клавиатура, монитор и принтер не слишком хорошо подходят для ввода и вывода. И тогда используют ***файлы***.

С точки зрения программы файл можно рассматривать, как последовательность заранее неизвестного количества элементов данных, располагающихся за пределами оперативной памяти компьютера, например, на его жестком диске. Каждый элемент данных в файле называется ***записью*** (не путайте с типом данных «запись» (**record**) языка Паскаль). Поскольку программа может непосредственно оперировать только с объектами в оперативной памяти компьютера, для взаимодействия с файлами она вынуждена обращаться к операционной системе. Для связи файлов операционной системы с программой используется ***файловая переменная***. А поскольку есть переменная, ее необходимо описывать, для чего в язык введен тип, который так и называется – ***файловый***. Следует отметить, что в операционной системе понятие файла несколько отличается от понятия файла в языке Паскаль.

Традиционный процесс работы с файлами строится по следующей схеме:

- определяется переменная файлового типа;
- эта переменная связывается с реальным файлом;
- файл открывается с целью его чтения, либо записи или дозаписи в него;
- выполняются операции по обмену данными с файлом;
- файл закрывается; при этом операционная система фиксирует сделанные в нем изменения.

PascalABC.NET позволяет при желании отойти от этой традиции, объединяя несколько этапов работы с файлом. Как будет показано в дальнейшем, в определенных случаях имеется возможность и все пять этапов выполнить в единственном операторе.

Существует устойчивое мнение, что освоить полноценную работу с файлами в Паскале несложно. Отчасти оно верно. Для базового Паскаля. А вот для PascalABC.NET данное мнение выглядит очень спорным. Эта часть книги писалась, а затем многократно подвергалась существенной правке на протяжении почти полугода. Но вы можете чувствовать себя вполне комфортно, определив некоторое подмножество средств работы с файлами и не выходя за его пределы.

9.1 Файловый тип данных

В Паскале файлы могут быть ***текстовыми*** или ***двоичными*** (бинарными). По сравнению с базовым Паскалем, средства для работы с файлами в PascalABC.NET были существенно расширены и дополнены. Насколько это оказалось возможным, сохранена совместимость с базовым Паскалем, однако работа с ***бестиповыми файлами*** в PascalABC.NET существенно изменена.

В ***текстовом файле*** операционной системы Windows записи разделяются комбинацией символов с кодами #13#10, а в файлах системы Linux – только символом с кодом #10. Чтобы не привязываться к этому коду, удобно использовать константу NewLine. Данные в текстовом файле хранятся в символьном виде, поэтому такой файл может обрабатываться в текстовом редакторе. Программа рассматривает текстовый файл как последовательность физических записей (строк), в которой к записи с номером k можно обратиться, только просмотрев предыдущие k-1 записи, поэтому такой доступ к данным называется ***последовательным***. Обмен данными с текстовым файлом, как и в базовом Паскале, может производиться по одной записи, но также имеются средства, позволяющие в одном операторе обмениваться между содержимым файла и последовательностью или массивом данных.

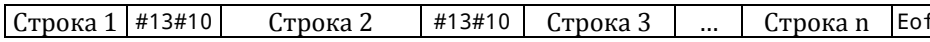


Рис. 9.1. Структура текстового файла Windows

На рисунке 9.1 показана структура текстового файла в операционной системе Windows. Eof – это условное обозначение признака конца файла, о достижении которого программе сообщает операционная система. Каждая строка может иметь собственную длину, в том числе, нулевую.

Пока вы работаете с текстовыми файлами на своем компьютере, проблем не возникает. При переносе файлов на другой компьютер необходимо учитывать кодировку символов и так называемые региональные стандарты. Даже на компьютере коллеги по работе или соседа в школьном компьютерном классе могут оказаться настройки, отличные от настроек вашего компьютера и всего лишь замена десятичной точки на запятую может привести к катастрофическому результату. Чего уж тут говорить о ситуации, когда на компьютерах имеются разные кодировки символов, установленные по умолчанию (Windows default)!

Двоичные файлы разделителей не содержат, поэтому обычно их считывают определенными порциями байт. Текстовый файл при необходимости также можно обрабатывать, как двоичный (например, читать в Windows файл, созданный в Linux). Интерпретация содержимого записи двоичного файла возлагается на программиста. В этом вся сила двоичных файлов (делаем, что пожелаем), но в этом же и их слабость (а вдруг мы ошиблись?). Двоичные файлы в PascalABC.NET могут быть ***типизированными*** или ***бестиповыми***. Вы уже знакомились с этими понятиями в части 7, когда рассматривали указатели.

Записи в типизированном файле представляют собой данные некоторого типа, обязательно известного в обрабатывающей программе. Эти записи всегда имеют **фиксированную длину**, что позволяет организовать ***произвольный доступ*** по номеру записи. Суть такого доступа заключается в том, что если каждая запись в файле имеет длину L, то запись с порядковым номером k смещена относительно начала файла на L×(k-1) байт и можно не выполнять последовательное чтение k-1 предшествующей записи. Произвольный доступ к записи внешне похож на доступ к элементу массива по его индексу. Он позволяет мгновенно найти местоположение нужной записи. Обмен данными с типизированным файлом осуществляется через переменную, которая имеет такой же тип, как и данные в файле.

Требование фиксированной длины записи налагает ограничение на тип данных, который может размещаться в типизированном файле: он может быть только размерным и/или базироваться на статических массивах. Понять это несложно: тип данных должен позволить вычислить длину записи на стадии компиляции.

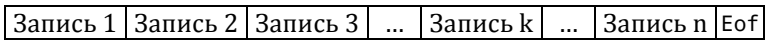


Рис. 9.2. Структура типизированного файла

В бестиповом файле длина физической записи не фиксирована. Обмен данными с такими файлами производится порциями байт требуемой длины. В PascalANC.NET бестиповый файл позволяет работать со строками любого типа, но зато не предоставляет возможности осуществлять произвольный доступ по номеру записи, поскольку ищется позиция в файле, отстоящая от его начала на указанное число байт.

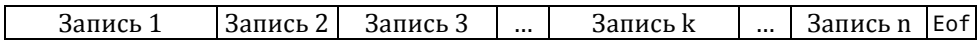


Рис. 9.3. Структура бестипового файла

Еще раз отметьте, что понятия типизированного и бестипового бинарного файла существуют лишь в пределах PascalABC.NET. Операционная система эти понятия не различает. Не различите их и вы, если попытаетесь рассмотреть содержимое ранее неизвестного бинарного файла. При обработке бинарного файла вы сами решаете считать его типизированным или бестиповым. Именно таким он и будет выступать в вашей программе.

9.1.1 Описание данных файлового типа

Описание переменной файлового типа определяет все особенности дальнейшей работы с файлом, в дальнейшем связываемым с этой переменной.

```
var
  f1 : Text; // переменная для связи с файлом текстового типа
  f2 : file of Тип; // переменная для связи с типизированным файлом
  f3 : file; // переменная для связи с бестиповым файлом
```

Для текстовых файлов зарезервирован тип Text, типизированный файл описывается с указанием любого известного в точке описания типа, а у бестипового файла тип данных не указывается вообще. Переменную, имеющую файловый тип, часто также называют ***файловой переменной***.

Рассмотрим пример описания еще одного типизированного файла:

```
type
  Точка = record
    x, y: real
  end;
  Многоугольник = array[1..24] of Точка;

begin
  var Многоугольники: file of Многоугольник;
end.
```

Тип Точка описывает точку на плоскости с вещественными координатами (x, y). Тип Многоугольник предназначен для описания координат вершин многоугольников числом до 24. В программе описана переменная Многоугольники, которая может быть связана с типизированным файлом. Переменная типа **real** занимает в памяти восемь байт, поэтому под данные типа Точка будет отведено 2×8 = 16 байт, а данные типа Многоугольник займут в памяти 24×16 = 384 байта. Соответствующий типизированный файл будет состоять из записей длиной 384 байта, ничем друг от друга не отделенных.

Рассмотрим пример начала такого файла в шестнадцатиричном виде.

```
0000000000: 38 13 1D 16 9C 89 4C C0 | D4 B6 9B DA 69 DB 50 C0
0000000010: 00 C4 62 88 FB 61 01 40 | 54 03 E5 F6 A9 81 51 C0
0000000020: FB 53 D2 89 FD 29 50 C0 | FC 54 41 9E 7D AA 46 40
0000000030: E8 57 B2 BD F2 2B 4B 40 | EC 36 DE 74 75 1B 3B C0
0000000040: C0 F9 E4 41 DD 7C 02 C0 | A4 C1 AF 2D D3 E0 4D 40
```

Даже заметив, что каждый восьмой байт содержит код C0 или 40 и предположив, что файл содержит данные типа **real**, мы ничего больше установить не можем, если заранее не знаем структуру записи. Это является одним из недостатков использования типизированных файлов: к ним нужно прилагать описание структуры записи. Второй недостаток мы ощутим, если попробуем записать в файл информацию о многоугольнике, у которого не 24 вершины. К сожалению, массив тут может быть только статическим, поскольку длина записи фиксирована. По этой же причине и строки в типизированных файлах допустимы только короткие, с фиксированной

длиной. Теперь вы можете догадаться, что архаичные статические массивы и короткие строки уйдут из PascalABC.NET лишь «под ручку» с типизированными файлами.

Современное решение, позволяющее эффективно обрабатывать данные с динамической структурой, состоит в использовании ***сериализации***. Структура данных трансформируется в последовательность бит, которые упаковываются в байты, а затем выгружаются в файл. После чтения файла структура данных восстанавливается при помощи обратного процесса – ***десериализации***. В библиотеке Microsoft .NET Framework имеются классы, позволяющие выполнять сериализацию и десериализацию, но объяснение принципов работы с ними выходит за рамки данной книги.

И последнее. Нельзя описать «файл файлов», т.е. в описании не может появиться конструкция по типу **file of file**.

9.1.2 Некоторые термины

Непонятно, чем руководствовался Н. Вирт, создавая язык Паскаль, но для объявления типов данных сложной структуры он использовал ключевое слово **record** (запись), создав тем самым коллизию с терминологией файловых систем, в которых файл рассматривается как совокупность записей – минимальных порций данных. Русский язык добавляет проблем, поскольку слово «запись» в нем может быть как существительным, так и глаголом. И получаются фразы о том, что в процессе *записи* в типизированный файл каждая операция *записи* помещает в него в качестве очередной *записи* текущее значение переменной типа *запись*. С целью хотя бы частичного устранения коллизий в этой части книги по отношению к структуре файла наравне с термином «запись файла» будет употребляться термин «блок данных». Аналогично, в отношении типа данных может быть использован термин «**record**». Тогда приведенная выше фраза приобретет более пристойный вид: в процессе *записи* в типизированный файл каждая операция *записи* помещает в него в качестве очередного блока данных текущее значение переменной типа **record**.

С понятием файла тесно связаны еще два понятия: ***файловый указатель*** и ***буфер***. Файловый указатель связывается с позицией относительно начала файла. Именно с этой позиции будет происходить чтение/запись очередного блока данных. Обмен данными происходит через буфер – некоторую область памяти, достаточную для их размещения. При открытии файла для чтения, записи или перезаписи файловый указатель устанавливается в начало файла. Определена также операция дозаписи в файл, позволяющая добавлять в файл данные, размещая их после уже имеющихся. При открытии файла для дозаписи файловый указатель устанавливается в конец файла. После совершения очередной операции обмена данными с файлом, файловый указатель перемещается в позицию, следующую непосредственно за последним прочитанным байтом. После считывания всего файла, файловый указатель устанавливается в конец файла, и наступает ситуация, называемая «конец файла»

(EOF – end of file). Достижения конца файла можно отследить программным путем, что позволяет читать файл в цикле **while** или **repeat**.

В типизированных файлах имеется возможность по желанию переустанавливать файловый указатель в начало любого существующего блока данных, а также в конец файла. Это позволяет организовать прямой (произвольный) доступ к записям файла.

У бестиповых файлов понятие блока данных отсутствует, но можно установить файловый указатель на любой байт в файле.

9.2 Связь программы с файлом

Любой файл, с которым работает программа:
- имеет имя в операционной системе, посредством которого можно обратиться к файлу;
- имеет неизвестную заранее длину.

Если файл текстовый или типизированный, он содержит записи одного типа. Для бестипового файла это не всегда так.

В Паскале переменную файлового типа связывают с реальным файлом при помощи процедуры Assign:

```
Assign(f, name);
```

где f – переменная файлового типа, а name – выражение символьного типа, значением которого является имя реального файла (возможно, включающее полный путь к файлу).

После вызова процедуры Assign имя файла больше нигде не используется: работа с файлом ведется через переменную файлового типа. Тип файла при этом определяется типом связанной с ним файловой переменной. Установленную связь можно разорвать до завершения работы программы, связав файловую переменную с каким-то другим файлом. В определенных случаях, рассмотренных ниже, PascalABC.NET позволяет обойтись без процедуры Assign.

Ниже даны несколько примеров использования процедуры Assign.

```
type
  tt = array[1..5] of char;

begin
  var f1: Text;
  var f2: file of tt;
  var f3: file;
  Assign(f1, 'myFile.txt');
  Assign(f2, 'C:\DataBuf.bin'); // включает путь к файлу
  Assign(f3, 'pic325.bmp');
end.
```

При установлении связи наличие файла в операционной системе не проверяется. И это логично, поскольку файл может создаваться. Но если в имени файла присутствует путь, этот путь должен существовать на стадии компиляции.

9.3 Открытие и закрытие файлов

Для того, чтобы начать читать книгу, нужно ее открыть. Для того, чтобы что-то записать в тетрадь, эту тетрадь тоже придется открыть. С файлами все происходит аналогично, но в отличие от книги или тетради, которые можно открыть на любом месте, при открытии файла подводится его начало (файловый указатель устанавливается в начало файла для создания, чтения или записи) или конец (файловый указатель устанавливается после последней записи в файле для дозаписи). Операция открытия файла позволяет программе удостовериться в его наличии, а операционной системе – приготовиться к совершению операций чтения/записи.

При осуществлении записи в файл различают операции собственно записи, перезаписи и дозаписи. Запись производится, начиная с предварительно определенной позиции внутри файла. При последовательном доступе запись можно производить с начала файла (перезапись с удалением прежнего содержимого), либо с текущей позиции файлового указателя (запись с заменой оставшейся части файла). Если файловый указатель установлен в конец файла, то операция записи добавит в файл новое содержимое; в таком случае говорят о дозаписи. При произвольном доступе запись можно производить, начиная с любого места внутри файла, в том числе, с конца файла, осуществляя дозапись. В типизированном файле перезаписывается порция (блок) данных с конкретным номером. В бестиповом файле перезаписывается некоторое количество байт, что при некорректном указании их количества может сделать логически недоступной оставшуюся часть файла.

После окончания работы с файлом, его необходимо закрыть. Закрытие можно производить явно, но можно этого и не делать, поскольку при завершении программы PascalABC.NET все файлы в любом случае окажутся закрыты. Закрытие файла – важная операция с точки зрения файловой системы: именно оно позволяет зафиксировать правильные параметры файла, такие как его длина, физическое местоположение на внешнем носителе, количество блоков (если оно поддерживается) и т.д. Даже если программа завершится аварийно, файлы все равно будут закрыты операционной системой, однако при этом не гарантируется, что их содержимое будет полностью корректным. И это веская причина, чтобы взять за правило явно закрывать файлы в программе. Один и тот же файл в программе можно закрывать и открывать произвольное количество раз, но надо понимать, что эта операция выполняется путем обращения к операционной системе и с точки зрения процессора является крайне медленной.

9.3.1 Открытие текстового файла

Текстовый файл может быть открыт для чтения, создания/записи и дозаписи.

В базовом Паскале открытие текстового файла осуществляется в следующей последовательности:
- описывается некоторая файловая переменная (назовем ее f) типа Text;
- вызывается процедура Assign, связывающая переменную f с файлом по имени name;
- производится собственно открытие файла посредством процедуры Reset (для чтения существующего файла с его начала), Rewrite (для создания нового файла или перезаписи существующего файла), либо Append (для дозаписи в существующий файл).

PascalABC.NET имеет дополнительные средства, позволяющие в той или иной степени объединить перечисленные выше действия.

9.3.1.1 Кодировка текстового файла

Когда Н.Вирт создавал язык Паскаль, у него не было необходимости задумываться о кодировках: операционная система имела единственный способ представления символов. Microsoft .NET позволяет использовать различные кодировки, и с одной стороны это благо, а с другой – головная боль для неискушенных пользователей.

Прежде всего, имеется кодировка Encoding.Default, установленная операционной системой Windows по умолчанию. Так, для «русской Windows» используется кодовая страница 1251, где каждый символ кодируется одним байтом. Но Microsoft .NET работает с таблицами Unicode, где каждый символ может кодироваться одним, двумя или даже четырьмя байтами. Кириллицу гарантированно поддерживают кодовые страницы 1200 (UTF-16) и 65001 (UTF-8) с двухбайтной кодировкой, а также кодовая страница 12000 с четырехбайтной кодировкой (UTF-32). Также, для «русской Windows», кириллица поддерживается, в частности, кодовыми страницами 866 (однобайтная кодировка «MS DOS») и 1251 (однобайтная кодировка Windows).

Если не указывать кодировку, то «русская Windows» будет считать, что текстовые файлы имеют кодировку 1251 (Encoding.Default). Для того, чтобы использовать другую кодировку, следует указать ее явно. Например, инициализировать в программе некоторую переменную типа Encoding, а затем использовать ее при открытии файла.

```
var en1:=Encoding.GetEncoding(866); // GetEncoding('cp866') - регистр важен!
var en2:=Encoding.GetEncoding(1200); // GetEncoding('UTF-16')
var en3:=Encoding.GetEncoding(65001); // GetEncoding('UTF-8')
var en4:=Encoding.GetEncoding(1251); // символического имени нет
var en5:=Encoding.ASCII; // кириллица не кодируется
var en6:=Encoding.Unicode; // то же, что UTF-16
var en7:=Encoding.UTF8; // то же, что UTF-8
```

9.3.1.2 Открытие текстового файла для чтения

Когда текстовый файл открывается для чтения, выполняется проверка существования файла. При отсутствии файла генерируется исключение вида «Ошибка времени выполнения: Файл … не найден.». Если в PascalABC.NET файл был записан в Unicode, то сведения о кодировке берутся из файла, в противном случае при отсутствии указания кодировки будет использована Encoding.Default.

> Если при открытии файла, записанного PascalABC.NET в Unicode, указывается кодировка, отличная от использованной при записи, это указание игнорируется.

- `Reset(файловая переменная)` – процедура подготавливает файл к чтению;
- `.Reset` – расширение для объектов файлового типа Text. Подготавливает файл к чтению;
- `Reset(файловая переменная, name)` – процедура связывает файловую переменную типа Text с файлом по имени name, затем подготавливает файл к чтению. Не требует предварительного вызова Assign;
- `Reset(файловая переменная, name, en)` – процедура связывает файловую переменную типа Text с файлом по имени name, затем подготавливает файл к чтению в кодировке, заданной параметром en. Не требует предварительного вызова Assign;
- `OpenRead(name)` – функция возвращает переменную типа Text, связывая ее с файлом по имени name. Подготавливает файл к чтению. Не требует предварительного описания файловой переменной и обращения к процедуре Assign;
- `OpenRead(name, en)` – функция возвращает переменную типа Text, связывая ее с файлом по имени name. Подготавливает файл к чтению в кодировке, заданной параметром en. Не требует предварительного описания файловой переменной и обращения к процедуре Assign.

Переменная, обозначенная en, предварительно определяется в программе (см. 9.3.1.1). По формату здесь указывается имя переменной, но никак не выражение.

Использовать Reset или OpenRead – решает программист. OpenRead позволяет вместо трех строк кода обходиться одной и обладает лучшей запоминаемостью. Сравните:

```
var f1: text;
Assign(f1, 'test1.txt');
Reset(f1);
var f2 := OpenRead('test2.txt'); // вместо трех строк – одна
```

Бывает, что файл читают непосредственно после завершения записи в него. В таких случаях может оказаться удобным закрыть файл и открыть его повторно посредством Reset.

9.3.1.3 Открытие текстового файла для записи

Вначале проверяется существование файла. Если файл существует, он очищается (фактически, операционная система устанавливает нулевую длину файла), в противном случае создается новый пустой файл. При попытке повторно открыть уже открытый файл, возникает исключение с сообщением «Ошибка времени выполнения: Процесс не может получить доступ к файлу "...", так как этот файл используется другим процессом.». Кодировкой по умолчанию является Windiws.Default.

- `Rewrite(файловая переменная)` – процедура подготавливает файл к записи в кодировке по умолчанию;
- `Rewrite(файловая переменная, name)` – процедура связывает файловую переменную типа Text с файлом по имени name, затем подготавливает файл к записи в кодировке по умолчанию. Не требует предварительного вызова Assign;
- `Rewrite(файловая переменная, name, en)` – процедура связывает файловую переменную типа Text с файлом по имени name, затем подготавливает файл к записи в кодировке, заданной параметром en. Не требует предварительного вызова Assign;
- `OpenWrite(name)` – функция возвращает переменную типа Text, связывая ее с файлом по имени name. Подготавливает файл к записи в кодировке по умолчанию. Не требует предварительного описания файловой переменной и обращения к процедуре Assign;
- `OpenWrite(name, en)` – функция возвращает переменную типа Text, связывая ее с файлом по имени name. Подготавливает файл к записи в кодировке, заданной параметром en. Не требует предварительного описания файловой переменной и обращения к процедуре Assign.

Переменная, обозначенная en, предварительно определяется в программе (см. 9.3.1.1). По формату здесь указывается имя переменной, но никак не выражение.

9.3.1.4 Открытие текстового файла для дозаписи

Когда текстовый файл открывается для дозаписи, выполняется проверка существования файла. При отсутствии файла генерируется исключение вида «Ошибка времени выполнения: Файл ... не найден.». Попытка открыть для дозаписи ранее открытый файл, как и в случае открытия файла на запись, считается ошибочной ситуацией. Дозапись в файл происходит после существующих записей. Если в PascalABC.NET файл был записан в Unicode, то сведения о кодировке берутся из файла, в противном случае при отсутствии указания кодировки будет использована Encoding.Default.

> Если при открытии файла, записанного PascalABC.NET, указывается кодировка, отличная от ранее использованной при записи, дозапись будет производиться в указанной кодировке. Отличный способ создать себе проблему при чтении.

- `Append(файловая переменная)` – процедура подготавливает файл к дозаписи;
- `Append(файловая переменная, name)` – процедура связывает файловую переменную типа Text с файлом по имени name, затем подготавливает файл к дозаписи. Не требует предварительного вызова Assign;
- `Append(файловая переменная, name, en)` – процедура связывает файловую переменную типа Text с файлом по имени name, затем подготавливает файл к дозаписи в кодировке, заданной параметром en. Не требует предварительного вызова Assign;
- `OpenAppend(name)` – функция возвращает переменную типа Text, связав ее с файлом по имени name. Подготавливает файл к дозаписи. Не требует предварительного описания файловой переменной и обращения к процедуре Assign;
- `OpenAppend(name, en)` – функция возвращает переменную типа Text, связав ее с файлом по имени name. Подготавливает файл к дозаписи в кодировке, заданной параметром en. Не требует предварительного описания файловой переменной и обращения к процедуре Assign.

Переменная, обозначенная en, предварительно определяется в программе (см. 9.3.1.1). По формату здесь указывается имя переменной, но никак не выражение.

9.3.2 Открытие типизированных файлов

Как отмечалось выше, типизированные файлы содержат записи фиксированной длины. Типизированные файлы могут открываться для чтения/записи или создания/перезаписи. В последнем случае, если файл существует, он очищается, иначе – создается. Файл, открытый для создания/перезаписи, считается также открытым для чтения/записи. Даже если файл создавался, как бестиповый, но имеет записи фиксированной длины, вы можете открыть его, как типизированный. Символьные данные по умолчанию записываются в файл в однобайтной кодировке Encoding.Default. В частности, в «русской Windows» будет использована кодировка 1251. Можно указать и другую кодировку, но она должна быть только однобайтной, иначе при выполнении программы возникнет исключение.

В базовом Паскале открытие типизированного файла типа T осуществляется следующим образом:
- При необходимости тип T описывается в секции описания типов **type**;
- описывается файловая переменная (назовем ее f) типа T;
- вызывается процедура Assign, связывающая переменную f с файлом по имени name;
- производится собственно открытие файла посредством процедуры Reset (для чтения/записи существующего файла) или Rewrite (для создания нового файла или перезаписи существующего).

В PascalABC.NET имеются дополнительные средства, позволяющие объединять перечисленные выше действия.

9.3.2.1 Открытие типизированного файла для чтения/записи

Работа с файлом, содержимое которого можно выборочно корректировать – еще одно достоинство типизированных файлов. Файл, открытый на чтение, одновременно открыт и на запись. Поскольку каждая запись имеет фиксированный размер, она может быть обновлена независимо от остальных записей.

Когда файл открывается для чтения/записи, выполняется проверка существования файла. При отсутствии файла генерируется исключение вида «Ошибка времени выполнения: Файл … не найден.». Если файл существует, подводится его начало.

- `Reset(файловая переменная)` – процедура подготавливает файл к чтению/записи;
- `Файловая_переменная.Reset()` – расширение для объектов типа **file of** T. Подготавливает файл к чтению/записи;
- `Reset(файловая переменная, name)` – процедура связывает файловую переменную типа **file of** T с файлом по имени name, затем подготавливает файл к чтению/записи. Не требует предварительного вызова Assign;
- `Reset(файловая переменная, name, en)` – процедура связывает файловую переменную типа **file of** T с файлом по имени name, затем подготавливает файл к чтению/записи в кодировке, заданной параметром en (см. 9.3.1.2). Не требует предварительного вызова Assign;
- `OpenFile&<T>(name)` – функция возвращает переменную типа **file of** T, связав ее с файлом по имени name. Подготавливает файл к чтению/записи. Не требует предварительного описания файловой переменной и обращения к процедуре Assign. ***Внимание!*** *Тип T не должен содержать в описании данных ссылочных типов; в таких случаях следует использовать процедуру Reset*;
- `OpenFileInteger(name)` – функция, возвращающая переменную типа **file of integer**, связав ее с файлом по имени name. Подготавливает файл к чтению/записи данных типа **integer** с длиной записи 4 байта. Не требует предварительного описания файловой переменной и обращения к процедуре Assign;
- `OpenFileReal(name)` – функция, возвращающая переменную типа **file of real**, связав ее с файлом по имени name. Подготавливает файл к чтению/записи данных типа **real** с длиной записи 8 байт. Не требует предварительного описания файловой переменной и обращения к процедуре Assign.

Ниже дан пример открытия типизированных файлов для чтения/записи.

```
type
  Person = record
    Имя, Фамилия: string[20];
    Возраст: integer
  end;
```

```
begin
  var f1: file of Person;
  Assign(f1, 'persons.bin');
  Reset(f1);
  var f2 := OpenFileReal('MyReals.dat');
  var f3 := OpenFile&<integer>('ФайлДанных1.data');
  var f4 := OpenFileInteger('ФайлДанных2.data');
  // а вот так файл со строками не открывайте:
  var f5 := OpenFile&<Person>('Persons.fil')
end.
```

9.3.2.2 Открытие типизированного файла для создания/перезаписи

Типизированный файл может быть открыт для создания или перезаписи. При попытке повторно открыть ранее открытый файл, возникает исключение с сообщением «Ошибка времени выполнения: Процесс не может получить доступ к файлу "...", так как этот файл используется другим процессом.».

- `Rewrite(файловая переменная)` – процедура подготавливает файл к созданию/перезаписи;
- `Файловая_переменная.Rewrite()` – расширение для объектов типа **file of** T. Подготавливает файл к созданию/перезаписи;
- `Rewrite(файловая переменная, name)` – процедура связывает файловую переменную типа **file of** T с файлом по имени name, затем подготавливает файл к созданию/перезаписи. Не требует предварительного вызова Assign;
- `Rewrite(файловая переменная, name, en)` – процедура связывает файловую переменную типа **file of** T с файлом по имени name, затем подготавливает файл к созданию/перезаписи в кодировке, заданной параметром en (см. 9.3.1.1). Не требует предварительного вызова Assign;
- `CreateFile&<T>(name)` – функция возвращает переменную типа **file of** T, связав ее с файлом по имени name. Подготавливает файл к созданию/перезаписи. Не требует предварительного описания файловой переменной и обращения к процедуре Assign. ***Внимание!*** *Тип T не должен содержать в описании данных ссылочных типов; в таких случаях следует использовать процедуру Reset*;
- `CreateFileInteger(name)` – функция, возвращающая переменную типа **file of integer**, связав ее с файлом по имени name. Подготавливает файл к созданию/перезаписи данных типа **integer** с длиной записи 4 байта. Не требует предварительного описания файловой переменной и обращения к процедуре Assign;
- `CreateFileReal(name)` – функция, возвращающая переменную типа **file of real**, связав ее с файлом по имени name. Подготавливает файл к созданию/перезаписи данных типа **real** с длиной записи 8 байт. Не требует предварительного описания файловой переменной и обращения к процедуре Assign.

Ниже дан пример открытия типизированных файлов для создания/перезаписи.

```
type
  Person = record
    Имя, Фамилия: string[20];
    Возраст: integer
  end;

begin
  var f1: file of Person;
  Rewrite(f1, 'persons.bin');
  var f2 := CreateFileReal('MyReals.dat');
  var f3 := CreateFile&<integer>('ФайлДанных1.data');
  var f4 := CreateFileInteger('ФайлДанных2.data');
  // а вот так файл со строками не открывайте:
  var f5 := CreateFile&<Person>('Persons.fil')
end.
```

9.3.3 Открытие бестиповых файлов

Бестиповые файлы, как и типизированные, могут открываться для чтения/записи или создания/перезаписи. В последнем случае, если файл существует, он очищается, иначе – создается. Файл, открытый для создания/перезаписи, считается также открытым для чтения/записи. Даже если файл создавался как типизированный, вы можете его открыть, как бестиповый. Символьные данные по умолчанию записываются в файл в однобайтной кодировке Encoding.Default. В частности, в «русской Windows» будет использована кодировка 1251. Можно указать и другую кодировку, но она должна быть только однобайтной, если этот файл в дальнейшем планируется открывать, как типизированный. Полезно также иметь в виду, что при использовании таблиц Unicode в начале записи дополнительно размещается сигнатура конкретной кодировки.

9.3.3.1 Открытие бестипового файла на чтение/запись

Когда файл открывается для чтения/записи, выполняется проверка существования файла. При отсутствии файла генерируется исключение вида «Ошибка времени выполнения: Файл … не найден.».

- `Reset(файловая переменная)` – процедура проверяет наличие файла и подводит его начало, приготавливая файл к чтению/записи. Используется, если предварительно вызывалась процедура Assign;
- `Файловая_переменная.Reset()` – расширение для объектов файлового типа. Проверяет наличие файла и подводит его начало, приготавливая к чтению/записи. Используется, если предварительно вызывалась процедура Assign;
- `Reset(файловая переменная, name)` – процедура связывает ранее описанную файловую переменную с файлом по имени name. Проверяет наличие файла и подводит его начало, приготавливая к чтению/записи. Не требует предварительного обращения к процедуре Assign;
- `Reset(файловая переменная, name, en)` – процедура связывает ранее описанную файловую переменную с файлом по имени name. Проверяет наличие файла и

подводит его начало, приготавливая к чтению/записи в кодировке, заданной параметром en (см. 9.3.1.1). Не требует предварительного обращения к процедуре Assign;

- `OpenBinary(name)` – функция, возвращающая переменную, связанную с файлом по имени name. Проверяет наличие файла и подводит его начало, приготавливая к чтению/записи. Не требует предварительного обращения к процедуре Assign и предварительного описания переменной файлового типа.

9.3.3.2 Открытие бестипового файла на создание/перезапись

Бестиповый файл может быть открыт для создания или перезаписи. Вначале проверяется его существование. Если файл существует, он очищается, в противном случае создается новый пустой файл. При попытке повторно открыть ранее открытый файл, возникает исключение с сообщением «Ошибка времени выполнения: Процесс не может получить доступ к файлу "...", так как этот файл используется другим процессом.».

- `Rewrite(файловая переменная)` – процедура подготавливает бестиповый файл к созданию или перезаписи. Используется, если предварительно вызывалась процедура Assign;
- `Файловая_переменная.Rewrite()` – расширение для объектов файлового типа. Подготавливает бестиповый файл к созданию или перезаписи. Используется, если предварительно вызывалась процедура Assign;
- `Rewrite(файловая переменная, name)` – процедура связывает ранее описанную файловую переменную соответствующего типа с файлом по имени name. Подготавливает бестиповый файл к созданию или перезаписи. Не требует предварительного обращения к процедуре Assign;
- `Rewrite(файловая переменная, name, en)` – процедура связывает ранее описанную файловую переменную с файлом по имени name. Подготавливает бестиповый файл к созданию или перезаписи в кодировке, заданной параметром en (см. 9.3.1.1). Не требует предварительного обращения к процедуре Assign;
- `CreateBinary(name)` – функция, возвращающая переменную, связанную с файлом по имени name. Подготавливает бестиповый файл к созданию или перезаписи. Не требует предварительного обращения к процедуре Assign и предварительного описания переменной файлового типа.

9.3.4 Закрытие файлов

Закрытие файла является необходимой операцией, фиксирующей сделанные в файле изменения. Оно осуществляется одинаково для файлов любого типа. Обычно закрытие файлов выполняется в программе явно. Все открытые файлы, не закрытые явно, будут автоматически закрыты при завершении выполнения программы, но это дурной стиль в программировании. Закрытие файла не разрывает связи файла с файловой переменной, поэтому закрытый файл можно при необходимости повторно открыть, не устанавливая такой связи еще раз.

- `Close(файловая переменная)` – процедура закрывающая файл, связанный с указанной файловой переменной;
- `CloseFile(файловая переменная)` – синоним процедуры Close;
- `Файловая_переменная.Close` – метод, закрывающий файл, связанный с указанной файловой переменной.

9.4 Запись данных в файл

Операция записи в файл заключается в помещении в него порции (***блока***) некоторых данных в целях их хранения и последующего извлечения. Выполняется после открытия файла в режиме, допускающем запись/перезапись.

9.4.1 Запись данных в текстовый файл

Записи в текстовом файле отделяются друг от друга символами-разделителями (см. 9.1). За размещение символа-разделителя в файле отвечает программист. Создать разделитель можно, например, используя при выводе константу NewLine, или вызов процедуры Writeln/Println, завершающей вывод сменой строки. Это позволяет одной операции записи сформировать в файле сразу несколько блоков данных. Если созданный текстовый файл в дальнейшем предполагается читать другой программой, следует обеспечить такую возможность. В противном случае, при считывании могут возникать сложности с распознаванием данных.
Рассмотрим простой пример программы, пока что не имеющий отношения к записи в файл.

```
begin
  Writeln(Arr(1,5,9,3,2,8));
  Writeln(SeqRandom(10));
  Writeln(['a'..'h']);
  Writeln('2+3=',2+3)
end.
```

Программа отправит на вывод четыре строки следующего вида:

```
[1,5,9,3,2,8]
[13,89,4,80,10,88,47,40,55,43]
{e,d,c,b,a,h,g,f}
2+3=5
```

Если представить себе файл, содержащий данные в подобном виде, то как считывать их в другой программе? Паскаль ведь принимает значения, разделенные пробелами или символами смены строки. Что делать со скобками? Конечно, можно читать файл построчно, убирать ненужные символы и заменять запятые пробелами, но хорошо ли это – своей неумелостью или безалаберностью создавать другим программистам проблемы?

Перед тем как выводить данные в текстовый файл, новичкам полезно хотя бы часть их отобразить на мониторе с тем, чтобы представить себе структуру получаемого файла.

Вот еще один пример создания будущих проблем.

```
// p09001
type
  t1 = record
    fld1: array[1..5] of byte;
    fld2: integer;
    fld3: real;
    fld4: array[0..1] of boolean;
    fld5: string
  end;


begin
  var v: t1;
  for var i := 1 to 5 do
    v.fld1[i] := 16 * i - 15;
  v.fld2 := 10;
  v.fld3 := -25.34;
  v.fld4[0] := False;
  v.fld4[1] := True;
  v.fld5 := '01234АВСDЕАБВГД';
  Write(v);
  Writeln(255);
  Writeln('Работа программы завершена');
end.
```

На выводе получаем две строки:

```
([1,17,33,49,65],10,-25.34,[False,True],01234АВСDЕАБВГД)255
Работа программы завершена
```

Содержимое записи v, имеющей тип t1, выведено в круглых скобках. В квадратных скобках выведены значения элементов массивов fld1 и fld4. Все выводимые элементы записи v, а также элементы массивов, перечислены через запятую, а длина строки усечена по фактическому количеству символов в ней. Значение константы 255 без каких-либо разделителей выведено после вывода записи v, а затем процедура Writeln произвела смену строки. На новой строке размещен текст «Работа программы завершена».

Если теперь мы изменим программу с тем, чтобы направить вывод в файл, содержимое этого файла будет идентично тому, что отображалось на мониторе.

Направить вывод данных в текстовый файл очень просто. Достаточно открыть файл в режиме записи или дозаписи и при обращении к процедурам Write/Writeln (Print/Println) в качестве первого элемента списка вывода указывать имя файловой переменной, связанной с открытым файлом.

Просмотрите, как мало изменилась предыдущая программа.

```
// p09002
type
  t1 = record
    fld1: array[1..5] of byte;
    fld2: integer;
    fld3: real;
    fld4: array[0..1] of boolean;
    fld5: string
  end;


begin
  var f := OpenWrite('Test.txt'); // открываем текстовый файл
  var v: t1;
  for var i := 1 to 5 do
    v.fld1[i] := 16 * i - 15;
  v.fld2 := 10;
  v.fld3 := -25.34;
  v.fld4[0] := False;
  v.fld4[1] := True;
  v.fld5 := '01234АВСDЕАБВГД';
  Write(f, v); // первой в списке идет файловая переменная
  Writeln(f, 255);
  Writeln(f, 'Работа программы завершена');
  f.Close // закрываем файл
end.
```

Добавились две строки программного кода, связанные с открытием и закрытием файла. В процедурах вывода Write появился дополнительный параметр, идущий первым – имя файловой переменной. Теперь вы умеете направлять вывод в текстовый файл. Главное – не забывать в нужных местах вводить разделители записей, выполняя смену строки.

Файл мы получили легко, но извлечь из него данные с тем, чтобы загрузить их в другую программу, – задача не всегда столь простая. Мы вернемся к этому вопросу, когда рассмотрим чтение данных из файла.

9.4.1.1 Write – и это все?

К счастью, нет. Иначе бы мы не слишком продвинулись по отношению к базовому Паскалю. Конечно, нельзя применить расширение .Print, умеющее встраиваться в цепочки, но разработчики все же позаботились о пользователях. Пусть fn – переменная, содержащая имя файла или, при необходимости, полный путь к файлу, тогда:

- Любители ООП-стиля вместо Write(f, a, b, c, …), где f – переменная типа Text, a, b, c, … – выводимые элементы, могут писать f.Write(a, b, c, …);
- WriteFormat(f, cf, params) – метод, выводящий в файл, связанный с файловой переменной типа Text, результат составного форматирования данных, представленных строкой params с использованием формата cf (см. 6.2.26.4);

- WritelnFormat(f, cf, params) – то же, что и предыдущий метод, но в конце дополнительно делается смена строки;
- WriteAllLines(fn, a) – процедура выводит в файл fn содержимое массива строк a. Если файл fn существует, он будет предварительно очищен. Вывод осуществляется в кодировке Windows по умолчанию (Encoding.Default);
- WriteAllLines(fn, ss, en) – процедура выводит в файл fn содержимое массива строк a. Если файл fn существует, он будет предварительно очищен. Вывод осуществляется в кодировке en;
- WriteLines(fn, ss) – процедура выводит в файл fn содержимое последовательности строк ss. Если файл fn существует, он будет предварительно очищен. Вывод осуществляется в кодировке Windows по умолчанию (Encoding.Default);
- WriteLines(fn, ss, en) – процедура выводит в файл fn содержимое последовательности строк ss. Если файл fn существует, он будет предварительно очищен. Вывод осуществляется в кодировке en;
- WriteAllText(fn, s) – процедура выводит в файл fn содержимое строки s. Если файл fn существует, он будет предварительно очищен. Вывод осуществляется в кодировке Windows по умолчанию (Encoding.Default);
- WriteAllText(fn, s, en) – процедура выводит в файл fn содержимое строки s. Если файл fn существует, он будет предварительно очищен. Вывод осуществляется в кодировке en.

Переменная, обозначенная en, предварительно определяется в программе (см. 9.3.1.1).

В процедурах WriteAllLines, WriteLines и WriteAllText используется строка, которая может содержать любые символы, в том числе и символы, обычно являющиеся признаками смены строки (#13#10 или #10). Это позволяет отождествить с единственной строкой даже целый текстовый файл.

Если вам лень запоминать, что WriteAllLines работает с массивом, а WriteLines – с последовательностью, можете пользоваться прямым обращением к методу библиотеки Microsoft.NET Framework, указывая в качестве s массив или последовательность:

```
System.IO.File.WriteAllLines(fn,s);
```

Следует отметить, что WriteLines и WriteAllLines выводят каждый элемент последовательности (массива) с новой строки. Можете убедиться в этом сами:

```
begin
  WriteLines('Test.txt', SeqGen(10, i -> (2 * i + 1).ToString))
end.
```

9.4.1.2 Отзвуки прошлого (файлы input/output)

Некогда было весьма популярно перенаправлять стандартные ввод и вывод на текстовые файлы. Для этого достаточно вставить в нужном месте вызов по типу

```
Assign(input, 'MyInput.txt') или Assign(output, 'MyView.txt')
```

Никаких файловых переменных, открытий и закрытий файлов!

При выводе данных из хорошего тут простота: посмотрели на мониторе, как идет вывод, добавили Assign – вот и все изменения для вывода в файл. Из не очень хорошего – в PascalABC.NET вывод пойдет в кодировке UTF-8 (Unicode).

При вводе данных для отладки программы создали текстовый файл, перенаправили с помощью Assign ввод на чтение из этого файла и избавились от повторного ввода с клавиатуры. Отладили программу – удалили Assign. Как говорится, простенько и со вкусом. Но и в этой бочке меда имеется своя ложка дегтя. Файл должен создаваться в однобайтной кодировке, а если там есть «национальные символы» (например, кириллица) – то в кодировке Encoding.Default.

9.4.2 Запись данных в типизированный файл

В типизированном файле все записи имеют одинаковый тип, а сами данные хранятся во внутреннем представлении. Наиболее часто для просмотра содержимого типизированных файлов используется отображение данных в шестнадцатеричном представлении.

Как и в случае с текстовыми файлами, научиться выводить данные в типизированный файл несложно. С этой целью нужно открыть его в нужном режиме записи и при каждом обращении к процедуре Write в качестве первого параметра указывать имя файловой переменной, связанной с открытым файлом, не меняя остального списка вывода.

Создадим типизированный файл, содержащий единственную запись данных, взяв за основу программу из предыдущего раздела, но строку объявим как «короткую» (размерную), потому что при работе с типизированными файлами использовать динамические строки запрещено. Для типизированных файлов требование использовать только строки с фиксированной длиной – благо, поскольку иначе невозможно гарантировать фиксированную длину записи в файле.

```
// p09003
type
  T1 = record
    fld1: array[1..5] of byte;
    fld2: integer;
    fld3: real;
    fld4: array[0..1] of boolean;
    fld5: string[15];
  end;
```

```
begin
  var f: file of T1;
  Assign(f,'Test.bin');
  f.Rewrite;
  var v: T1; // буфер для обмена с файлом
  for var i := 1 to 5 do
    v.fld1[i] := 16 * i - 15;
  v.fld2 := 10;
  v.fld3 := -25.34;
  v.fld4[0] := False;
  v.fld4[1] := True;
  v.fld5 := '01234АВСDЕАБВГД';
  Write(f, v); // первой в списке идет файловая переменная
  f.Close // закрываем файл
end.
```

Рассмотрим содержимое полученного файла в шестнадцатеричном представлении.

```
0000000000: 01 11 21 31 41 0A 00 00 | 00 D7 A3 70 3D 0A 57 39
0000000010: C0 00 01 0F 30 31 32 33 | 34 41 42 43 44 45 C0 C1
0000000020: C2 C3 C4                |
```

Значения 01_{16}, 11_{16}, … 41_{16} – не что иное, как однобайтные представления чисел 1, 17, … 65. Далее записывалось значение 10 в формате числа типа **integer**. В процессорах типа Intel такие числа хранятся в виде четырех байт, причем в порядке от младших байт к старшим. 0A 00 00 00 – это $0000000A_{16}=10_{10}$. Следующее значение, которое помещалось в файл – это величина -25.34 в формате представления **real** – восьмибайтного с плавающей точкой и в соответствии со стандартом IEEE754. Воспользуемся одним из имеющихся в Интернет онлайн-калькуляторов, и найдем для величины -25.34 и типа «Double precision 64-bit» значение $C039570A3D70A3D7_{16}$. Снова запишем его справа налево, окончательно получая D7 A3 70 3D 0A 57 39 C0, что полностью совпадает с информацией в файле. Следующие два байта – это 00 и 01. Именно так в PascalABC.NET представляются логические константы False и True – нулем и единицей соответственно. Далее следует байт-дескриптор со значением 0F – целочисленная переменная типа byte, хранящая длину строки ($0F_{16} = 15_{10}$ – в полном соответствии с описанием поля *fld5*). Вслед за дескриптором записано содержимое строки. Кодировка не была указана, поэтому моя Windows в соответствии с локализацией использовала Encoding.Default, т.е. 1251.

Проведенный анализ подтверждает все ранее написанное о типизированных файлах. Запись помещается в файл без каких-либо разделителей полей. Разделители между записями также отсутствуют. Поля записи представляют собой внутреннее содержимое переменных. Для интерпретации содержимого файла необходимо знать структуру записи.

Запись типа T1 по расчетам должна иметь длину 5×1+4+8+2×1+1+15 = 35 байт. И на самом деле ее длина именно такова.

В программе переменная v названа ***буфером***. Это достаточно популярный при рассмотрении обмена с файлами термин. Для чего же, собственно, нужен буфер?

Обмен с файлом всегда производится блоками некоторой длины. Конечно, можно было бы обмениваться и побайтно, но такой обмен несет большую нагрузку для файловой системы, а также резко снижает эффективность использования процессора вследствие его простоя. Простая аналогия: нужно срочно наполнить ведро водой. Как это эффективнее сделать – бегая между источником воды и ведром со стаканчиком, или нося воду кувшином? Стаканчик и кувшин тут играют ту же роль посредника, что и буфер, а их носитель – роль файловой системы. Чем буфер больше, тем быстрее и эффективнее обмен. Если нам надо поместить в файл данные типа T, то мы создаем переменную этого типа, которая будет служить буфером, заполняем ее данными, а потом помещаем содержимое буфера в файл при помощи операции записи. И это намного эффективнее, чем писать в файл отдельные байты или значения полей в случае, если тип T - **record**. В приведенном выше примере мы за одну операцию помещаем в файл 35 байт, что существенно эффективнее побайтовой записи.

Следует заметить, что приведенные рассуждения корректны в рамках исполняемой программы. Файлы находятся в ведении операционной системы и она на своем уровне вводит собственную буферизацию, о которой мы имеем право ничего не знать. А если еще вспомнить, что устройства памяти имеют собственную аппаратную буферизацию на уровне контроллера, то неопределенность становится еще выше. Но в любом случае, мы исключаем хотя бы цикл побайтной записи на уровне своего программного кода.

Но как быть, если нам требуется поместить в типизированный файл символьные данные, длина которых может изменяться? В этом случае мы должны задать длину строки так, чтобы в ней самое длинное значение данных. Для более коротких данных часть байт останется пустой.

```
// p09004
type
  T1 = record
    fld1: array[1..5] of byte;
    fld2: integer;
    fld3: real;
    fld4: array[0..1] of boolean;
    fld5: string[20];
  end;
```

```
begin
  var f: file of T1;
  Rewrite(f, 'Test.bin');
  var v: T1; // буфер для обмена с файлом
  for var i := 1 to 5 do
    v.fld1[i] := 16 * i - 15;
  v.fld2 := 10;
  v.fld3 := -25.34;
  v.fld4[0] := False;
  v.fld4[1] := True;
  v.fld5 := '01234АВСDЕАБВГД';
  Write(f, v); // первой в списке идет файловая переменная
  f.Close // закрываем файл
end.
```

Содержимое полученного файла также рассмотрим в шестнадцатиричном представлении.

```
0000000000: 01 11 21 31 41 0A 00 00 | 00 D7 A3 70 3D 0A 57 39
0000000010: C0 00 01 0F 30 31 32 33 | 34 41 42 43 44 45 C0 C1
0000000020: C2 C3 C4 00 00 00 00 00
```

Первые 19 байт ожидаемо не изменились. А за ними в файл добавился еще 21 байт. 20 байт – это понятно, мы описали короткую строку. И еще один байт-дескриптор со значением $0F_{16} = 15_{10}$ перед строкой. Неиспользованные байты оказались заполнены двоичными нулями. Так пишутся короткие строки, чтобы потом их можно было корректно считывать. Общая длина записи составляет 40 байт.

9.4.3 Запись данных в бестиповый файл

Бестиповый файл – свалка байт (в хорошем смысле этого термина). Хорошее состоит в том, что мы сами определяем, откуда читать очередную порцию байт, какой длины будет эта порция и как она должна интерпретироваться в программе. Полная свобода для программиста! Но, как известно, «Кому много дано, с того и спросится много».

Рассмотрим шестнадцатиричное представление содержимого некоторого файла. Возникают ли у Вас идеи, как его интерпретировать?

```
66 66 66 66 66 66 32 40 00 00 00 00 00 00 28 C0 98 99 99 99 99 99 19 40
```

66 66 66 66 66 66 32 40 – это записанное для процессора типа Intel значение величины типа **real**, равной 1.83999999999999985789145284798E1 ≈ 18.4.

00 00 00 00 00 00 28 C0 – аналогичным образом представленное значение величины типа **real**, равной -1.2E1 = -12.0

98 99 99 99 99 99 19 40 – всего лишь сумма этих двух чисел, равная 6.4, но в указанном представлении она определена как 6.39999999999999946709294817992E0.

С другой стороны, ничто не мешает нам эти 24 байта группировать как угодно и давать при этом самые разные интерпретации результатов такой группировки. Например, как шесть чисел типа **integer**. Так что программист при работе с бестиповыми файлами должен быть очень аккуратен и внимателен.

Создадим бестиповый файл, содержащий одну запись данных типа T1, взяв за основу одну из приведенных выше программ.

```
// p09005
type
  T1 = record
    fld1: array[1..5] of byte;
    fld2: integer;
    fld3: real;
    fld4: array[0..1] of boolean;
    fld5: string[20] // короткая строка, как в p09004
  end;


begin
  var f := CreateBinary('Test.bin'); // создаем бестиповый файл
  var v: t1;
  for var i := 1 to 5 do
    v.fld1[i] := 16 * i - 15;
  v.fld2 := 10;
  v.fld3 := -25.34;
  v.fld4[0] := False;
  v.fld4[1] := True;
  v.fld5 := '01234АВСDЕАБВГД';
  Write(f, v); // первой в списке идет файловая переменная
  f.Close // закрываем файл
end.
```

Рассмотрим содержимое полученного файла в шестнадцатеричном представлении.

```
0000000000: 01 11 21 31 41 0A 00 00 | 00 D7 A3 70 3D 0A 57 39
0000000010: C0 00 01 0F 30 31 32 33 | 34 41 42 43 44 45 C0 C1
0000000020: C2 C3 C4
```

Содержимое файла очень похоже на полученное в результате работы программы p09004 за небольшим исключением: в его конце нет пяти обнуленных байтов. Но ведь поле *fld5* рассчитано на двадцать символов, почему же в файле размещены только первых пятнадцать, которым были присвоены значения?

Давайте посмотрим, что произойдет, если следующая запись в файле будет содержать строку иной длины и с этой целью добавим в программу пару строк.

```
// p09006
type
  T1 = record
    fld1: array[1..5] of byte;
    fld2: integer;
    fld3: real;
    fld4: array[0..1] of boolean;
    fld5: string[20]
  end;


begin
  var f := CreateBinary('Test.bin'); // создаем бестиповый файл
  var v: T1;
  for var i := 1 to 5 do
    v.fld1[i] := 16 * i - 15;
  v.fld2 := 10;
  v.fld3 := -25.34;
  v.fld4[0] := False;
  v.fld4[1] := True;
  v.fld5 := '01234АВСDЕАБВГД'; // строка из 15 символов
  Write(f, v); // первой в списке идет файловая переменная
  v.fld5 := '12345'; // теперь в строке 5 символов
  Write(f, v);
  f.Close// закрываем файл
end.
```

И вот результат:

```
0000000000: 01 11 21 31 41 0A 00 00 | 00 D7 A3 70 3D 0A 57 39
0000000010: C0 00 01 0F 30 31 32 33 | 34 41 42 43 44 45 C0 C1
0000000020: C2 C3 C4 01 11 21 31 41 | 0A 00 00 00 D7 A3 70 3D
0000000030: 0A 57 39 C0 00 01 05 31 | 32 33 34 35
```

Начало второй записи (байт с кодом 01) подчеркнуто. И снова поле, содержащее символьное значение, начинается с байта, описывающего его текущую длину, а далее следует его фактическое содержимое. Длина второй записи – 25 байт. Как видите, бестиповые файлы даже при записи из буфера фиксированной длины, могут состоять из блоков данных, имеющих переменную длину.

Но если поле *fld5*, описанное как короткая строка с длиной 20 символов, ведет себя как строка с переменной длиной, нельзя ли отказаться от архаизма и использовать при описании этого поля обычную строку? Без проблем, бестиповому файлу все равно. И результат при этом будет точно такой же. Быть может, можно и массив объявить динамическим? К сожалению, нельзя. Динамический массив, а также иные объекты ссылочного типа нельзя выводить в бестиповый файл и подобную попытку пресекает компилятор. Да может оно и к лучшему: вывели бы, а как потом это считывать из мешанины байт?

9.5 Чтение данных из файла

Рассмотрим, как в PascalABC.NET читать данные из файлов. Предстоит понять, как считывать сами данные и как потом их правильно интерпретировать.

Надо понимать, что в файле нет информации о количестве помещенных в него записей. Операционная система может нам подсказать длину файла в байтах – но и только. Следовательно, циклический процесс чтения записей может продолжаться либо до получения потребного их количества, либо до исчерпания файла (до наступления события «Конец файла»). Чтение данных производится посредством обращения к операционной системе. Она же сигнализирует программе, что данные в файле исчерпаны. Игнорирование необходимости отслеживать при чтении файла момент достижения его конца обычно приводит к неожиданным аварийным завершениям программы («А на контрольном примере у меня все работало!»).

Операция чтения данных из файла заключается в извлечении из него порции (блока) некоторых данных. Если файл обрабатывается как текстовый, данные извлекаются построчно. Из типизированного файла данные извлекаются по одной записи. Бестиповый файл читается блоками байт указываемой длины.

9.5.1 Чтение данных из текстового файла

Записи в текстовом файле отделяются друг от друга символами-разделителями (см. 9.1). За размещение символа-разделителя в файле отвечает программист, используя при выводе константу NewLine, или вызывая процедуру Writeln (Println), завершающую вывод сменой строки. Нужно быть особо внимательным в случаях, когда используется двухбайтовая кодировка символов Unicode, поскольку неверное истолкование разделителя может быть чревато зацикливанием программы. Если файл изначально записывался в Unicode, проблем не возникнет: даже ошибочное указание кодовой таблицы будет проигнорировано, поскольку номер кодовой таблицы записан в начале файла.

Текстовый файл открывается на чтение (см. 9.3.1.2). Это может быть сделано явно при помощи процедуры Reset, либо неявно при использовании продвинутых средств PascalABC.NET для чтения текстовых файлов.

Для чтения данных из файла достаточно при обращении к процедурам Read/Readln в качестве первого элемента списка ввода указать имя файловой переменной, связанной с открытым файлом. При этом, если считываются не символьные данные, использование Read и Readln эквивалентно. О чтении символьных уже все было сказано в части 6. Так что настоящие проблемы не в считывании, они – в интерпретации считанного.

Попробуем написать простейшую программу, которая создаст текстовый файл, поместит туда значения двух переменных целочисленных переменных (a=1 и b=2), а затем прочитает записанные в файл данные и выведет их на монитор. Используем кодировку по умолчанию.

Вначале посмотрите на образец программы p09007, содержащей элементарную ошибку. К сожалению, такие ошибки при работе с текстовыми файлами нередки. Задержите взгляд на тексте программы: вы видите эту ошибку? С виду все хорошо: пишем a и b, потом читаем эти же a и b. Но… действительно эти же?

```
begin // p09007
  var f := OpenWrite('Test.txt');
  var (a, b) := (1, 2);
  Writeln(f, a, b);
  f.Close;
  f := OpenRead('Test.txt');
  Readln(f, a, b);
  Println(a, b);
  f.Close
end.
```

Запустив программу на выполнение, получим сообщение: «Ошибка времени выполнения: Попытка считывания за концом текстового файла». Это означает, что оператор Readln пытался что-то читать, а читать было уже неоткуда. Давайте посмотрим, что содержит созданный программой файл Test.txt.

```
0000000000: 31 32 0D 0A                |                        12♪◙
```

В символьном виде это «12» с последующими разделителями #13#10. Теперь все понятно: значение 12 прочиталось в переменную a, после чего последовал разделитель, а значения для b уже не нашлось. Наша ошибка в том, что мы «забыли» вывести разделитель между значениями a и b и на выводе эти значения «слиплись». Поправим программу и снова запустим на выполнение. Теперь все работает как нужно.

```
begin // p09008
  var f := OpenWrite('Test.txt');
  var (a, b) := (1, 2);
  Writeln(f, a, ' ', b);
  f.Close;
  f := OpenRead('Test.txt');
  Readln(f, a, b);
  Println(a, b);
  f.Close
end.
```

Вывод из приведенного примера все тот же: когда пишете данные в файл, думайте о том, что их оттуда придется считывать.

9.5.1.1 Средства чтения данных из текстового файла

Прежде всего, это «классический подход» к чтению данных из файла, предварительно открытого на чтение. Данные в файле могут разделяться пробелами, знаками табуляции и символами смены строки (строки не могут раздеться пробелами, поскольку пробел – это просто символ, который может входить в строку).

- Read(f, d1, d2, ...) – процедура читает данные из файла, связанного с файловой переменной f, и помещает их в переменные d1, d2 и т.д. Не должна использоваться для чтения данных строкового типа;

- Readln(f, d1, d2, ...) – то же, может использоваться для чтения данных любого типа.

Когда файл уже открыт на чтение, связан с некоторой файловой переменной, можно воспользоваться одним из следующих методов

- `Файловая_переменная.Lines` – возвращает содержимое файла, как последовательность строк. Возьмите себе на заметку!;
- `Файловая_переменная.ReadToEnd` – считывает оставшуюся часть файла, как последовательность строк. И это тоже стоит для себя отметить;
- `Файловая_переменная.Readln` – переходит к следующей записи.
- `Файловая_переменная.ReadBoolean` – делает попытку прочитать из файла очередной элемент, как значение типа **`boolean`**. При неудаче возникает исключение с выдачей сообщения «Ошибка времени выполнения: Входная строка имела неверный формат»;
- `Файловая_переменная.ReadlnBoolean` – то же, но после считывания осуществляется переход к следующей записи;
- `Файловая_переменная.ReadChar` – читает из файла один очередной символ;
- `Файловая_переменная.ReadlnChar` – то же, но после считывания осуществляется переход к следующей записи;
- `Файловая_переменная.ReadInteger` – делает попытку прочитать из файла очередной элемент, как значение типа **integer**. При неудаче возникает исключение с выдачей сообщения «Ошибка времени выполнения: Входная строка имела неверный формат»;
- `Файловая_переменная.ReadlnInteger` – то же, но после считывания осуществляется переход к следующей записи;
- `Файловая_переменная.ReadReal` – делает попытку прочитать из файла очередной элемент, как значение типа **real**. При неудаче возникает исключение с выдачей сообщения «Ошибка времени выполнения: Входная строка имела неверный формат»;
- `Файловая_переменная.ReadlnReal` – то же, но после считывания осуществляется переход к следующей записи;
- `Файловая_переменная.ReadString` – читает запись с текущей позиции до конца строки как значение типа **string**;
- `Файловая_переменная.ReadlnString` – полный аналог ReadString;
- `Файловая_переменная.ReadWord` – читает из файла очередное слово. Считается, что слово ограничивается любым количеством пробельных символов (см. 6.1.1.4). Отличная вещь!

Удобство работы с методами .Lines и .ReadToEnd заключается в том, что данные из файла, связанного с файловой переменной f, можно читать как текстовые строки, последовательно помещая в переменную s типа **string** каждую запись при помощи цикла. В теле цикла каждая строка при необходимости подвергается преобразованию с целью извлечения из нее данных.

```
foreach var s in f.Lines do
```

Метод .ReadWord позволяет разбирать строки и даже целый файл на слова. Это отличное начало для текстового анализа содержимого файла.

Но и это еще не все! PascalABC.NET предлагает функции, которые позволяют работать с текстовым файлом, как с единым целым и не требуют при этом не только отрывать файл, но даже создавать файловую переменную.

- ReadAllLines(fn) – функция возвращает массив элементов типа **string**, в котором каждая строка представляет собой запись из файла с именем fn. Если файл с таким именем не найден, возникает исключение. Чтение осуществляется в кодировке Windows по умолчанию (Encoding.Default), если в файле отсутствует указание кодировки Unicode, либо в кодировке, указанной в файле;
- ReadAllLines(fn, en) – то же, но если кодировка в файле не указана, она задается параметром en;
- ReadLines(fn) – функция возвращает последовательность элементов типа **string**, в которой каждая строка представляет собой запись из файла с именем fn. Если файл с таким именем не найден, возникает исключение. Чтение осуществляется в кодировке Windows по умолчанию (Encoding.Default), если в файле отсутствует указание кодировки Unicode, либо в кодировке, указанной в файле. В один момент времени в памяти хранится только одна строка;
- ReadLines(fn, en) – то же, но если кодировка в файле не указана, она задается параметром en;
- ReadAllText(fn) – функция возвращает единственную строку типа **string**, в которой размещаются все записи файла с именем fn. Если файл с таким именем не найден, возникает исключение. Чтение осуществляется в кодировке Windows по умолчанию (Encoding.Default), если в файле отсутствует указание кодировки Unicode, либо в кодировке, указанной в файле. Полученная строка содержит последовательности «конец строки», разделяющие фрагменты, соответствующие каждой записи в файле;
- ReadAllText(fn, en) – то же, но если кодировка в файле не указана, она задается параметром en.

Переменная, обозначенная en, предварительно определяется в программе (см. 9.3.1.1).

Рассмотрим содержательный пример. Пусть в файле MyTest.txt находится следующий текст (вы можете создать его в текстовом редакторе):

```
В чащах юга жил бы цитрус? Да, но фальшивый экземпляръ!
Съешь же ещё этих мягких французских булок, да выпей чаю!
Флегматичная эта верблюдица жуёт у подъезда засыхающий горький шиповник
The quick brown fox jumps over the lazy dog
The five boxing wizards jump quickly
```

Эти строки часто используют, чтобы проверить клавиатуру или принтер, поскольку каждая из них содержит «все» буквы русского или латинского алфавита. Давайте проверим, так ли это? Нам потребуется в каждой строке выделить буквы, перекодировать их к одному (например, нижнему) регистру, удалить дубликаты букв и

убедиться, что в первых трех строках останутся по 33 символа, а в двух последних – по 26. Конечно, эту задачу не так уж сложно решить и в базовом Паскале, но мы же знакомимся с современным программированием!

```
begin
  foreach var s in ReadLines('MyTest.txt') do
    s.ToLower.ToCharArray.Distinct.Where(c -> c.IsLetter).Count.Print
end.
```

На выводе получим строку: 32 33 33 26 26

С латинским шрифтом все в порядке, а вот цитрус оказался действительно фальшивым, не обеспечив в кириллице одну букву.

В предложенном варианте решения цикл **foreach** позволяет последовательно перебрать все записи файла. Для каждой записи, которая будет помещена в переменную *s*, осуществляется перекодировка к нижнему регистру, преобразование в массив char, удаление дубликатов, превращающее массив в последовательность символов, фильтрация «буквенных» символов, подсчет длины полученной последовательности и последующий вывод этой длины.

Задача, уровнем существенно выше школьного ЕГЭ, решена в пару строк. Для сравнения ниже приведена программа на Free Pascal (ее можно выполнить в среде PascalABC.NET), решающая эту же задачу. Можно ли сделать программу короче? Да, если написать ее в виде консольного приложения на Delphi или в Lazarus. Free Pascal не умеет корректно преобразовывать кириллицу к нижнему регистру при помощи функции LowerCase, да и в кодовой таблице буквы «Ё» и «ё» стоят особняком. Поэтому приходится все делать «руками» (на самом деле, «головой», конечно).

```
var
  f: Text;
  s: string;
  ac: array[0..32] of boolean;
  al: array[0..25] of boolean;
  i, k: integer;
  c: char;
```

```
begin
  Assign(f, 'MyTest.txt');
  Reset(f);
  while not Eof(f) do
  begin
    for i := 0 to 32 do
      ac[i] := False;
    for i := 0 to 25 do
      al[i] := False;
    Readln(f, s);
    for i := 1 to Length(s) do
    begin
      c := s[i];
      if c in ['A'..'Z'] then c := Chr(Ord(c)+32);
      if c in ['a'..'z'] then al[Ord(c)-Ord('a')] := True
      else
      begin
        if c in ['A'..'Я'] then c := Chr(Ord(c)+32)
        else
          if (c = 'Ё') or (c = 'ё') then ac[32] := True;
        if c in ['a'..'я'] then ac[Ord(c)-Ord('a')] := True
      end
    end;
    k := 0;
    for i := 0 to 25 do
      if al[i] then k := k + 1;
    for i := 0 to 32 do
      if ac[i] then k := k + 1;
    Write(k,' ')
  end;
  Close(f);
end.
```

Тридцать девять строк кода против четырех. Ну как, вы все еще ностальгируете по базовому Паскалю?

9.5.1.2 Обработка ситуации «конец файла» для текстовых файлов

При попытке чтения данных за концом файла возникает исключение. Избежать его возникновения можно путем отслеживания исчерпания записей в файле.

- Eof(f) – функция, возвращающая True, если достигнут конец файла, связанного с файловой переменной f, и False в противном случае:
- f.Eof – метод расширения файловой переменной f, делающий то же самое.

Традиционно, при работе с файлами используется цикл **while**:

```
while not f.Eof do // пока не достигнут конец файла
begin
   . . .
end;
```

9.5.1.3 О пробельных символах в конце записи и файла

Для файла, уже открытого на чтение и связанного с некоторой файловой переменной, можно отследить две ситуации, связанные с наличием пробельных символов:

- `Файловая_переменная.SeekEof` – пропускает все подряд идущие в направлении к концу файла пробельные символы, начиная с текущей позиции файлового указателя. Если при этом достигается конец файла, возвращается значение True, иначе возвращается значение False. Метод указывает, можно ли еще хоть что-то полезное прочитать из открытого файла;
- `Файловая_переменная.SeekEoln` – пропускает все подряд идущие в направлении к концу текущей записи (строки) пробельные символы, начиная с текущей позиции файлового указателя. Если при этом достигается конец записи, возвращается значение True, иначе возвращается значение False. Метод указывает, можно ли еще хоть что-то полезное прочитать из текущей строки.

9.5.2 Чтение данных из типизированного файла

Я подозреваю, что если существует Ад, то для грешных душ программистов там есть специальное наказание: вечно читать чужие типизированные файлы. Выше уже отмечались проблемы и ограничения, которые такие файлы накладывают. Также, давалась рекомендация открывать бинарный файл, как типизированный только в тех случаях, когда требуется быстрый произвольный доступ к определенным записям. В прочих случаях в PascalABC.NET удобнее читать файл, как бестиповый.

Для чтения из типизированного файла обычно описывается буферная переменная, имеющая тип, подходящий под структуру записи данных в файле. Именно с этой целью требуется знать структуру данных, использованную в процессе записи в файл.

9.5.2.1 Средства чтения данных из типизированного файла

Старая, проверенная временем процедура Read умеет работать и с типизированными файлами. Она достаточно удобна при чтении данных с описанием **record**, поскольку обеспечивает простое заполнение данными всех полей записи. Если f – файловая переменная, связанная с открытым на чтение типизированным файлом, а v – буферная переменная, имеющая необходимый для чтения записи тип, то ее заполнение обеспечивается вызовом процедуры Read(f, v). Для демонстрации работы с этой процедурой воспользуемся модифицированной программой p09004 (см. 9.4.1.2).

```
// p09009
type
  T1 = record
    fld1: array[1..5] of byte;
    fld2: integer;
    fld3: real;
    fld4: array[0..1] of boolean;
    fld5: string[20]; // разрешены только короткие строки
  end;

begin
  // создаем файл
  var f: file of T1;
  Rewrite(f, 'Test.bin');
  var v: T1; // буфер для обмена с файлом
  for var i := 1 to 5 do
    v.fld1[i] := 16 * i - 15;
  v.fld2 := 10;
  v.fld3 := -25.34;
  v.fld4[0] := False;
  v.fld4[1] := True;
  v.fld5 := '01234ABCDEАБВГД';
  Write(f, v); // первой в списке идет файловая переменная
  v.fld2 := 20;
  v.fld4[0] := True;
  v.fld4[1] := False;
  v.fld5 := 'Вторая запись';
  Write(f, v);
  f.Close;
  // читаем созданный файл
  Reset(f, 'Test.bin');
  while not f.Eof do
  begin
    Read(f, v); // та самая процедура, читающая запись в буфер.
    Writeln(v)
  end;
  f.Close
end.
```

На выводе получим две строки:

```
([1,17,33,49,65],10,-25.34,[False,True],01234ABCDEАБВГД)
([1,17,33,49,65],20,-25.34,[True,False],Вторая запись)
```

Отметьте, сколько операторов потребовалось, чтобы заполнить буфер данными перед записью в файл и сколько – при обмене с файлом. Удобство чтения и записи – одна из причин, по которой используются типизированные файлы. Вторая – возможность быстрого подвода требуемой записи – будет рассмотрена позднее.

Помимо процедуры Read имеются расширения для файловой переменной, связанной с открытым для чтения типизированным файлом.

- `файловая_переменная.Elements&<T>` – возвращает последовательность всех записей файла;
- `файловая_переменная.Read&<T>` – считывает и возвращает следующую запись файла;
- `файловая_переменная.Read2&<T>` – считывает и возвращает две следующих записи файла в виде кортежа элементов типа T;
- `файловая_переменная.Read3&<T>` – считывает и возвращает три следующих записи файла в виде кортежа элементов типа T;
- `файловая_переменная.ReadElements&<T>` – возвращает последовательность записей файла от текущей до конечной;
- `файловая_переменная.ReadElements&<T>(fn)` – открывает файл с именем fn, возвращает последовательность его записей и закрывает файл.

К сожалению, использование в программе расширений для файловой переменной налагает на использованные типы данных ряд ограничений, в числе которых запрет на использование строк и других ссылочных типов. Не могут также использоваться типы данных, являющиеся статическими массивами.

Рассмотрим пример, в котором создается типизированный файл, содержащий данные о координатах вершин некоторого количества треугольников, лежащих на плоскости. Каждая вершина задается точкой, имеющей две координаты. Созданный файл читается с целью нахождения величины максимального периметра. Координаты точек – целые числа, сгенерированные случайным образом в диапазоне от -99 до 99.

Записи файла имеют тип Треугольник, в котором каждое поле имеет тип Вершина. Определены также функции, позволяющие создавать вершины и треугольники, а также вычислять длину стороны треугольника и его периметр. Использование в программе привычных русскоязычных имен освобождает от необходимости давать подробные комментарии. После запуска программа запрашивает требуемое количество треугольников, создает их, а затем помещает в типизированный файл с именем Test.bin. Нахождение периметра с максимальным значением сводится к чтению записей файла в последовательность данных типа Треугольник, проецированию их в последовательность значений вычисляемых периметров и получение максимального значения последовательности. Найденное значение является частью интерполированной строки, направляемой на вывод. Закрывать файл после записи и повторно открывать не требуется, поскольку открытый на запись файл также считается открытым и на чтение.

```
// p09010
type
  Вершина = record
    x, y: integer
  end;

  Треугольник = record
    A, B, C: Вершина
  end;

function Длина(A, B: Вершина) :=
  Sqrt((B.x - A.x) ** 2 + (B.y - A.y) ** 2);

function Периметр(p: Треугольник) :=
  Длина(p.A, p.B) + Длина(p.B, p.C) + Длина(p.C, p.A);

function СоздатьВершину: Вершина;
begin
  (Result.x, Result.y) := Random2(-99, 99);
end;

function СоздатьТреугольник: Треугольник;
begin
  Result.A := СоздатьВершину;
  Result.B := СоздатьВершину;
  Result.C := СоздатьВершину
end;

begin
  var n := ReadInteger('Количество треугольников:');
  var f := CreateFile&<Треугольник>('Test.bin');
  loop n do
    Write(f, СоздатьТреугольник);
  $'Pmax={f.Elements.Select(t->Периметр(t)).Max:f1}'.Println;
  f.Close
end.
```

Ниже показан вариант диалога с программой:

```
Количество треугольников: 5
Pmax=347.8
```

9.5.2.2 Ситуация «конец файла» для типизированных файлов

Как и в случае работы с текстовыми файлами, при чтении типизированного файла может возникать ситуация «конец файла» (см. 9.5.1.2). Все, сказанное по этому поводу для текстовых файлов, верно и для типизированных.

9.5.3 Чтение данных из бестипового файла

Бестиповый файл позволяет интерпретировать его содержимое произвольным образом. Мы сами решаем, где читать, сколько читать и в каком формате представ-

лять прочитанное. В конечном счете, считываются просто байты, а их последующую интерпретацию определяет выбранный инструмент (например, расширение метода). Зная, что запись в файл осуществлялась из какой-то переменной типа record, можно читать файл в переменную того же типа (по сути, работать как с типизированным файлом, но при этом получить возможность использовать и строки). Но можно читать только интересующие поля, пропуская остальное и интерпретируя считанное по собственному усмотрению.

Для чтения бестиповых файлов можно использовать все ту же процедуру Read:
Read(f, d1, d2, …), где f – имя файловой переменной, связанной с открытым бестиповым файлом, d1, d2 – имена переменных, в которых данные считываются в соответствии с типами. При этом тип данных определяет и количество считываемых из файла байт, и их интерпретацию.

Помимо процедуры Read имеются расширения для файловой переменной, связанной с открытым бестиповым файлом.

- `файловая_переменная.ReadBytes(n)` – возвращает массив типа **array of byte** длиной n;
- `файловая_переменная.ReadBoolean` – возвращает значение типа **boolean**;
- `файловая_переменная.ReadByte` – возвращает значение типа **byte**;
- `файловая_переменная.ReadChar` – возвращает значение типа **char**;
- `файловая_переменная.ReadInteger` – возвращает значение типа **integer**;
- `файловая_переменная.ReadReal` – возвращает значение типа **real**;
- `файловая_переменная.String` – возвращает значение типа **string**.

Если ни один из указанных типов не подходит, используется упомянутая выше процедура Read.

В качестве примера рассмотрим создание и последующее чтение бестипового файла, взяв за основу программу p09010 (см. 9.5.2.1). Обратите внимание, что файл в программе p09011 создается как типизированный, а считывается как бестиповый. В частности, это позволило обойти на запрет использования строк в типизированных файлах. Мы по-прежнему используем в полях записи статические массивы, чтобы не создавать себе проблем при работе с файлом. В строках используется кодировка по умолчанию. Считывание данных намеренно ведется по отдельным полям, чтобы показать использование соответствующих расширений. В то же время, далее приведена программа p09012, в которой происходит чтение сразу всей записи при помощи процедуры Read.

```
// p09011
type
  T1 = record
    fld1: array[1..5] of byte;
    fld2: integer;
    fld3: real;
    fld4: array[0..1] of boolean;
    fld5: string // строки разрешены, в файл пишется их длина
  end;

begin
  // создаем файл
  var f := CreateBinary('Test.bin');
  var v: T1; // буфер для обмена с файлом
  for var i := 1 to 5 do
    v.fld1[i] := 16 * i - 15;
  v.fld2 := 10;
  v.fld3 := -25.34;
  v.fld4[0] := False;
  v.fld4[1] := True;
  v.fld5 := '01234АВСDЕАБВГД';
  f.Write(v);
  v.fld2 := 20;
  v.fld4[0] := True;
  v.fld4[1] := False;
  v.fld5 := 'Вторая запись';
  f.Write(v);
  // читаем файл
  f.Reset;
  while not f.Eof do
  begin
    f.ReadBytes(5).Print;
    Write(' '); // после вывода массива пробел не делается
    f.ReadInteger.Print;
    f.ReadReal.Print;
    loop 2 do
      f.ReadBoolean.Print;
    f.ReadString.Println
  end;
  f.Close
end.
```

Программа выведет две строки

```
1 17 33 49 65 10 -25.34 False True 01234АВСDЕАБВГД
1 17 33 49 65 20 -25.34 True False Вторая запись
```

Следующая программа отличается лишь в части чтения файла и вывода результатов. Используются процедуры Read и Write.

```
// p09012
type
  T1 = record
    fld1: array[1..5] of byte;
    fld2: integer;
    fld3: real;
    fld4: array[0..1] of boolean;
    fld5: string // строки разрешены, в файл пишется их длина
  end;

begin
  // создаем файл
  var f := CreateBinary('Test.bin');
  var v: T1; // буфер для обмена с файлом
  for var i := 1 to 5 do
    v.fld1[i] := 16 * i - 15;
  v.fld2 := 10;
  v.fld3 := -25.34;
  v.fld4[0] := False;
  v.fld4[1] := True;
  v.fld5 := '01234ABCDEАБВГД';
  f.Write(v);
  v.fld2 := 20;
  v.fld4[0] := True;
  v.fld4[1] := False;
  v.fld5 := 'Вторая запись';
  f.Write(v);
  // читаем файл
  f.Reset;
  while not f.Eof do
  begin
    Read(f, v);
    Writeln(v)
  end;
  f.Close
end.
```

Здесь вывод результатов не форматировался:

```
([1,17,33,49,65],10,-25.34,[False,True],01234ABCDEАБВГД)
([1,17,33,49,65],20,-25.34,[True,False],Вторая запись)
```

9.6 Операции с файловым указателем

В бинарных файлах можно установить файловый указатель в начало файла, в конец файла или в произвольное место внутри файла. Именно с этой позиции будет производиться следующая операция чтения или записи. Операция перестановки файлового указателя в Паскале часто называется поиском (seek), но она не имеет никакого отношения к поиску данных. Правильнее говорить о подводе записи с нужным номером в типизированных файлах или установке файлового

указателя в нужную позицию в бестиповых файлах. С этой точки зрения мне представляется удачным выбор разработчиками PascalABC.NET имени Position для расширения, выполняющего перестановку файлового указателя в требуемое место.

Средства для перестановки файлового указателя применимы для любых бинарных файлов, но нужно помнить, что в типизированных файлах осуществляется перестановка по номерам записей, а в бестиповых – по номерам байт.

PascalABC.NET предлагает следующий набор средств для перестановки файлового указателя в предварительно открытом бинарном файле:

- Seek(`файловая_переменная, n`) – устанавливает файловый указатель на запись (или байт) с номером n;
- `файловая_переменная.Position` – хранит значение текущей позиции файлового указателя. Осуществляет перестановку файлового указателя путем присваивания нужного значения в виде `файловая_переменная.Position` := n;
- `файловая_переменная.Seek(n)` – устанавливает файловый указатель на запись (или байт) с номером n.

Значение номера позиции n имеет тип int64. Нумерация ведется от нуля.

Кроме .Position, получить значение позиции файлового указателя позволяет функция `FilePos(файловая_переменная)`. Как обычно, PascalABC.NET предоставляет вам выбор.

В программе p09010 (см. 9.5.2.1) определялось максимальное значение периметра набора треугольников, имеющих случайные координаты вершин. Посмотрите, как можно модифицировать программу для дополнительного вывода координат вершин треугольника, имеющего максимальный периметр.

```
begin
  var n := ReadInteger('Количество треугольников:');
  var f := CreateFile&<Треугольник>('Test.bin');
  loop n do
    Write(f, СоздатьТреугольник);
  f.Position := f.Elements.Select(t -> Периметр(t)).ToArray.IndexMax;
  var ТреугольникМакс := f.Read;
  $'Pmax={Периметр(ТреугольникМакс):f1}'.Println;
  Print('Координаты вершин:');
  $'A({ТреугольникМакс.A.x},{ТреугольникМакс.A.y})'.Print;
  $'B({ТреугольникМакс.B.x},{ТреугольникМакс.B.y})'.Print;
  $'C({ТреугольникМакс.C.x},{ТреугольникМакс.C.y})'.Println;
  f.Close
end.
```

Ниже приведен пример диалога с программой:

```
Количество треугольников: 5
Pmax=433.1
Координаты вершин: A(13,92) B(-92,-31) C(21,-62)
```

9.7 Прочие операции с файлами

PascalABC.NET предоставляет набор операций с файлами, относящийся к функциям операционной системы, таких как переименование файлов, их удаление, усечение до требуемой длины и т.п. При этом методы расширения, использующие файловую переменную, требуют чтобы файл был закрыт.

- `Erase(файловая_переменная)` – удаление файла, связанного с указанной файловой переменной;
- `файловая_переменная.Erase` – то же, для файла, связанного с указанной файловой переменной;
- `Rename(файловая_переменная, новое_имя)` – переименование файла, связанного с указанной файловой переменной;
- `файловая_переменная.Rename(новое_имя)` – то же, для файла, связанного с указанной файловой переменной;
- `файловая_переменная.Name` – возвращает имя файла, связанного с указанной файловой переменной;
- `файловая_переменная.FullName` – возвращает полное имя файла, связанного с указанной файловой переменной;
- `файловая_переменная.Size` – возвращает количество записей в типизированном файле или количество байт в бестиповом файле как значение типа **int64**;
- `файловая_переменная.Truncate` – усекает файл по текущей позиции файлового указателя;
- `файловая_переменная.Name` – возвращает имя файла, связанного с указанной файловой переменной;
- `файловая_переменная.Name` – возвращает имя файла, связанного с указанной файловой переменной.

Еще одна группа функций возвращает полное имя файла с именем fn или различные части этого имени:

- `ExpandFileName(fn)` – возвращает полное имя файла;
- `ExtractFileDir(fn)` – возвращает выделенный из полного имени файла путь;
- `ExtractFileExt(fn)` – возвращает выделенное из полного имени файла расширение;
- `ExtractFileName(fn)` – возвращает выделенное из полного имени файла имя;
- `ExtractFilePath(fn)` – возвращает выделенный из полного имени файла путь (отличается от ExtractFileDir наличием обратного слэша в качестве последнего символа).

Ниже приведены примеры использования некоторых операций для файла с полным именем C:\PABCWork.NET\Samples\LINQ\Linq1.pas:

```
ExpandFileName(fn).Println; // C:\PABCWork.NET\Samples\LINQ\Linq1.pas
ExtractFileDir(fn).Println; // C:\PABCWork.NET\Samples\LINQ
ExtractFileExt(fn).Println; // .pas
ExtractFileName(fn).Println; // Linq1.pas
ExtractFilePath(fn).Println; // C:\PABCWork.NET\Samples\LINQ\
```

9.8 Рекомендации по работе с файлами

Приводимые рекомендации носят достаточно общий характер и предназначены для начинающих изучать работу с файлами.

Используйте файлы для обмена данными между программами. Это единственное средство, позволяющее без искажений передать данные большого объема.

Для отладки программы, требующей значительного объема вводимых данных, поместите их в текстовый файл с целью последующего считывания.

Типизированные файлы используйте лишь там, где без них нельзя обойтись: при необходимости активно перемещаться между записями файла. Современные компьютеры располагают достаточно большим объемом оперативной памяти, поэтому бывает намного эффективнее прочитать содержимое файла в массив или последовательность и в дальнейшем к файлу не обращаться.

Бестиповый файл позволяет достаточно эффективно скорректировать находящиеся в нем данные, если при этом сохраняется отведенное для хранимого значения количество байт. Это может быть полезно и для редактирования текстового файла, открываемого как бестиповый с целью его построчной обработки.

Зачастую вместо корректировки содержимого файла его проще считать в память, скорректировать и перезаписать файл. Это касается также случаев сжатия файла после удаления из него каких-либо элементов.

Если необходимо один файл разбить на несколько, иногда проще писать каждый фрагмент в файл с одним и тем же именем, а затем полученные файлы закрывать и переименовывать.

9.9 Для самостоятельного решения

T9.1. Создайте текстовый и бестиповый файлы. Организуйте цикл на миллион проходов. В каждом проходе по циклу генерируйте случайное вещественное значение x из интервала [-9.0; 9.0], накапливайте сумму s полученных значений и записывайте значение x в текстовый и бестиповый файл. После завершения цикла выведите значение суммы s. Затем вычислите и выведите сумму значений sf, полученных из бестипового файла и разность s – sf. Повторите эту операцию для тестового файла. Убедитесь, что полученная из текстового файла сумма имеет погрешность, а размер текстового файла превышает размер бестипового более чем вдвое. Сделайте вывод об использовании файлов при работе с данными вещественного типа.

T9.2. Создайте текстовый файл, каждая запись которого содержит тройку случайных натуральных чисел из диапазона [1; 99]. Считая эти числа сторонами треугольника, удалите из файла записи, данные которых не позволяют построить

треугольник (сумма длин двух любых сторон треугольника меньше или равна третьей стороне). Закройте отредактированный файл. Затем подсчитайте и выведите с тремя знаками после запятой среднюю площадь треугольников, созданных на основе записей файла. Исходный файл должен содержать не менее пятисот тысяч записей.

T9.3. Решить задачу T9.2, используя типизированные файлы. Должны присутствовать все три этапа работы (создание файла, удаление лишних записей, получение средней суммы площадей треугольников). Структуру записи типизированного файла определите самостоятельно.

T9.4. Даны два файла, содержащие произвольные текстовые фрагменты, взятые из литературных произведений (файлы T9.4a.txt и T9.4b.txt). Получить список слов, присутствующих в обоих текстах и состоящих не менее чем из двух букв, отсортировать его в алфавитном порядке, сохранить в текстовый файл и вывести на монитор. Регистр символов игнорировать. Слова должны содержать только символы кириллицы.

Часть 10

Стандартные коллекции

Пристрастие к коллекционированию – первая ступень умственного расстройства.
Оноре де Бальзак

Прежде чем делать открытие – загляни в справочник.
К. Прутков-инженер.
«Советы начинающему гению»

В предыдущих частях книги вы уже познакомились с последовательностями, массивами, множествами и строками. Всех их объединяет способность хранить данные некоторого типа и предоставлять набор средств для операций над ними.

В программировании используется понятие ***коллекции*** – некоторой структуры, содержащей в себе набор элементов одного или различных типов и позволяющей манипулировать этими объектами посредством установленного набора операций. Коллекция предоставляет средства для помещения в себя данных, извлечения их, а также обеспечивает доступ к данным. Реализация набора операций для манипулирования данными существенно упрощает программирование, а также облегчает человеку понимание программного кода.

Одни коллекции хранят данные в порядке их поступления, другие некоторым образом эти данные переупорядочивают. Например, последовательности хранят элементы в порядке их включения, массивы – в соответствии с индексом элемента, а множества – «неупорядоченно». Слово «неупорядоченно» взято в кавычки не случайно: на самом деле имеется совершенно четкий внутренний алгоритм хранения элементов множества, но мы не должны принимать это в расчет.

В PascalABC.NET можно напрямую обращаться к ***обобщенным*** стандартным коллекциям Microsoft .NET Framework, реализованным в виде набора классов. Термин «обобщенная» означает, что тип элементов коллекции не фиксирован. Перед тем, как рассматривать эти классы, познакомимся с еще одним понятием – ***контейнером***.

В нашем окружении контейнеры встречаются повсеместно. Это и коробки с конфетами, и сумки с вещами, и полки с книгами, и сами книги, состоящие из страниц. Все, во что можно что-то положить, упаковать или просто завернуть.

Контейнер – это тип, позволяющий хранить в себе объекты других (или такого же) типов.

Все рассматриваемые в этой части книги стандартные коллекции построены на базе контейнерных классов.

10.1 Линейные списки

Линейный список – это последовательность из нуля или более элементов (узлов), главной особенностью которого является такое относительное расположение узлов, будто они образуют одну линию. Если в линейном списке n>0 элементов, то узел j[k] (0<k<n) следует непосредственно за j[k-1] и предшествует j[k+1].

В общем случае над линейными списками могут быть определены девять операций:

1. Доступ к узлу k для просмотра или модификации хранимых данных;
2. Вставка нового узла непосредственно перед или после узла k;
3. Удаление узла k;
4. Объединение линейных списков в новом списке;
5. Разбиение списка на линейные списки;
6. Копирование списка;
7. Нахождение количества узлов списка;
8. Сортировка узлов списка в некотором порядке, определяемом содержимым записанных в узле данных;
9. Поиск узла в соответствии со значением, связанным с содержимым записанных в узле данных.

Следует выделить особые случаи, когда обрабатывается первый или последний элемент списка, поскольку доступ к таким узлам организовать проще, чем к остальным.

Программирование реальных задач редко требует использования структур данных, умеющих выполнять все девять определенных выше операций. И это понятно: чем разнообразнее набор операций с данными, который допускает выполнять информационная структура, тем она сложнее и тем сложнее с ней работать. А создание сложностей самим себе – это не наш путь. Выход состоит в использовании некоторых установившихся разновидностей линейных списков, ориентированных на решение конкретных задач. Это уже знакомые последовательности (**sequence**) и массивы (**array**). Ниже представлены еще две разновидностями линейных списков – стек (stack) и очередь (queue).

10.1.1 Стек (Stack)

О стеках превосходно сказал Д. Кнут. Сказал настолько удачно, что эти слова теперь встречаются во многих рефератах и статьях, но почему-то без ссылки на их автора. Не будем же столь неблагодарны:

«Многие исследователи независимо пришли к выводу о важности стеков и очередей, а поэтому присвоили им собственные имена. Так, стеки часто называют магазинными списками (push-down lists), реверсивными хранилищами (reversion storages), магазинами (cellars), вложенными хранилищами (nesting stores), кучами (piles), дисциплинами обслуживания в обратном порядке (last-in-first-out lists –

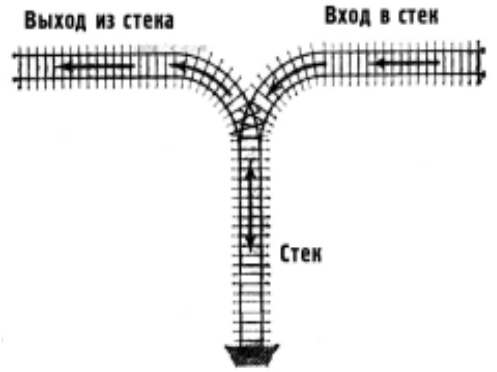
LIFO lists) и даже флюгерными списками (yo-yo lists)» [в кн. «Искусство программирования», том 1, глава 2 «Информационные структуры»].

Что же такое программный стек? Почему он так важен и нужен программистам?

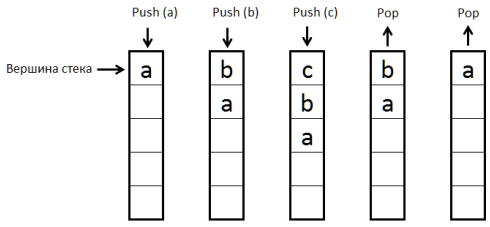
Стек (англ. Stack – стопка) – линейный список, в котором добавление и исключение элементов производится на одном его конце, называемом ***вершиной***.

Стек легко представить себе в виде стопки книг или колоды карт. Книгу или карту можно добавить только сверху (операция push – добавление). Название книги и значение карты видно только для одной книги или карты, лежащей наверху (значение top – доступ к элементу на вершине стека). Да и забрать можно лишь то, что лежит сверху (операция pop – извлечение).

Э. Дейкстра любил представлять стек в виде железнодорожного разъезда с тупиком. Вагоны закатываются в тупик (стек), а затем выкатываются из него. Последний из закаченных вагонов становится первым, что демонстрирует особенность стека: он изменяет порядок следования помещенных объектов на противоположный. Алгоритм работы стека схематически описывается фразой «Последним зашел – первым вышел» (англ. Last In – First Out, сокращенно LIFO), что оправдывает существование второго устоявшееся названия стека – список LIFO.



Рассмотрим принцип работы стека на примере помещения в него объектов «a», «b» и «c» и последующего извлечения объектов «c и «b». После выполнения операции Push(a), объект «a» появляется на вершине стека. Операция Push(b) «проталкивает» объект «a» в глубину стека и теперь на его вершине будет доступен объект «b». Для объекта «c» происходит аналогичное действие. Далее, операция Pop удаляет с верши-



ны стека объект «c», а из глубины стека «поднимается» на вершину объект «b». Затем операция Pop удаляет с вершины стека объект «b» и на вершине стека оказывается объект «a».

Безусловно, физически никакого «проталкивания» данных в стек не происходит, но это уже особенности программной реализации стека. Существует несколько способов такой реализации, каждый из которых имеет свои достоинства и недостатки. Например, реализация на основе массива проста, да и работает быстро, но взамен требует заранее определять максимальный объем стека и впоследствии следить за тем, чтобы стек не переполнялся данными. При записи данные помещаются на очередное свободное место в массиве, а индекс элемента служит указанием на вершину стека. Реализация на основе ***связанного списка*** (см. 10.2) позволяет не думать о переполнении стека, но она более сложна, требует относительно медленного динамического выделения памяти под очередной элемент данных и поддержания связей между элементами списка.

В PascalABC.NET стек реализован на основе ***обобщенной*** стандартной коллекции System.Collections.Generic.Stack. Стек является стандартным типом языка, поэтому его можно создать с использованием оператора присваивания вида

```
var имя := new Stack<тип_данных>;
```

В качестве типа данных, помещаемых в стек, можно указать любой тип, в том числе, предварительно определенный пользователем. Поскольку коллекция реализована как контейнер, типом данных, помещаемых в стек, может быть и стек.

```
type
  r1 = record
    f1: integer;
    f2: real;
    f3: array of boolean;
    f4: string
  end;

begin
  var a := new Stack<Stack<real>>;
  var b := new Stack<array[,] of real>;
  var c := new Stack<r1>;
end.
```

Конструктор стека дает возможность совместить создание стека с его заполнением элементами последовательности (или данными с типом, приводящимся к последовательности), указанной в качестве параметра. При этом порядок следования элементов в новом стеке будет обратным по отношению к принимаемым данным.

```
begin
  var st1 := new Stack<integer>(SeqGen(5,1,t->t+2));
  st1.Println; // 9 7 5 3 1
end.
```

В приведенном примере в стек st1 были помещены элементы последовательности 1, 3, 5, 7 и 9, заданной генератором нечетных чисел. Вывод содержимого стека

производится, начиная с его вершины, поэтому на мониторе отображаются значения 9 7 5 3 1. Это еще раз иллюстрирует особенность стека: он изменяет порядок следования данных на противоположный.

10.1.1.1 Операции для работы со стеком

В работе со стеком поддерживаются следующие средства:

- `.Count` – свойство, возвращающее количество элементов в стеке;
- `.Push(объект)` – метод, помещающий указанный объект на вершину стека;
- `.Peek` – метод, возвращающий объект, находящийся на вершине стека, при этом сам объект остается на месте. Может вызываться в виде функции;
- `.Pop` – метод, удаляющий объект с вершины стека. Возвращает удаляемый объект. Может вызываться в виде функции;
- `.Clear` – метод, очищающий стек;
- `.Contains(объект)` – метод, возвращающий True, если указанный объект находится в стеке и False в противном случае (проверка на наличие объекта в стеке);
- `.ToArray` – метод, копирующий содержимое стека в одномерный динамический массив и возвращающий этот массив в качестве результата. Не требует предварительного описания массива;
- `.CopyTo(имя_массива, индекс)` – метод, копирующий содержимое стека в существующий одномерный динамический массив, начиная с элемента, указанного индексом. Если данные не помещаются в массиве, возникает исключение;
- `.Print` – стандартный метод PascalABC.NET. Осуществляет вывод содержимого стека, начиная с его вершины. В качестве параметра можно указать символ или строку - разделитель, по умолчанию используется пробел;
- `.Println` – то же самое; в конце вывода осуществляется переход к новой строке.

К содержимому стека можно применять все методы для работы с последовательностями.

10.1.1.2 Примеры использования стека

Пусть дана символьная строка, содержащая некоторое выражение с круглыми скобками. Требуется определить, верно ли расставлены скобки. Предлагается следующий несложный алгоритм:

1. Стек пуст.
2. Выполняем последовательный просмотр всех символов строки.
3. Если очередной символ является круглой открывающей скобкой, заносим его на вершину стека.
4. Если очередной символ является круглой закрывающей скобкой, проверяем стек. Если он не пуст, удаляем символ с вершины, если пуст – фиксируем ошибку.
5. Если после окончания просмотра строки стек не пуст, фиксируем ошибку.

```
begin // p10001
  var St := new Stack<char>;
  foreach var c in ReadlnString('Введите выражение') do
    case c of
      '(': St.Push(c);
      ')':
        if St.Count > 0 then St.Pop
           else
        begin
          Println('Скобки расставлены неверно');
          Exit
        end
    end;
  if St.Count = 0 then Println('Скобки расставлены верно')
  else Println('Скобки расставлены неверно')
end.
```

Пример выполнения программы

```
Введите выражение (a+3*b)*((c+d)*a+1)/(a-1)
Скобки расставлены верно
```

Усложним задание. Пусть при записи выражения разрешается использовать три вида скобок – круглые, квадратные и фигурные. Конечно же, при балансе скобок следует проверить, чтобы каждой открывающей скобке соответствовала закрывающая скобка того же типа. Алгоритм усложнится, но совсем не намного. При появлении закрывающей скобки вместо проверки стека на пустоту нужно убедиться, что на вершине стека находится открывающая скобка того же типа. И все.

```
begin // p10002
  var Error := False;
  var St := new Stack<char>;
  foreach var c in ReadlnString('Введите выражение') do
  begin
    case c of
      '(', '[', '{': St.Push(c);
      ')', ']', '}':
        if St.Count > 0 then
          case St.Peek of
            '(':
              if c = ')' then St.Pop
              else Error := True;
            '[':
              if c = ']' then St.Pop
              else Error := True;
            '{':
              if c = '}' then St.Pop
              else Error := True
          end
        else Error := True
    end;
    if Error = True then Break
  end;
  if (St.Count = 0) and not Error then Println('Скобки расставлены верно')
  else Println('Скобки расставлены неверно')
end.
```

Пример выполнения программы

```
Введите выражение (a+3*b)*[(c+d)*a+1}/(a-1)
Скобки расставлены неверно
```

Рассмотрим еще одну задачу, в которой можно использовать стек. Дана строка, содержащая некоторый русский текст. Требуется получить из нее строку, в которой согласные буквы будут чередоваться с гласными в исходном порядке следования. Если количество согласных и гласных букв разное, оставшиеся буквы следует приписать к концу строки. Например, из строки «НашМаленькийТест» должна получиться строка «НашаМелинекйТст».

Создадим два стека – отдельно для согласных и гласных букв. А затем выведем их содержимое в строку, чередуя символы, извлекаемые из каждого стека. При опустошении одного из стеков допишем к строке содержимое непустого стека. Поскольку стек реализует алгоритм LIFO, извлекаемые символы нужно приписывать не к концу строки, а к ее началу.

```
begin // p10003
  var Строка := ReadlnString('Введите строку');
  var Стек1 := new Stack<char>;
  var Стек2 := new Stack<char>;
  var Согласные := 'БВГДЖЗЙКЛМНПРСТФХЦЧШЩЪЬ';
  var Гласные := 'АЕЁИОУЫЭЮЯ';
  var ПерваяГласная := Строка[1].ToUpper in Гласные;
  foreach var Символ in Строка do
    if ПерваяГласная then
      if Символ.ToUpper in Гласные then Стек1.Push(Символ)
      else Стек2.Push(Символ)
    else
    if Символ.ToUpper in Согласные then Стек1.Push(Символ)
    else Стек2.Push(Символ);
  Строка := '';
  var РазницаДлинСтеков := Стек1.Count - Стек2.Count;
  if РазницаДлинСтеков >= 0 then
  begin
    loop РазницаДлинСтеков do
      Строка := Стек1.Pop + Строка;
    loop Стек1.Count do
      Строка := Стек1.Pop + Стек2.Pop + Строка
  end
  else
  begin
    loop -РазницаДлинСтеков do
      Строка := Стек2.Pop + Строка;
    loop Стек1.Count do
      Строка := Стек1.Pop + Стек2.Pop + Строка
  end;
  Строка.Println
end.
```

Пример работы программы

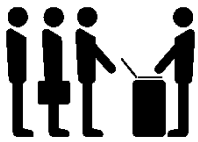
```
Введите строку НашМаленькийТест
НашаМелинеькйТст
```

Программа p1003 приведена в иллюстративных целях. В PascalABC.NET имеются достаточно мощные средства для работы с символами и строками. Ниже представлен один из вариантов решения этой же задачи без использования стеков. Программа получилась короче, и кто-то даже может счесть ее более понятной. В очередной раз подтверждается тезис о том, что полноценный и развитый язык программирования предоставляет несколько путей реализации задачи, а умение выбрать конкретные средства определяется квалификацией программиста.

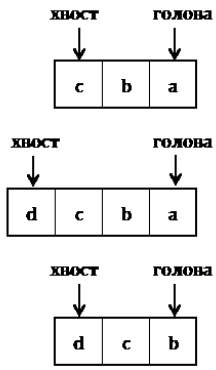
```
begin // p10004
  var Строка := ReadlnString('Введите строку');
  var Согласные := 'БВГДЖЗЙКЛМНПРСТФХЦЧШЩЪЬ';
  var Гласные := 'АЕЁИОУЫЭЮЯ';
  var Подстрока1 :=
      Строка.Where(Символ -> Символ.ToUpper in Гласные).JoinIntoString('');
  var Подстрока2 :=
     Строка.Where(Символ -> Символ.ToUpper in Согласные).JoinIntoString('');
  if Строка.First = Подстрока1.First then
    Строка := Подстрока1.Interleave(Подстрока2).JoinIntoString('')
  else
    Строка := Подстрока2.Interleave(Подстрока1).JoinIntoString('');
  var РазницаДлин := Подстрока1.Length - Подстрока2.Length;
  if РазницаДлин > 0 then
    Строка += RightStr(Подстрока1, РазницаДлин)
  else
    Строка += RightStr(Подстрока2, -РазницаДлин);
  Строка.Println
end.
```

10.1.2 Очередь (Queue)

Понятие очереди взято из реальной жизни: мы отлично знаем, что представляет из себя очередь. Обслуживание всегда производится в начале очереди, а постановка в очередь происходит в ее конце. В информатике понятия начала и конца очереди не отличаются от общепринятых. Начало очереди также называют её ***головой***, а конец – ***хвостом***. Итак, включение в очередь происходит на ее конце (в хвосте), а исключение и доступ – в начале очереди (в голове). В соответствии с дисциплиной обслуживания, очередь часто называют списком FIFO (англ. First In – First Out – первым вошел, первым вышел).



Очередь (англ. Queue) – линейный список, в котором добавление элементов производится на одном его конце, а исключение элементов (и, как правило, доступ к ним) производится на другом его конце.



На представленном рисунке схематически показано изменение состояния очереди, содержащей объекты «a», «b» и «c». Объект «d» добавляется в хвост, вызывая увеличение размера очереди, затем объект «a» удаляется из очереди, уменьшая ее размер. Как и стек, очередь можно реализовать при помощи массива или связанного списка.

В PascalABC.NET очередь реализована на основе стандартной коллекции Sys-

tem.Collections.Generic.Queue. Очередь является стандартным типом языка, поэтому его можно создать с использованием оператора присваивания вида

```
var имя := new Queue<тип_данных>;
```

В качестве типа данных, помещаемых в очередь, можно указать любой тип, в том числе, предварительно определенный пользователем. Поскольку коллекция реализована как контейнер, типом данных, помещаемых в очередь, может быть и очередь.

```
type
  r1 = record
    f1: integer;
    f2: real;
    f3: array of boolean;
    f4: string
  end;

begin
  var a := new Queue<Queue<boolean>;
  var b := new Queue<array[,] of real>;
  var c := new Queue<r1>;
end.
```

Конструктор очереди дает возможность совместить создание очереди с ее заполнением элементами последовательности (или данных с типом, приводящимся к последовательности), указанной в качестве параметра.

```
begin
  var q1 := new Queue<integer>(SeqRandom(4, 10, 99));
  q1.Println; // 54 92 42 33
  var q2 := new Queue<integer>(q1.Sorted);
  q2.Println// 33 42 54 92
end.
```

В приведенном примере очередь q1 заполняется данными от генератора случайных чисел. Очередь q2 заполняется данными из очереди q1, которые предварительно сортируются в порядке возрастания.

10.1.2.1 Операции для работы с очередью

В работе с очередью поддерживаются следующие средства:

- `.Count` – свойство, возвращающее количество элементов в очереди;
- `.Enqueue(объект)` – метод, помещающий указанный объект в конец очереди;
- `.Peek` – метод, возвращающий объект, находящийся в начале очереди. Может вызываться в виде функции;
- `.Dequeue` – метод, удаляющий объект из начала очереди. Возвращает удаляемый объект. Может вызываться в виде функции;
- `.Clear` – метод, очищающий очередь;
- `.Contains(объект)` – метод, возвращающий True, если указанный объект находится в очереди и False в противном случае (проверка на наличие объекта в очереди);

- `.ToArray` – метод, копирующий содержимое очереди в одномерный динамический массив и возвращающий этот массив в качестве результата. Не требует предварительного описания массива;
- `.CopyTo(имя_массива, индекс)` – метод, копирующий содержимое очереди в существующий одномерный динамический массив, начиная с элемента, указанного индексом. Если данные не помещаются в массиве, возникает исключение;
- `.Print` – стандартный метод PascalABC.NET. Осуществляет вывод содержимого очереди от ее начала. В качестве параметра можно указать символ или строку - разделитель, по умолчанию используется пробел;
- `.Println` – то же самое; в конце вывода осуществляется переход к новой строке.

К содержимому очереди можно применять все методы для работы с последовательностями.

10.1.2.2 Примеры использования очереди

Массив, даже динамический, требует указания размера. Очередь этого не требует. Отсюда вытекает очевидное преимущество использования очереди в задачах, где количество данных неизвестно, например, при работе с файлами или приеме данных с некоторым условием окончания приема.

Задача 1. С клавиатуры вводится последовательность натуральных чисел, ограниченная нулем. Значение чисел не превышают 999. Вывести значения поступивших чисел по группам: сначала одноразрядные, затем двухразрядные и в конце трехразрядные. Порядок следования чисел в каждой группе не должен изменяться по сравнению с порядком ввода чисел.

Для решения задачи создадим три очереди, содержимое которых затем выведем.

```
begin // p10005
  var q1 := new Queue<integer>;
  var q2 := new Queue<integer>;
  var q3 := new Queue<integer>;
  foreach var d in ReadSeqIntegerWhile(t -> t <> 0) do
    if d < 10 then q1.Enqueue(d)
    else
      if d < 100 then q2.Enqueue(d)
      else q3.Enqueue(d);
  q1.Println;
  q2.Println;
  q3.Println
end.
```

```
3 18 42 15 9 14 621 1 42 311 9 0
3 9 1 9
18 42 15 14 42
621 311
```

Задача 2. Очереди q1 и q2 содержат данные типа **integer**, отсортированные в порядке неубывания. Выполнить слияние, получив очередь q, содержащую все элементы очередей q1 и q2, расположенные в неубывающем порядке.

Здесь удобно использовать маркер конца, добавив в хвост очередей q1 и q2 максимально допустимое значение **integer**.MaxValue. Это обеспечит завершение процесса слияния по наличию в каждой очереди ровно одного элемента (маркера), избавляя от дополнительных ветвлений в алгоритме, как делалось в программах p1003 и p10004.

```
begin // p10006
  var q1 := new Queue<integer>(SeqRandom(5, 10, 99).Sorted);
  q1.Println;
  q1.Enqueue(integer.MaxValue); // маркер конца
  var q2 := new Queue<integer>(SeqRandom(8, 10, 99).Sorted);
  q2.Println;
  q2.Enqueue(integer.MaxValue); // маркер конца
  var q := new Queue<integer>;
  while (q1.Count <> 1) or (q2.Count <> 1) do
    q.Enqueue(q1.Peek < q2.Peek ? q1.Dequeue : q2.Dequeue);
  q.Println
end.
```

```
19 22 24 75 93
26 48 54 69 69 72 78 86
19 22 24 26 48 54 69 69 72 75 78 86 93
```

Можно ли решить данную задачу без очередей, используя массивы? Безусловно. И даже существенно короче, но только менее эффективно. Впрочем, при относительно небольших объемах данных это не замечается. Никаких процедур слияния не потребуется и на самом деле неважно, отсортированы ли исходные массивы либо последовательности: они просто объединяются, а затем полученная последовательность сортируется. Выбор, как всегда, за вами.

```
begin
  var a1 := ArrRandom(5, 10, 99);
  a1.Println;
  var a2 := ArrRandom(8, 10, 99);
  a2.Println;
  var a := (a1 + a2).Sorted.ToArray; // решение в одну строку
  a.Println
end.
```

Справедливости ради следует отметить, что предложено решение задачи, условие которой отличается от поставленного. Совпадает только конечная цель.

10.1.3 Линейные списки: заключение

Линейные списки хороши, когда не нужно задумываться о том, насколько эффективна их внутренняя реализация. За счет чего обеспечивается «безразмерность» стеков и очередей? Как получается, что их текущий размер в точности совпадает с количеством помещенных элементов? Эти вопросы выходят за рамки данной

книги. В PascalABC.NET стеки и очереди «встроены» в язык. Просто пользуйтесь ими.

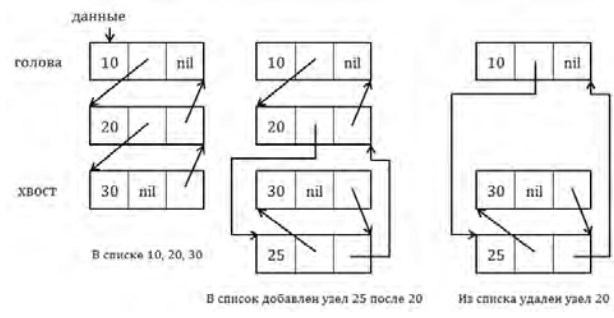
Что можно сказать о недостатках линейных списков? С элементами линейных списков можно работать только на концах списка. Если требуется выполнить поиск конкретного элемента, осуществить вставку, замену или добавление элемента где-то внутри списка, нужны другие информационные структуры.

10.2 Двусвязный список LinkedList

Связный список – это последовательность из нуля или более элементов (узлов), каждый из которых содержит собственно данные и одну или две ссылки (связки) на следующий и (или) предыдущий узел.

Связные списки являются динамической структурой, в которой память под элементы выделяется или утилизируется по мере необходимости. При добавлении нового узла в нужное место или удалении узла из списка производится лишь корректировка связанных с этим узлом ссылок. Отсутствие физического копирования данных позволяет эффективно делать массовую вставку и удаление узлов. Но не бывает достоинств без недостатков и за все нужно платить. Платой при работе со связными списками является невозможность быстрого доступа к конкретному узлу. Связный список – это структура с последовательным, а не произвольным доступом. Списки, как и очереди, имеют начало (голову) и конец (хвост).

Если в узле содержится одна ссылка (обычно, на следующий узел), список называют ***односвязным***. По односвязному списку можно двигаться только в одном направлении. Так, если в узле k определена ссылка на последующий узел k+1, то перейти к узлу, например, k-1 невозможно без просмотра списка от начала. В ***двусвязном*** списке одна ссылка указывает на следующий узел, а другая – на предшествующий. По такому списку можно двигаться в обоих направлениях, но только к непосредственно соседнему узлу.



На рисунке показан принцип организации и работы двусвязного списка. Вначале в списке имеются три узла со значениями 10, 20 и 30. Затем в список вставляется узел со значением 25 так, чтобы он располагался после узла со значением 20. Физически узлы никуда не перемещаются, корректируются лишь ссылки. Далее, узел со значением 20 удаляется, и это сводится к корректировке двух ссылок.

В PascalABC.NET ***двусвязный список*** (далее – список) реализован на основе стандартной коллекции System.Collections.Generic.LinkedList. Список является стандартным типом языка, поэтому его можно создать с использованием оператора присваивания вида

```
var имя := new LinkedList<тип_данных>;
```

В качестве типа данных, помещаемых в список, можно указать любой тип, в том числе, предварительно определенный пользователем. Поскольку коллекция реализована как контейнер, типом данных, помещаемых в список, может быть и список.

```
type
  r1 = record
    f1: integer;
    f2: real;
    f3: array of boolean;
    f4: string
  end;

begin
  var a := new LinkedList<boolean>;
  var b := new LinkedList<array[,] of real>;
  var c := new LinkedList<r1>;
end.
```

Конструктор списка дает возможность совместить создание списка с его заполнением элементами последовательности (или данных с типом, приводящимся к последовательности), указанной в качестве параметра.

```
begin
  var LL1 := new LinkedList<integer>(SeqRandom(4, 10, 99));
  LL1.Println; // 54 92 42 33
  var LL2 := new LinkedList<integer>(LL1.Sorted);
  LL2.Println // 33 42 54 92
end.
```

В приведенном примере список LL1 заполняется данными от генератора случайных чисел. Список LL2 заполняется данными из списка LL1, которые предварительно сортируются в порядке возрастания.

10.2.1 Операции для работы с двусвязным списком

Следует понимать, что при работе со списком мы оперируем не помещенными в него данными, а узлами LinkedListNode, содержащими эти данные, т.е. записями предопределенной структуры. LinkedListNode – это не номер узла по порядку и даже вообще не номер – это ссылка. Её можно получить, например, выполнив поиск по значению содержащихся в узле данных. Такие ссылки используются для указания узлов, с которыми (или относительно которых) совершаются операции.

В работе со списком LinkedList поддерживаются следующие средства:

- `.Count` – свойство, возвращающее количество узлов в списке;

- `.First` – свойство, возвращающее ссылку на первый узел;
- `.Last` – свойство, возвращающее ссылку на последний узел;
- `.AddAfter(узел, данное)` – метод, добавляющий указанное данное после заданного узлом. Узел должен иметь тип LinkedListNode;
- `.AddBefore(узел, данное)` – метод, добавляющий указанное данное перед заданным узлом. Узел должен иметь тип LinkedListNode;
- `.AddFirst(данное)` – метод, добавляющий указанное данное в начало (голову) списка. Возвращает добавленный узел типа LinkedListNode;
- `.AddLast(данное)` – метод, добавляющий указанное данное в конец (хвост) списка. Возвращает добавленный узел типа LinkedListNode;
- `.Clear` – метод, удаляющий из списка все узлы;
- `.Contains(значение)` – метод, возвращающий True, если в списке имеется узел, содержащий указанное значение и False в противном случае (проверка на наличие значения данных в списке);
- `.CopyTo(имя_массива, индекс)` – метод, копирующий содержимое списка в существующий одномерный динамический массив, начиная с элемента, указанного индексом. Если данные не помещаются в массиве, возникает исключение;
- `.Find(значение)` – метод, возвращающий первый из узлов LinkedListNode, в котором содержится указанное значение. Если такого значения нет, возвращается nil;
- `.FindLast(значение)` – метод, возвращающий последний из узлов LinkedListNode, в котором содержится указанное значение. Если такого значения нет, возвращается nil;
- `.Remove(значение)` – метод, удаляющий из списка первый из узлов LinkedListNode, в котором содержится указанное значение. Если такого значения нет, список не изменяется;
- `.Remove(узел)` – метод, удаляющий из списка указанный узел LinkedListNode;
- `.RemoveFirst` – метод, удаляющий из списка первый узел LinkedListNode;
- `.RemoveLast` – метод, удаляющий из списка последний узел LinkedListNode.

> К содержимому списка можно применять все методы для работы с последовательностями.

Для узла LinkedListNode чаще всего используются три следующих свойства, первые два их которых **доступны только для чтения**:

- `.Next` – возвращает следующий узел или nil, если узел последний;
- `.Previous` – возвращает предыдущий узел или nil, если узел первый;
- `.Value` – возвращает значение данных, хранящихся в узле. Это значение разрешается изменять.

Расширение .ToLinkedList, примененное к последовательности, массиву или к объекту стандартной коллекции, позволяет создать список LinkedList нужного типа и разместить в нем имеющиеся данные.

В PascalABC.NET включена так называемая ***короткая функция*** LLst, возвращающая список LinkedList. Она может использоваться в нескольких форматах:

- `LLst(последовательность)` – возвращает список, сформированный из данных, источником которых служит последовательность;
- `LLst&<тип>(список значений)` – возвращает список, сформированный из перечисленных данных с типом, автоприводимым к указанному;
- `LLst(список значений)` – возвращает список, сформированный из перечисленных данных единого типа.

10.2.2 Некоторые приемы работы со списком LinkedList

При работе с двусвязным списком LinkedList важно помнить, что в нем не существует понятия порядкового номера узла. Есть только первый узел, последний, текущий, предшествующий текущему и следующий за текущим. Некоторый узел внутри списка можно идентифицировать по значению хранящихся в нем данных. В разговорной речи часто вообще не оперируют понятием узла, подменяя его хранимыми в узле данными. Например, говорят «Удалить из списка сведения об Иванове П.С.». В этих условиях важное значение приобретает поиск узлов, содержащих требуемые данные. Но поиск – это совсем не то, ради чего задумывались связные списки; очень медленная процедура поиска является одним из наиболее слабых мест этой информационной структуры. Связные списки особенно эффективны там, где производится массовая вставка и/или удаление узлов внутри списка. На основе связного списка можно строить сложные информационные структуры, такие как деки (двусторонние очереди), деревья и графы. LinkedList является некоторой абстракцией над классическим двусвязным списком, поэтому в некоторых случаях бывает удобнее построить собственный связный список, используя запись, в полях которой определены ссылки.

Создадим двусвязный список в соответствии с примером, показанным на рисунке (см. 10.2). и проделаем с ним операции по вставке узла со значением 25 после узла со значением 20, а затем узел со значением 20 удалим:

```
begin
  var L := LLst(10, 20, 30);
  var node := L.Find(20);
  L.AddBefore(node, 25);
  L.Remove(20);
  L.Println;
end.
```

Все просто. Но всегда ли так будет? К сожалению, далеко не всегда. Стоит только допустить наличие в узлах одинаковых значений данных – и вся эта простота бесследно исчезает. Подвод узлов (LinkedListNode) в стандартной коллекции LinkedList базируется на поиске значений данных в этих узлах. Распознаются лишь два случая – первый узел с искомым значением и последний из таких узлов. А если узлов с одинаковыми значениями данных десятки и нужен шестой из них в направлении от начала списка? Тогда проще признать, что LinkedList для подобных нужд непригоден. Но если все же надо, проще разгрузить этот список в последова-

тельность или массив, сделать там нужную операцию, а потом создать список заново.

Можем ли мы пройти по узлам всего списка? Легко, потому что, как уже отмечалось, для списка пригодны практически все приемы работы с последовательностями.

```
foreach var a in L do
  a.Println;
```

Как-то это не вяжется с идеологией списков, где имеются ссылки на предыдущий и последующий узлы. Мы получаем доступ к значению данных в узле, но не к самому узлу. Хочется сложностей? Пожалуйста, тот же самый, но «идеологически выдержанный» вывод значений данных в узлах node:

```
node := L.First;
node.Value.Println;
while node.Next <> nil do
begin
  node := node.Next;
  node.Value.Println
end;
```

Для прохода по списку в обратном порядке приведенный фрагмент кода потребуется незначительно изменить:

```
node := L.Last;
node.Value.Println;
while node.Previous <> nil do
begin
  node := node.Previous;
  node.Value.Println
end;
```

Рассмотрим, как удалить k-й от начала списка узел (если, конечно, такой имеется).

```
begin // p10007
  var L := SeqRandom(15, 1, 9).ToLinkedList;
  var k := ReadInteger;
  L.Println;
  var node := L.First;
  var i := 1;
  while (node.Next <> nil) and (i <> k) do
  begin
    node := node.Next;
    i += 1;
  end;
  if i = k then
    L.Remove(node); // недокументированная возможность
  L.Println
end.
```

В приведенной программе использована недокументированная в Справке PascalABC.NET возможность удаления узла с указанием ссылки на него. В то же время,

стандартные коллекции Microsoft .NET Framework достаточно подробно документированы в Интернет. Например, методы .AddAfter и .AddBefore также могут использовать в качестве аргумента не значение данных в узле, а ссылку на узел.

Следующая программа удаляет из списка второй в направлении от начала узел, содержащий значение 4 (предполагается, что такой узел есть).

```
begin// p10008
  var L := LLst(2, 4, 7, 9, 4, 8, 7, 5, 4, 1);
  L.Println;
  var n1 := L.Find(4);
  repeat
    n1 := n1.Next
  until n1.Value = 4;
  L.Remove(n1);
  L.Println
end.
```

10.2.3 Пример работы со списком LinkedList

Пусть в некоторой организации требуется вести список сотрудников с данными об их детях. Рассмотрим, как это можно реализовать при помощи стандартной коллекции Linked List.

```
// p10009
type
  Дети = record
    ФИО: string;
    Пол: char // М/Ж
  end;
  Родители = record
    ФИО: string;
    Дети: LinkedList<Дети>
  end;

function СписокДетей(МассивДетей: array of (string, char)):
LinkedList<Дети>;
begin
  Result := new LinkedList<Дети>;
  var Ребенок: Дети;
  (Ребенок.ФИО, Ребенок.Пол) := (МассивДетей[0][0], МассивДетей[0][1]);
  var Узел := Result.AddFirst(Ребенок);
  for var i:=1 to МассивДетей.High do
  begin
    (Ребенок.ФИО, Ребенок.Пол) := (МассивДетей[i][0], МассивДетей[i][1]);
    Узел := Result.AddAfter(Узел, Ребенок)
  end;
end;
```

```
procedure Вывод(Родитель: Родители);
begin
  Write(Родитель.ФИО, ': ');
  foreach var Ребенок in Родитель.Дети do
    Write(Ребенок.ФИО, '(', Ребенок.Пол, ') ');
  Writeln;
end;

begin
  var Сотрудники := new LinkedList<Родители>;
  var Сотрудник: Родители;

  Сотрудник.ФИО := 'Иванова И.В.';
  Сотрудник.Дети :=
      СписокДетей(Arr(('Иванова Е.К.', 'Ж'), ('Иванов М.К.', 'М')));
  var УзелСотрудники := Сотрудники.AddFirst(Сотрудник);

  Сотрудник.ФИО := 'Петров С.Н.';
  Сотрудник.Дети :=
      СписокДетей(Arr(('Петров К.С.', 'М')));
  УзелСотрудники := Сотрудники.AddAfter(УзелСотрудники, Сотрудник);

  Сотрудник.ФИО := 'Николаева С.Е.';
  Сотрудник.Дети :=
      СписокДетей(Arr(('Николаев П.С.', 'М'), ('Казьмина Т.И.', 'Ж')));
  УзелСотрудники := Сотрудники.AddAfter(УзелСотрудники, Сотрудник);

  Сотрудник.ФИО := 'Курочкина В.Т.';
  Сотрудник.Дети :=
      СписокДетей(Arr(('Курочкина Е.П.', 'Ж')));
  УзелСотрудники := Сотрудники.AddAfter(УзелСотрудники, Сотрудник);

  foreach var Работник in Сотрудники do
    Вывод(Работник);

end.
```

В приведенной программе формируется список сотрудников, а затем он выводится в одном из возможных представлений.

```
Иванова И.В.: Иванова Е.К.(Ж) Иванов М.К.(М)
Петров С.Н.: Петров К.С.(М)
Николаева С.Е.: Николаев П.С.(М) Казьмина Т.И.(Ж)
Курочкина В.Т.: Курочкина Е.П.(Ж)
```

Вопрос о целесообразности ведения подобного списка на базе LinkedList вы решаете самостоятельно. В качестве задания придумайте разумную альтернативу.

10.2.4 Моделирование дека на основе LinkedList

Дек (Deque) – это линейная информационная структура, в которой включение и исключение данных может происходить на любом из ее концов. Проще всего пред-

ставить дек, как два стека, направленные навстречу друг другу. Первый стек служит началом дека, второй – его концом. Обычно дек достаточно эффективно моделируют при помощи пары стеков (если, конечно, сами стеки реализованы эффективно), но пусть это будет вашим домашним заданием (Т10.1).

Нам потребуются процедуры (или функции) для включения данных в дек, для исключения данных из дека и доступа к данным, находящимся в начале (Head) и конце (Tail) дека. Те же, что и у стека Push, Pop и Peek, но в двойном количестве. Еще потребуется функция, сообщающая, пуст ли дек.

```
// p10010
type
  tDeque = LinkedList<integer>;

var
  Deque: tDeque;

procedure PushHead(data: integer);
begin
  Deque.AddFirst(data);
end;

procedure PushTail(data: integer);
begin
  Deque.AddLast(data);
end;

function NotEmpty := Deque.Count > 0;

function PopHead: integer;
begin
  Assert(NotEmpty, 'Дек пуст');
  Result := Deque.First.Value;
  Deque.RemoveFirst
end;

function PopTail: integer;
begin
  Assert(NotEmpty, 'Дек пуст');
  Result := Deque.Last.Value;
  Deque.RemoveLast
end;

function PeekHead: integer;
begin
  Assert(NotEmpty, 'Дек пуст');
  Result := Deque.First.Value
end;
```

```
function PeekTail: integer;
begin
  Assert(NotEmpty, 'Дек пуст');
  Result := Deque.Last.Value
end;

begin
  Deque := new LinkedList<integer>;
  PushTail(1); // 1
  PushTail(2); // 1 2
  PushTail(3); // 1 2 3
  PushHead(4); // 4 1 2 3
  PushTail(5); // 4 1 2 3 5
  Writeln(PeekTail); // вывод 5
  Writeln(Deque); // вывод [4 1 2 3 5]
  Writeln(PopHead); // вывод 4, в деке 1 2 3 5
  Writeln(PopTail); // вывод 5, в деке 1 2 3
  Writeln(PopHead); // вывод 1, в деке 2 3
  Writeln(PopTail); // вывод 3, в деке 2
  Writeln(PopTail); // вывод 2, дек пуст
  Writeln(PopHead); // ошибка "Дек пуст"
end.
```

Конечно, в данном случае современным решением будет реализация собственного класса, но пока мы еще не познакомились с объектно-ориентированным программированием. Отметим также, что дек реализован для конкретного типа данных **integer**. О работе системной процедуры Assert будет сказано в 12.1.1.2. Пока отметим, что она используется для вывода диагностического сообщения об ошибке.

10.3 Список List

У всех рассмотренных ранее информационных структур имеются два общих недостатка: проблемы с реализацией быстрого поиска элементов и отсутствие произвольного доступа к элементу по его порядковому номеру. Эти недостатки обычно устраняют использованием массивов. Но почему бы не создать структуру, обладающую свойствами одновременно и динамического массива, и списка, хотя бы односвязного? Как раз такую структуру и реализует стандартная коллекция List.

В коллекции List доступ к элементу обеспечивается по его индексу (номеру в списке, начинающемуся от нуля). Элементы или их последовательность можно добавлять в конец списка или в его произвольное место. Также, можно удалять один или более элементов из произвольного места списка. Можно условно считать, что список List – это продвинутый динамический массив с возможностями удобного удаления и добавления элементов. Элементы списка хранятся в порядке их добавления. Имеется возможность поиска элемента, а если список упорядочен, то быстрого бинарного поиска. И, конечно, привычно доступны все приемы для работы со списком, как с последовательностью.

10.3.1 Создание списка List

В PascalABC.NET список List (далее – список) реализован на основе стандартной коллекции System.Collections.Generic.List. Список является стандартным типом языка, поэтому его можно создать с использованием оператора присваивания вида

```
var имя := new List<тип_данных>;
```

В качестве типа данных, помещаемых в список, можно указать любой тип, в том числе, предварительно определенный пользователем. Поскольку коллекция реализована как контейнер, типом данных, помещаемых в список, может быть и список.

```
type
  r1 = record
    f1: integer;
    f2: real;
    f3: array of boolean;
    f4: string
  end;

begin
  var a := new List<boolean>;
  var b := new List<array[,] of real>;
  var c := new List<r1>;
end.
```

Конструктор списка дает возможность совместить создание списка с его заполнением элементами последовательности (или данных с типом, приводящимся к последовательности), указанной в качестве параметра.

```
begin
  var L1 := new List<integer>(SeqRandom(4, 10, 99));
  L1.Println; // 54 92 42 33
  var L2 := new List<integer>(L1.Sorted);
  L2.Println // 33 42 54 92
end.
```

В приведенном примере список L1 заполняется данными от генератора случайных чисел. Список L2 заполняется данными из списка L1, которые предварительно сортируются в порядке возрастания.

Список можно заполнить непосредственно перечисленными данными, воспользовавшись «короткой функцией» Lst:

```
var L := Lst(25, -23, 47, 100, 0, 14);
```

Здесь тип данных списка будет автоматически выведен из типа перечисленных значений.

Список также можно создать при помощи расширения .ToList из данных, приводящихся к последовательности.

```
begin
  var L := SeqRandom(8, -999, 999).ToList;
  L.Println // -471 289 -499 611 -835 -568 212 -869
end.
```

Аргументом функции Lst может также быть объект любого типа, в том числе члены последовательности. Здесь начинающие могут допустить ошибки, связанные с тем, что в список (впрочем, как и в любой другой контейнер) можно положить как сложный объект, так и последовательность отдельных значений базового типа. Внимательно изучите приведенный ниже пример.

```
begin
  var L1 := Lst(Seq(14, 172, -5, 0, 39));
  L1.Println; // 14 172 -5 0 39
  var L2 := Lst(Arr(14, 172, -5, 0, 39));
  L2.Println; // 14, 172, -5, 0, 39
  var L3 := Lst(ArrRandom(5, -99, 99));
  L3.Println; // 38 -10 24 -70 -93
  var L4 := Lst(L3);
  L4.Println; // [38,-10,24,-70,-93] - в L4 один элемент - список
  L4.Count.Println; // 1
  var L5 := Lst('Приветик!');
  L5.Println(' '); // Приветик!
  var L6 := Lst('Приветик!'.ToCharArray);
  L6.Println(' '); // [П,р,и,в,е,т,и,к,!] - опять один элемент - массив
  L6.Count.Println; // 1
  var L7 := Lst('Приветик!'.Select(t -> t));
  L7.Println(' ') // П р и в е т и к !
end.
```

10.3.2 Операции для работы со списком List

В работе со списком поддерживаются следующие средства:

- `.Count` – свойство, возвращающее количество элементов в списке;
- `.Add(объект)` – метод, добавляющий объект к концу списка;
- `.AddRange(s)` – метод, добавляющий к концу списка последовательность s, состоящую из объектов;
- `.BinarySearch(значение)` – метод, возвращающий индекс элемента списка, значение которого равно указанному. Может вызываться в виде функции. Если поиск неуспешен, возвращается отрицательное значение. Список должен быть упорядочен по неубыванию элементов. Существуют другие форматы вызова этого метода, но они требуют предварительного создания объекта .NET-класса IComparer, о чем говорить несвоевременно;
- `.Clear` – метод, очищающий список;
- `.Contains(значение)` – метод, возвращающий True, если элемент с указанным значением присутствует в списке и False в противном случае (проверка на наличие элемента в списке);

- `.CopyTo(имя_массива)` – метод, копирующий содержимое списка в существующий одномерный динамический массив. Если данные не помещаются в массиве, возникает исключение;
- `.CopyTo(имя_массива, индекс_в_массиве)` – метод, копирующий содержимое списка в существующий одномерный динамический массив, начиная с элемента, указанного индексом. Если данные не помещаются в массиве, возникает исключение;
- `.CopyTo(is, имя_массива, it, k)` – метод, копирующий содержимое k элементов списка в существующий одномерный динамический массив, начиная с элемента it. Копирование производится из списка, начиная с элемента, имеющего индекс is. Если данные не помещаются в массиве, возникает исключение;
- `.ToArray` – метод, копирующий содержимое списка в одномерный динамический массив и возвращающий этот массив в качестве результата. Не требует предварительного описания массива;
- `.Find(T -> boolean)` – метод, возвращающий первое найденное значение элемента, удовлетворяющего условию. Если поиск неуспешен, возвращается ноль;
- `.FindAll(T -> boolean)` – метод, возвращающий последовательность всех найденных значений элементов, удовлетворяющих условию. Если поиск неуспешен, возвращается ноль;
- `.FindLast(T -> boolean)` – метод, возвращающий последнее найденное значение элемента, удовлетворяющего условию. Если поиск неуспешен, возвращается ноль;
- `.FindIndex(T -> boolean)` – метод, возвращающий индекс первого найденного элемента, удовлетворяющего условию. Если поиск неуспешен, возвращается отрицательное значение;
- `.FindLastIndex(T -> boolean)` – метод, возвращающий индекс последнего найденного элемента, удовлетворяющего условию. Если поиск неуспешен, возвращается отрицательное значение;
- `.GetRange(i, k)` – метод возвращающий список из k элементов, начиная с элемента, имеющего индекс i;
- `.IndexOf(значение)` – метод, возвращающий индекс первого найденного элемента с указанным значением. Если поиск неуспешен, возвращается отрицательное значение;
- `.LastIndexOf(значение)` – метод, возвращающий индекс последнего найденного элемента с указанным значением. Если поиск неуспешен, возвращается отрицательное значение;
- `.Insert(индекс, объект)` – метод, вставляющий в список объект в виде элемента с указанным индексом;
- `.InsertRange(индекс, последовательность_объектов)` – метод, вставляющий последовательность объектов в список так, что первый вставленный элемент имеет указанный индекс;
- `.Remove(значение)` – метод, удаляющий из списка первый встреченный элемент с указанным значением;
- `.RemoveAt(индекс)` – метод, удаляющий из списка элемент с указанным индексом;
- `.RemoveAll(T -> boolean)` – метод, удаляющий из списка элементы, значения которых удовлетворяют условию;

- `.RemoveRange(i, k)` – метод, удаляющий из списка k элементов, начиная с элемента, имеющего индекс i;
- `Reverse(список)` – метод, изменяющий порядок следования элементов списка на обратный;
- `Sort(список)` – метод, сортирующий элементы списка в порядке неубывания значений;
- `.TrueForAll(T -> boolean)` – метод, возвращающий True, если все значения всех элементов списка удовлетворяют заданному условию и False в противном случае;
- `.Print` – стандартный метод PascalABC.NET. Осуществляет вывод содержимого списка от его начала. В качестве параметра можно указать символ или строку - разделитель, по умолчанию используется пробел;
- `.Println` – то же самое; в конце вывода осуществляется переход к новой строке.

К содержимому списка можно применять все методы для работы с массивами и последовательностями. В частности, обращаться к элементам, указывая индекс и использовать срезы, которые также будут представлять собой списки.

Внимание, важно! Если обращение к элементу списка происходит по индексу, можно изменить значение только всего элемента. Например, если элемент содержит запись, попытка выполнить присваивание ее отдельному полю вызовет ошибку компиляции. В то же время для элемента массива такая операция корректна.

10.3.3 Примеры использования списков List

Списки можно использовать во всех задачах, рассмотренных ранее, где использовались последовательности и одномерные массивы. На основе списков особенно эффективно решаются задачи, связанные со вставкой и удалением элементов. Автор языка Python вообще отказался от массивов, используя вместо них списки List (которые многие школьники, да и некоторые учителя упорно продолжают называть массивами).

Пример 1. Определить, сколько раз в заданном тексте встречается каждый символ, не учитывая пробелов. Результат расположить в порядке невозрастания частоты, с которой встречаются символы.

Превратим текст в последовательность символов, приведенную к нижнему (для определенности) регистру. Будем читать полученную последовательность и если символ еще не встречался, установим в счетчике частоты повторений этого символа значение, равное единице, а если встречался, добавим единицу в счетчик. Для хранения символов и их счетчиков используем список, элементы которого будут записями, состоящими из двух полей. После просмотра всех символов удалим из списка элемент, соответствующий символу «пробел». Затем отсортируем список по неубыванию значений в поле счетчика и выведем результат.

```
// p10011
type
  D = record
    символ: char;
    счетчик: integer
  end;

begin
  var s := 'Язык Паскаль был разработан в 1970 г. Никлаусом Виртом как язык,' +
      ' обеспечивающий строгую типизацию и интуитивно понятный синтаксис.' +
      ' Он был назван в честь французского математика, физика и философа' +
      ' Блеза Паскаля.';
  var L := new List<D>;
  var buf: D; // буфер для обмена с элементами списка
  foreach var символ in s.ToLower do
  begin
    var индекс := L.FindIndex(t -> t.символ = символ);
    if индекс < 0 then
    begin
      (buf.символ, buf.счетчик) := (символ, 1);
      L.Add(buf)
    end
    else
    begin
      buf := L[индекс];
      buf.счетчик += 1;
      L[индекс] := buf
    end
  end;
  L.RemoveAll(t -> t.символ = ' ');
  L := L.OrderByDescending(t -> t.счетчик).ThenBy(t -> t.символ).ToList;
  L.Println
end.
```

Результат выполнения этой программы выглядит следующим образом:

```
(а,21) (и,18) (о,12) (к,11) (н,11) (с,11) (т,11) (з,8) (л,7) (в,6) (б,5) (е,5)
(п,5) (р,5) (ы,5) (м,4) (у,4) (ф,4) (я,4) (.,3) (г,3) (ю,3) (,,2) (й,2) (ц,2)
(ч,2) (ь,2) (0,1) (1,1) (7,1) (9,1) (щ,1)
```

Наиболее часто встретилась буква «а», наиболее редко – буква «щ» и цифры.

Отметьте, как выполнялась модификация поля счетчика в элементе списка – через буферную переменную. Вполне корректный в случае использования массива оператор L[индекс].счетчик += 1; для списка приводит к ошибке компиляции, о чем уже было сказано.

Пример 2. Получить методом простого перебора все делители натурального числа, не превышающего 10^9.

Мы не знаем заранее, какое количество делителей будет иметь заданное число n, поэтому применение списка можно счесть целесообразным. Занесем в него едини-

цу, а проверку делимости будем проводить от 2 до n/2. В конце добавим в список само число n.

```
begin // p10012
  var L := Lst(1);
  var n := ReadInteger('Введите натуральное число');
  for var i := 2 to n div 2 do
    if n mod i = 0 then L.Add(i);
  if n <> 1 then L.Add(n);
  L.Println
end.
```

Пример работы программы:

```
Введите натуральное число 19528423
1 41 67 2747 7109 291469 476303 19528423
```

Пример 3. Создать на основе списка хранилище значений элементов непрямоугольного двумерного массива целых чисел. В первой строке пять элементов, во второй – три, в третьей – 8. В качестве значений элементов взять случайные числа.

```
begin // p10013
  var L := new List<array of integer>;
  var a := ArrRandom(5, -99, 99);
  L.Add(a);
  a := ArrRandom(3, -99, 99);
  L.Add(a);
  a := ArrRandom(8, -99, 99);;
  L.Add(a);
  L.Printlines;
end.
```

Пример вывода:

```
[26,-84,-80,65,-80]
[72,-59,-46]
[-35,-38,-56,10,-86,73,41,75]
```

10.4 Множество HashSet

Стандартные для базового Паскаля множества Set of T рассматривались в части 5 (см. 5.2). Полезно вспомнить, что множество конечно, не упорядочено и содержит элементы одного типа с уникальным значением. Ввиду сильно ограниченных возможностей, этот тип множеств оставлен в языке в целях совместимости.

Современные средства для организации множеств поддерживают ряд дополнительных операций, в частности, удаление всех элементов, удовлетворяющих условию, копирование элементов множества в массив, объединение множеств, нахождения их пересечения, разности (в том числе, симметрической) и т.д. Тип элементов множества при этом может быть произвольным (но, конечно же, единым), а на их количество накладываются очень слабые по сравнению с множествами Set of T ограничения.

Множество HashSet реализовано на базе ***хэш-таблицы***. Хэш-таблицы достаточно широко используется в программировании и заслуживают того, чтобы рассмотреть их несколько подробнее.

10.4.1 Понятие о хэш-таблицах

Хэш-таблица – это обычный массив, предполагающий необычный способ адресации элементов, основанный на вычислении индекса как некоторой ***хэш-функции*** от ключа – части значения хранимого в элементе данного. В некоторых случаях в роли ключа может выступать и все хранимое в элементе значение. Назначение хэш-таблицы – обеспечение очень быстрого доступа к элементу данных по значению ключа. В английском языке слово hash (хэш) означает путаницу, мешанину. Назначение хэш-функции состоит в том, чтобы ключ данных преобразовать в индекс массива. Хэш-функция сужает множество значений указанного типа до подмножества целых чисел. Хорошая хэш-функция должна работать быстро и вырабатывать индексы, с приемлемой степенью равномерности распределенные по массиву. Нужно понимать, что с помощью хэш-функции мы распределяем ограниченное пространство машинной памяти под зачастую практически неограниченное количество вариантов значений ключей. Например, ключом может быть фамилия, имя и отчество человека, соединенные с его датой и местом рождения. Длина такого ключа составляет не один десяток символов. А нам требуется сохранить данные, скажем, о тысяче человек.

Существует теория хэш-функций, говорящая о том, как их правильно создавать под конкретные потребности, но здесь для нее не место. Вместо теории рассмотрим иллюстративный пример.

Пусть нам требуется хранить в массиве не более восьми произвольных строк, а затем уметь находить элемент с нужной строкой без перебора элементов массива. В этом случае отличной структурой данных будет хэш-таблица на основе массива строк. В качестве ключа используем значение строки, а вот какую взять хэш-функцию? Начнем с конца. Если размер массива 8 элементов, то его индексы могут иметь значение от нуля до семи. Возникает идея вычисления какой-то уникальной «контрольной суммы» строки, из которой индекс получится путем взятия остатка от деления на 8. Не вдаваясь в подробности, для получения такой суммы используем простой алгоритм. Полагаем сумму нулевой, а затем для каждого символа строки берем его код в Unicode, и складываем с накопленной ранее суммой, предварительно умножив ее на 31, получая новую сумму. Обработав все символы строки, вычисляем искомое значение хэш-функции как остаток от деления суммы на 8.

Гарантирует ли такой алгоритм получение уникального кода для произвольной строки? Ну конечно же, нет! Если возможно хранение лишь восьми строк из k существующих, то в k/8 случаях хэш-функция будет порождать одинаковые значения. В подобных случаях говорят, что имеет место ***коллизия хэширования***. Но об этом позднее.

```
// p10014
function MyHash(s: string): integer;
begin
  var (size, mult) := (8, 31);
  var sum: longword := 0;
  foreach var c in s do
    sum := sum * mult + Ord(c);
  Result := sum mod size;
end;

begin
  var s := Arr(
      'Иванов Иван Иванович',
      'PascalABC.NET',
      'Сегодня хорошая погода',
      '6 июня 2019 года, четверг',
      'Изучаем hash-таблицы',
      'Где же коллизия?',
      'Может, здесь?',
      'Или здесь?');
  var HashTable := new string[8];
  foreach var s1 in s do
    HashTable[MyHash(s1)] := s1;
  HashTable.PrintLines
end.
```

Посмотрим, какие именно строки и в каких элементах массива будут размещены:

```
Где же коллизия?
Может, здесь?
Изучаем hash-таблицы
6 июня 2019 года, четверг

PascalABC.NET
Иванов Иван Иванович
Или здесь?
```

Из восьми предложенных наугад строк в массиве разместились семь. Один элемент массива остался свободен, и одна строка пропала – проявила себя коллизия. Фразе «Сегодня хорошая погода» не повезло: на ее место попала другая фраза. Вы можете поэкспериментировать с этой программой, придумывая свои строки и наблюдая возникновение коллизий.

Следующая программа имеет некоторую практическую направленность. Она демонстрирует, как с помощью хэш-таблицы можно хранить записи и производить быстрый поиск по ключу. Хэш-функция для простоты оставлена той же самой.

```
// p10015
type
  MyStr = record
    f1: string; // Фамилия, И.О.
    f2: string; // дата рождения дд.мм.гггг
    f3: integer // оклад
  end;

function MyHash(s: string): integer;
begin
  var (size, mult) := (8, 31);
  var sum: longword := 0;
  foreach var c in s do
    sum := sum * mult + Ord(c);
  Result := sum mod size;
end;

procedure ToHash(fio, dr: string; oklad: integer; t: array of MyStr);
begin
  var h := MyHash(fio);
  t[h].f1 := fio;
  t[h].f2 := dr;
  t[h].f3 := oklad
end;

begin
  var HashTable := new MyStr[8];
  ToHash('Иванов И.И.', '12.07.1970', 62000, HashTable);
  ToHash('Петров П.П.', '03.11.1982', 36000, HashTable);
  ToHash('Сидорова С.С.', '21.04.1974', 57300, HashTable);
  ToHash('Козлов К.К.', '09.03.1985', 28500, HashTable);
  ToHash('Тарасова Т.Т.', '11.11.1992', 24000, HashTable);
  HashTable.PrintLines;
  // пример извлечения данных
  var buf := HashTable[MyHash('Сидорова С.С.')];
  Writeln(buf.f1, ', дата рождения ', buf.f2, ', оклад ', buf.f3)
end.
```

На выводе видно, что все данные нашли себе место, коллизий не возникло. Поиск работника по ключу – фамилии и инициалам – оказался успешным.

```
(,,0)
(Иванов И.И.,12.07.1970,62000)
(,,0)
(,,0)
(Козлов К.К.,09.03.1985,28500)
(Сидорова С.С.,21.04.1974,57300)
(Петров П.П.,03.11.1982,36000)
(Тарасова Т.Т.,11.11.1992,24000)
Сидорова С.С., дата рождения 21.04.1974, оклад 57300
```

Напоследок несколько слов о коллизиях хэширования. Для их разрешения существует два метода. Первый основан на том, что в элемент хэш-таблицы помещают-

ся не данные, а их список. Если возникает коллизия, данное добавляется к списку. При поиске нужно сравнивать ключ с хранимыми в списке значениями, фактически проводя дополнительный поиск по списку. Второй метод в случае коллизии предписывает размещать данное в ближайшем свободном элементе хэш таблицы. Оба метода имеют свои достоинства и недостатки.

Но вам повезло – вам доступно множество HashSet. И не нужно думать о том, что именно сделать ключом.

10.4.2 Создание множества HashSet

В PascalABC.NET множество HashSet (далее – множество) реализовано на основе стандартной коллекции System.Collections.Generic.HashSet. Множество является стандартным типом языка, поэтому его можно создать с использованием оператора присваивания вида

```
var имя := new HashSet<тип_данных>;
```

В качестве типа данных, помещаемых во множество, можно указать любой тип, в том числе, предварительно определенный пользователем. Поскольку коллекция реализована как контейнер, типом данных, помещаемых во множество, может быть и множество.

```
type
  r1 = record
    f1: integer;
    f2: real;
    f3: array of boolean;
    f4: string
  end;

begin
  var a := new HashSet<boolean>;
  var b := new HashSet<array[,] of real>;
  var c := new HashSet<r1>;
end.
```

Конструктор множества дает возможность совместить создание множества с его заполнением элементами последовательности (или данных с типом, приводящимся к последовательности), указанной в качестве параметра.

```
begin
  var h1 := new HashSet<integer>(SeqRandom(4, 10, 99));
  h1.Println; // 54 92 42 33
  var h2 := new HashSet<integer>(h1.Sorted);
  h2.Println // 33 42 54 92
end.
```

В приведенном примере множество h1 заполняется данными от генератора случайных чисел. Множество h2 заполняется данными из множества h1, которые предварительно сортируются в порядке возрастания.

Множество можно заполнить непосредственно перечисленными данными, воспользовавшись «короткой функцией» HSet:

```
var h := HSet(25, -23, 47, 100, 0, 14);
```

Здесь тип данных множества будет автоматически выведен из типа перечисленных значений.

Множество также можно создать при помощи расширения .ToHashSet из данных, приводящихся к последовательности.

```
begin
  var h := SeqRandom(8, -999, 999).ToHashSet;
  h.Println // -471 289 -499 611 -835 -568 212 -869
end.
```

Аргументом функции HSet может также быть объект любого типа, в том числе, члены последовательности. Здесь начинающие могут допустить ошибки, связанные с тем, что в множество можно поместить как сложный объект, так и последовательность отдельных значений базового типа. Внимательно изучите приведенный ниже пример.

```
begin
  var h1 := HSet(Seq(14, 172, -5, 0, 39));
  h1.Println; // 14 172 -5 0 39
  var h2 := HSet(Arr(14, 172, -5, 0, 39));
  h2.Println; // 14, 172, -5, 0, 39
  var h3 := HSet(ArrRandom(5, -99, 99));
  h3.Println; // 38 -10 24 -70 -93
  var h4 := HSet(h3);
  h4.Println; // (38,-10,24,-70,-93) - в h4 один элемент – множество
  h4.Count.Println; // 1
  var h5 := HSet('Приветик!');
  h5.Println(' '); // Приветик!
  var h6 := HSet('Приветик!'.ToCharArray);
  h6.Println(' '); // [П,р,и,в,е,т,и,к,!] - опять один элемент - массив
  h6.Count.Println; // 1
  var h7 := HSet('Приветик!'.Select(t -> t));
  h7.Println(' ') // П р и в е т и к !
end.
```

Множества HashSet и Set of T являются совместимыми по присваиванию.

```
begin
  var s1 := [5, 8, 11, 4, -2, 0, 7];
  s1.Println; // -2 11 8 7 5 4 0
  var s2: HashSet<integer>;
  s2 := s1;
  s2.Println; // -2 11 8 7 5 4 0
  var s3: set of integer;
  s3 := s2;
  s3.Println // -2 11 8 7 5 4 0
end.
```

Вот так можно реализовать программу p10014 на основе множества HashSet:

```
begin
  var s := Arr(
      'Иванов Иван Иванович',
      'PascalABC.NET',
      'Сегодня хорошая погода',
      '6 июня 2019 года, четверг',
      'Изучаем hash-таблицы',
      'Где же коллизия?',
      'Может, здесь?',
      'Или здесь?');
  var HashTable := s.ToHashSet;
  HashTable.PrintLines
end.
```

Вывод результатов показывает, что коллизии тут не возникает.

```
Иванов Иван Иванович
PascalABC.NET
Сегодня хорошая погода
6 июня 2019 года, четверг
Изучаем hash-таблицы
Где же коллизия?
Может, здесь?
Или здесь?
```

10.4.3 Операции для работы с множеством HashSet

В работе с множеством поддерживаются следующие средства:

- `.Count` – свойство, возвращающее количество элементов в множестве;
- `.Add(объект)` – метод, добавляющий во множество указанный объект;
- `.Clear` – метод, удаляющий из множества все элементы;
- `.Contains(значение)` – метод, возвращающий True, если элемент с указанным значением присутствует во множестве и False в противном случае (проверка на наличие элемента во множестве);
- `.CopyTo(имя_массива, индекс)` – метод, копирующий элементы множества в существующий одномерный динамический массив, начиная с элемента, указанного индексом. Если данные не помещаются в массиве, возникает исключение;
- `.ExceptWith(последовательность)` – метод, исключающий из множества элементы, содержащиеся в последовательности;
- `.IntersectWith(последовательность)` – метод, исключающий из множества элементы, отсутствующие в последовательности (операция пересечения);
- `.IsProperSubsetOf(последовательность)` – метод, возвращающий True, если множество является строгим подмножеством последовательности (все элементы множества присутствуют в последовательности и при этом последовательность содержит дополнительные элементы – операция «строгое вложение») и False в противном случае;
- `.IsProperSupersetOf(последовательность)` – метод, возвращающий True, если множество является строгим надмножеством над последовательностью (все

элементы последовательности присутствуют во множестве и при этом множество содержит дополнительные элементы – операция «строго содержит») и False в противном случае;

- `.IsSubsetOf(последовательность)` – метод, возвращающий True, если множество является подмножеством последовательности (все элементы множества присутствуют в последовательности – операция «нестрогое вложение») и False в противном случае;
- `.IsSupersetOf(последовательность)` – метод, возвращающий True, если множество является надмножеством над последовательностью (все элементы последовательности присутствуют во множестве – операция «нестрогое вложение») и False в противном случае;
- `.Overlaps(последовательность)` – метод, возвращающий True, если во множестве и в последовательности есть хотя бы один совпадающий элемент и False в противном случае;
- `.Remove(значение)` – метод, удаляющий из множества элемент с указанным значением. Если такого элемента нет, никаких действий не выполняется;
- `.RemoveWhere(T -> boolean)` – метод, удаляющий из множества все элементы типа T, для которых истинно логическое лямбда-выражение. Если таких элементов нет, никаких действий не выполняется;
- `.SetEquals(последовательность)` – метод, возвращающий True, если все элементы во множестве и в последовательности совпадают (имеет место эквивалентность) и False в противном случае;
- `.SymmetricExceptWith(последовательность)` – метод, исключающий из множества элементы, одновременно присутствующие в этом множестве и в последовательности;
- `.UnionWith(последовательность)` – метод, добавляющий во множество элементы последовательности, которые отсутствовали в нем;
- `.Print` – стандартный метод PascalABC.NET. Осуществляет вывод содержимого множества. В качестве параметра можно указать символ или строку - разделитель, по умолчанию используется пробел;
- `.Println` – то же самое; в конце вывода осуществляется переход к новой строке.

К множествам HashSet можно применять все методы для работы с массивами и последовательностями. В приведенном выше перечне методов на месте параметра «последовательность» может быть использован массив или любая коллекция, приводящаяся к последовательности.

К множествам HashSet могут быть применены все операции, которые перечислены для множеств Set of T (см. 5.2.3).

10.4.4 Примеры работы со множествами HashSet

Множества удобно использовать в случаях, когда требуется получать и хранить наборы данных с уникальными значениями. Другое назначение множеств – выполнение над ними теоретико-множественных операций, таких как пересечение, объединение, разность и т.д.

Пример 1. Получить массив, содержащий 20 различных случайных натуральных двухзначных чисел.

Можно воспользоваться множеством, помещая в него случайные числа до тех пор, пока в нем не окажется 20 элементов. Затем преобразовать это множество в массив и перемешать случайным образом его элементы.

```
begin // p10016
  var s := new HashSet<integer>;
  while s.Count < 20 do
    s += Random(10, 99);
  var a := s.ToArray;
  a.Shuffle;
  a.Println
end.
```

Эту идею можно использовать, например, моделируя перетасованную колоду карт, соответствующим образом пронумеровав карты.

Пример 2. Даны три слова. Верно ли, что для их записи использован один и тот же набор букв?

Составим множества из букв каждого слова. Достаточно убедиться, что два из множеств являются нестрогим включением в третье, имеющее мощность не меньше, чем остальные.

```
begin // p10017
  var a := Arr('апельсин', 'лепнина', 'спаниель');
  var L := new List<HashSet<char>>;
  foreach var w in a do
    L.Add(w.ToHashSet);
  var imax := L.Select(t -> t.Count).ToArray.IndexMax;
  var Smax := L[imax];
  L.RemoveAt(imax);
  var good := True;
  foreach var s in L do
    if not (s <= Smax) then
    begin
      good := False;
      break
    end;
  if good then Println('Верно')
  else Println('Неверно')
end.
```

Решение не ограничено тремя словами, их можно указать любое количество. Строится список, элементами которого являются множества HashSet, содержащие набор символов, из которых составлены соответствующие слова. Определяется индекс элемента с множеством, содержащим максимальное количество символов. Этот элемент помещается в множество Smax, а затем удаляется из списка. Остается лишь проверить отношения между оставшимися в списке множествами и Smax.

10.5 Словарь Dictionary

Словарь – это структура данных, в которой единицей хранения является пара «ключ – значение». Словарь позволяет быстро находить искомое значение, указывая его ключ. Заданный ключ с помощью хэш-функции преобразуется в индекс элемента хранения данных (это называется ***ассоциированием***). Словари иначе называют ассоциированными массивами на базе хэш-таблицы. Добавляемые в словарь данные должны иметь уникальный ключ.

В стандартной коллекции Microsoft .NET имеется структура System.Collections.Generic.KeyValuePair, на основе которой строятся элементы словаря.

10.5.1 Структура KeyValuePair<ключ, значение>

Структура определяет пару «ключ – значение», которую можно задавать или получать. Типы данных ключа TKey и значения TValue могут быть произвольными. Для создания объекта KeyValuePair<TKey, TValue> используется конструктор. Поддерживаются свойства .Key и .Value, доступные только для чтения.

Объекты класса KeyValuePair можно описывать и создавать следующим образом:

```
begin
  var kv1: KeyValuePair<integer, string>;
  var kv2: KeyValuePair<string, List<integer>>;
  var kv3 :=
     new KeyValuePair<integer, array of integer>(28, Arr(1, 8, 3, 2));
  Writeln(kv3); // (28,[1,8,3,2])
  var kv4 := new KeyValuePair<string, real>('j12k29', 1.39e-12);
  Writeln(kv4); // (j12k29,1.39E-12)
end.
```

Рассмотрим примеры работы со свойствами .Key и .Value, предоставляющие доступ к ключу и значению соответственно.

```
begin
  var kv1 :=
     new KeyValuePair<integer, array of integer>(28, Arr(1, 3, 4, 2));
  Print('kv1 с ключом', kv1.Key, 'хранит массив значений');
  kv1.Value.Print;
  Println(' с суммой, равной', kv1.Value.Sum)
  // kv1 с ключом 28 хранит массив значений 1 3 4 2 с суммой, равной 10
end.
```

Создать и/или инициализировать структуру KeyValuePair можно при помощи «короткой функции» KV(ключ, значение).

```
begin
  var kv1 := KV(-18, 'Тест');
  Println(kv1); // (-18,Тест)
  var kv2: KeyValuePair<char, List<integer>>;
  kv2 := KV('*', Lst(1, 3, 5, 7, 2, 4, 6, 0));
  Println(kv2) // (*,[1,3,5,7,2,4,6,0])
end.
```

10.5.2 Создание словаря

В PascalABC.NET словарь Dictionary<Tkey, TValue> (далее – словарь) реализован на основе стандартной коллекции System.Collections.Generic.Dictionary. Словарь создается, как коллекция объектов класса KeyValuePair<TKey, TValue> и является стандартным типом языка, поэтому его можно создать с использованием оператора присваивания вида

```
var имя := new Dictionary<тип_ключа, тип_значения>;
```

Типы ключа и значения могут быть произвольными, в том числе предварительно определенными пользователем. При этом надо понимать, что значения ключа хэшируются и не проявлять особого фанатизма в выборе экстравагантного типа для ключей. Поскольку коллекция реализована как контейнер, типом данных, помещаемых в словарь, может быть и словарь.

```
type
  r1 = record
    f1: integer;
    f2: real;
    f3: array of boolean;
    f4: string
  end;

begin
  var a := new Dictionary<integer, boolean>;
  var b := new Dictionary<integer, array[,] of real>;
  var c := new Dictionary<string, r1>;
end.
```

Конструктор словаря не дает возможности совместить создание словаря с его заполнением данными, но для этой цели можно использовать «короткую функцию» Dict. В качестве источника данных для заполнения словаря можно задавать последовательность пар «ключ – значение» (в виде массива кортежей) или коллекцию объектов KeyValuePair.

```
begin
  var d1 := Dict(Arr(('a1', 142), ('b2', 216), ('d4', 18)));
  d1.Println; // (a1,142) (b2,216) (d4,18)
  var d2 := new Dictionary<integer, char>;
  var kvp := new KeyValuePair<integer, char>[4];
  for var i := 0 to kvp.High do
    kvp[i] := KV(i + 1, Chr(Ord('a') + i));
  d2 := Dict(kvp);
  d2.Println // (1,a) (2,b) (3,c) (4,d)
end.
```

Словарь также можно создать при помощи расширения .ToDictionary из данных, приводящихся к последовательности. При этом необходимо указать хотя бы один параметр в виде лямбда-функции, задающей преобразование элементов последовательности в уникальный ключ. Второй, необязательный параметр, также задается в виде «лямбды» и определяет преобразование элементов последовательности в хранимые значения.

```
begin
  var a := ArrRandom(4, -99, 99);
  var d := a.ToDictionary(t -> t, t -> Sign(t) * Abs(t) ** (1 / 3));
  d.Println
end.
```

В данном случае ключом словаря будет случайное целое число в диапазоне от -99 до 99, а значением – кубический корень из этого числа:

```
(60,3.91486764116886)
(96,4.57885697021333)
(-67,-4.06154810044568)
(5,1.7099759466767)
```

Отметьте, что элементы словаря Dictionary хранятся неупорядоченными относительно значений своих ключей.

> Обращение к элементу словаря в формате D[k] подразумевает обращение к данным, хранящимся в элементе, имеющем ключ k. Эти данные доступны для изменения. Если указанный ключ не найден, во время исполнения программы возникает исключение, приводящее к выводу сообщения «Ошибка времени выполнения: Данный ключ отсутствует в словаре». Сам ключ элемента менять недопустимо.

10.5.3 Операции для работы со словарем

В работе со словарем поддерживаются следующие средства:

- `.Count` – свойство, возвращающее количество элементов в словаре;
- `.Keys` – свойство, возвращающее последовательность ключей, имеющихся в словаре;
- `.Values` – свойство, возвращающее последовательность значений, имеющихся в словаре;

- `.Add(ключ, значение)` – метод, добавляющий в словарь элемент с указанными ключом и значением. При попытке добавления элемента с ключом, уже имеющимся в словаре, возникает исключение;
- `.Clear` – метод, удаляющий из словаря все элементы;
- `.ContainsKey(ключ)` – метод, возвращающий True, если в словаре имеется элемент с указанным ключом и False в противном случае (проверка на наличие ключа в словаре);
- `.ContainsValue(значение)` – метод, возвращающий True, если в словаре имеется элемент с указанным значением и False в противном случае (проверка на наличие значения в словаре);
- `.Get(ключ)` – расширение PascalABC.NET, возвращающее значение элемента, связанного с указанным ключом или значение по умолчанию, если ключ не найден;
- `.Remove(ключ)` – метод, удаляющий из словаря элемент с указанным ключом. Если такого ключа нет, ничего не происходит;
- `.TryGetValue(ключ, имя_переменной)` – метод, выполняющий поиск элемента словаря по указанному ключу. Если поиск успешен, переменная заполняется значением из найденного элемента и возвращается True. В противном случае возвращается False;
- `.Print` – стандартный метод PascalABC.NET. Осуществляет вывод содержимого множества. В качестве параметра можно указать символ или строку - разделитель, по умолчанию используется пробел;
- `.Println` – то же самое; в конце вывода осуществляется переход к новой строке.

К словарям Dictionary можно применять все методы для работы с массивами и последовательностями, но при этом следует помнить, что каждый элемент такой последовательности имеет тип KeyValuePair и в «лямбдах» указывать свойства по типу t -> t.Key, t -> t.Value.

10.5.4 Примеры работы со словарем

Пример 1. Определить, сколько раз в заданном тексте встречается каждый символ, не учитывая пробелов. Результат расположить в порядке невозрастания частоты, с которой встречаются символы. Эта задача уже решалась (см. 10.3.3, программа p10011), но словарь позволяет сделать решение более коротким и понятным.

```
begin // p10018
  var s := 'Язык Паскаль был разработан в 1970 г. Никлаусом Виртом как язык,' +
       ' обеспечивающий строгую типизацию и интуитивно понятный синтаксис.' +
       ' Он был назван в честь французского математика, физика и философа' +
       ' Блеза Паскаля.';
  s.ToLower.Where(c -> c <> ' ').GroupBy(c -> c)
       .ToDictionary(t -> t.Key, t -> t.Count).OrderByDescending(t -> t.Value)
       .ThenBy(t -> t.Key).Println;
end.
```

Все решение уложилось в два исполняемых оператора PascalABC.NET. Вот что значит удачно выбранный для решения инструментарий. Привели строку к ниж-

нему регистру, отфильтровали непробельные символы, сгруппировали их по значению и создали словарь, элементы которого затем отсортировали и вывели.

Пример 2. Написать программу, шифрующую текст на кириллице при помощи азбуки Морзе.

```
begin // p10019
  var a: array of (char, string) := (('а', '.-'), ('б', '-...'),
      ('в', '.--'), ('г', '--.'), ('д', '-..'), ('е', string('.')),
      ('ж', '...-'), ('з', '--..'), ('и', '..'), ('й', '.---'),
      ('к', '-.-'), ('л', '.-..'), ('м', '--'), ('н', '-.'),
      ('о', '---'), ('п', '.--.'), ('р', '.-.'), ('с', '...'),
      ('т', string('-')), ('у', '..-'), ('ф', '..-.'),
      ('х', '....'), ('ц', '-.-.'), ('ч', '---.'), ('ш', '....'),
      ('щ', '--.-'), ('ъ', '-..-'), ('ы', '-.--'), ('ь', '-..-'),
      ('э', '..-..'), ('ю', '..--'), ('я', '.-.-'), (' ', '----'));
  var d := a.ToDictionary(t -> t[0], t -> t[1]);
  var s := 'тестовое сообщение';
  var r: string := '';
  var m: string;
  foreach var c in s do
    if d.TryGetValue(c, m) then r := r + m + ' ';
  r.Println;
end.
```

Результат работы программы:

```
- . ... - --- .-- --- . -...- ... --- --- -... --.- . -. .. .
```

10.6 Коллекции с упорядоченностью элементов

До сих пор рассматривались стандартные коллекции, элементы которых хранились или в порядке, зависящем от порядка добавления в коллекцию, или в произвольном порядке, обусловленном внутренними алгоритмами размещения элементов. В некоторых случаях требуется, чтобы объекты в коллекции всегда были упорядочены в соответствии с определенным правилом, например, по возрастанию значений (или по неубыванию, если коллекция допускает повторения).

Правило, в соответствии с которым упорядочиваются объекты коллекции, задается объектом, который называется ***компаратором***. Каждая коллекция по умолчанию содержит компаратор, поддерживающий упорядоченность по возрастанию (или неубыванию) ее элементов. Если элемент коллекции содержит ключ, компаратор оперирует значением ключа, а если ключа нет, значением хранимых данных. В случае, когда хранимые данные имеют собственную структуру, обычно выбирается ее первый элемент. При создании коллекции можно указать собственный компаратор с тем, чтобы обеспечить требуемую упорядоченность. С этой целью создается класс, описывающий элемент хранения, в котором описывается метод сравнения пары элементов, например, текущего с другим. Такой класс должен поддерживать так называемый ***интерфейс*** IComparble, а метод - носить имя CompareTo. Другой вариант компаратора сравнивает пару указанных элементов. В этом

случае класс должен поддерживать интерфейс IComparer, а метод – называться Compare. Подробно об интерфейсах говорится в части 13 (см. 13.9).

Из обобщенных стандартных коллекций Microsoft .NET в PascalABC.NET напрямую поддерживаются три: SortedSet, SortedList и SortedDictionary. От своих «неупорядоченных родственников» они отличаются лишь несколькими дополнительными свойствами и методами, которые и будут рассмотрены ниже.

10.6.1 Упорядоченное множество SortedSet

Реализовано на основе коллекции System.Collections.Generic.SortedSet и создается как коллекция объектов класса SortedSet<T>. Коллекция построена на базе бинарного дерева поиска, что позволяет осуществлять очень быстрый поиск ее элементов. Схожа с коллекцией HashSet, но хранит элементы, поддерживая их упорядоченность, в том числе, при добавлении.

Коллекцию можно создать с использованием оператора присваивания вида

```
var имя := new SortedSet<тип_данных>;
```

Имеются также варианты конструктора, в которых указывается компаратор.

Создание множества SortedSet можно совместить с его заполнением данными из последовательности, как это было описано для HashSet.

```
begin
  var ss := new SortedSet<char>(Seq('в', 'с', 'а', 'р', 'с', 'е'));
  ss.Println // аверс
end.
```

В приведенном примере продублированный символ «с» не вошел в созданное множество, а при выводе наблюдается наличие упорядоченности по алфавиту.

Создать множество на основе перечисляемых или приводящихся к последовательности значений позволяет «короткая функция» SSet:

```
begin
  var ss := SSet(5, 7, 2, 9, 11, 3, 5, 4);
  ss.Println // 2 3 4 5 7 9 11
  ss.UnionWith(Seq(9, 1, 5));
  ss.Println; // 1 2 3 4 5 7 9 11
end.
```

И в этом случае дубликаты значений в множество не попадают, а его элементы сохраняют упорядоченность.

Множество SortedSet также можно создать при помощи расширения .ToSortedSet из данных, приводящихся к последовательности.

```
begin
  var ss := SeqRandom(7, 0, 9).ToSortedSet;
  ss.Println // 0 1 6 8
end.
```

В данном случае было создано упорядоченное множество из четырех элементов. Конечно же, случайная последовательность содержала семь чисел, но три из них оказались дубликатами.

В отличии от HashSet, множество SortedSet не является совместимым по присваиванию со множеством Set of T. Это означает, в частности, что множества типа Set of T не могут появляться в одном выражении с множествами SortedSet. Например, если s1 множество SortedSet, а s2 – множество Set of T, ошибочно писать s1+=[3] или s1.IntersectWith(s2).

10.6.1.1 И снова компараторы, снова интерфейс

Рассмотрим простейший пример использования компаратора. Если что-то в оформлении класса будет непонятно – до изучения объектно – ориентированного программирования пропустите это место и считайте написанное образцом.

```
type
  MyComparer = class(IComparer<integer>) // для элементов типа integer
  public
    function Compare(a, b: integer) := b.CompareTo(a);
    // a.CompareTo(b) - по возрастанию, b.CompareTo(a) - по убыванию
end;

begin
  var a := ArrRandom;
  a.Println; // 4 36 35 3 53 65 86 42 30 6 – случайный порядок
  var s1 := new SortedSet<integer>(a);
  s1.Println; // 3 4 6 30 35 36 42 53 65 86 – порядок по умолчанию
  s1 := new SortedSet<integer>(a, new MyComparer);
  s1.Println // 86 65 53 42 36 35 30 6 4 3 – порядок задан компаратором
end.
```

Компаратор, заданный при создании множества, действует в дальнейшем при любых операциях с ним. Метод X.Comparer возвращает компаратор, использованный при создании коллекции X, что позволяет писать конструкции, подобные следующей:

```
var s2:=new SortedSet<integer>(Seq(2,7,3,4,0,9),s1.Comparer);
```

Здесь для множеств s1 и s2 используется один и тот же компаратор, что позволяет поддерживать в них один и тот же алгоритм упорядоченности.

Пусть компаратор получает на вход значения a и b. Чтобы их упорядочить по неубыванию, нужно вернуть -1 для a<b, 0 для a=b и 1 для a>b. Упорядоченность по невозрастанию достигается, если вернуть 1 для a<b, 0 для a=b и -1 для a>b.

Можно возразить, что бессмысленно анализировать случай a=b, поскольку во множестве нет одинаковых значений. Но ведь компараторы пишут не только для множеств. Кроме того, компаратор в любом случае должен возвращать какое-то значение.

Помещаемые в коллекцию SortedSet данные должны иметь тип, поддерживающий интерфейс IComparable. В противном случае при попытке добавить в коллекцию второй элемент возникнет исключение с сообщением «Ошибка времени выполнения: По крайней мере в одном объекте должен быть реализован интерфейс IComparable». Все типы PascalABC.NET, для которых определена операция сравнения, поддерживают этот интерфейс.

Если вы определяете собственный тип данных, то для помещения в коллекцию, поддерживающую упорядоченность, придется определить для него интерфейс IComparable. По этой причине собственный тип данных удобно описать, как класс.

```
// p10020
type
  Товар = class(IComparable<Товар>)
  private
    Наим: string; // наименование товара
    Цена: real; // цена единицы товара в рублях
    Кол: real// запас товара
  public
    constructor(Наим: string; Цена: real; Кол: real);
    begin
      (Self.Наим, Self.Цена, Self.Кол) := (Наим, Цена, Кол)
    end;

    function CompareTo(МойТовар: Товар): integer;
    begin
      Result := -МойТовар.Наим.CompareTo(Наим);
      if Result = 0 then Result := МойТовар.Цена.CompareTo(Цена)
    end;
end;

begin
  var Товары := new SortedSet<Товар>;
  Товары.Add(new Товар('Шкаф', 9800, 4));
  Товары.Add(new Товар('Стол', 7200, 6));
  Товары.Add(new Товар('Стул', 1930, 12));
  Товары.Add(new Товар('Стул', 2150, 8));
  Товары.Add(new Товар('Стол', 11300, 6));
  Товары.Add(new Товар('Стул', 2099, 12));
  foreach var Товар1 in Товары do
    Println(Товар1.Наим, Товар1.Цена, Товар1.Кол);
end.
```

В данном примере компаратор класса Товар обеспечивает упорядоченность элементов по полю Наим в алфавитном порядке, а при совпадении значений – по полю Цена в порядке убывания.

```
Стол 11300 6
Стол 7200 6
Стул 2150 8
Стул 2099 12
Стул 1930 12
Шкаф 9800 4
```

10.6.1.2 Дополнительные операции SortedSet

В дополнение к средствам, имеющимся у HashSet, включены следующие:

- `.Max` – свойство, возвращающее максимальное из значений элементов;
- `.Min` – свойство, возвращающее минимальное из значений элементов;
- `.GetViewBetween(minV, maxV)` – метод, возвращающий подмножество SortedSet, содержащее элементы исходного множества со значениями, входящими в диапазон от minV до maxV;
- `.Reverse` – метод, возвращающий последовательность элементов множества, расположенных в порядке, обратном порядку хранения в этом множестве.

10.6.1.3 Пример использования множества SortedSet

В примере p10012 (10.3.3) рассматривалась задача получения всех делителей натурального числа n. Был использован неоптимальный алгоритм перебора делителей до n/2. Количество циклов можно существенно сократить, если проводить перебор до √n, запоминая сразу два делителя: k и n/k, поскольку n ≡ k × (n/k). Мы не могли себе этого позволить с коллекцией List, ведь была бы утрачена упорядоченность делителей. Но SortedSet дает возможность использовать именно такой алгоритм.

Для небольших значений n этот алгоритм не демонстрирует явного превосходства, но для числа 123456789 он отработал быстрее примерно в 5000 раз!

```
begin // p10021
  var ss := new SortedSet<integer>;
  ss.Add(1);
  var n := ReadInteger('Введите натуральное число');
  if n>1 then
  begin
    ss.Add(n);
    for var i := 2 to Trunc(Sqrt(n)) do
      if n mod i = 0 then
      begin
        ss.Add(i);
        ss.Add(n div i)
      end
  end;
  ss.Println
end.
```

10.6.2 Упорядоченный список SortedList

Реализован на основе коллекции System.Collections.Generic.SortedList и создается как коллекция объектов класса SortedList<ключ, значение>. Коллекция представляет собой ассоциативный массив, построенный на базе динамического массива пар. Схож с коллекцией List, но хранит элементы структуры KeyValuePair, сохраняя их упорядоченность по ключу в соответствии с заданным компаратором.

Коллекцию можно создать с использованием оператора присваивания вида

```
var имя := new SortedList<тип_ключа, тип_значения>;
```

Имеются также варианты конструктора, в которых указывается компаратор.

Создание списка SortedList можно совместить с его заполнением данными из словаря Dictionary.

```
begin
var d := Dict(Arr(('b1', 216), ('a2', 142), ('d3', 18)));
  d.Println; // (b1,216) (a2,142) (d3,18)
  var ss := new SortedList<string, integer>(d);
  ss.Println // (a2,142) (b1,216) (d3,18)
end.
```

Ключ имеет тип string, для которого в языке определен интерфейс IComparable, поэтому проблем с добавлением данных не возникает, а сами данные будут отсортированы по ключу в алфавитном порядке.

«Короткая функция» наподобие SSet, позволяющая создать упорядоченный список SortedList, в языке отсутствует. Также, отсутствует возможность создавать список SortedList при помощи расширения по типу .ToList.

В работе со списком поддерживаются следующие средства:

- `.Count` – свойство, возвращающее количество элементов в списке;
- `.Keys` – свойство, возвращающее последовательность ключей, имеющихся в списке;
- `.Values` – свойство, возвращающее последовательность значений, имеющихся в списке;
- `.Add(ключ, значение)` – метод, добавляющий в список элемент с указанными ключом и значением. При попытке добавления элемента с ключом, уже имеющимся в списке, возникает исключение;
- `.Clear` – метод, очищающий список;
- `.ContainsKey(ключ)` – метод, возвращающий True, если в списке имеется элемент с указанным ключом и False в противном случае (проверка на наличие ключа в списке);
- `.ContainsValue(значение)` – метод, возвращающий True, если в списке имеется элемент с указанным значением и False в противном случае (проверка на наличие значения в списке);

- `.Get(ключ)` – расширение PascalABC.NET, возвращающее значение элемента, связанного с указанным ключом или значение по умолчанию, если ключ не найден;
- `.IndexOfKey(ключ)` – метод, возвращающий индекс первого найденного элемента с указанным ключом. Если поиск неуспешен, возвращается отрицательное значение;
- `.IndexOfValue(значение)` – метод, возвращающий индекс первого найденного элемента с указанным значением. Если поиск неуспешен, возвращается отрицательное значение;
- `.Remove(ключ)` – метод, удаляющий из списка элемент с указанным ключом. Если ключ не найден, ничего не происходит;
- `.RemoveAt(индекс)` – метод, удаляющий из списка элемент с указанным индексом. Если такой индекс отсутствует, возникает исключение;
- `.ToArray` – метод, копирующий содержимое списка в динамический массив, каждый элемент которого имеет тип <Key, Value> и возвращающий этот массив в качестве результата. Не требует предварительного описания массива. Обращение к полям элемента массива возможно в виде имя[индекс].Key и имя[индекс].Value;
- `.TryGetValue(ключ, имя_переменной)` – метод, выполняющий поиск элемента списка по указанному ключу. Если поиск успешен, переменная заполняется значением из найденного элемента и возвращается True. В противном случае возвращается False;
- `.Print` – стандартный метод PascalABC.NET. Осуществляет вывод содержимого списка от его начала. В качестве параметра можно указать символ или строку - разделитель, по умолчанию используется пробел;
- `.Println` – то же самое; в конце вывода осуществляется переход к новой строке.

Список SortedList может быть преобразован к последовательности элементов типа KeyValuePair, из которой впоследствии можно сформировать другие стандартные коллекции посредством методов .ToDictionary, .ToHashSet, .ToLinkedList, .ToList и .ToSortedSet. К этим последовательностям также можно применять любые другие методы последовательностей.

Когда и зачем используют список SortedList? Сложно дать определенный ответ. Например, в языке Java коллекция SortedList вообще отсутствует. Неясно, почему столько внимания уделяется поиску индексов, если основная часть операций происходит с ключом, как в словаре, а индексы используются только для удаления элементов. Выражение вида S[k] означает не элемент словаря S с индексом k, а элемент с ключом k. Да и удалить элемент можно по значению ключа, не прибегая к индексу.

10.6.3 Упорядоченный словарь SortedDictionary

Реализован на основе коллекции System.Collections.Generic.SortedDictionary и создается как коллекция объектов класса SortedDictionary<ключ, значение>. Коллекция представляет собой ассоциативный массив, построенный на базе так называемого красно-черного бинарного дерева поиска. Схож с коллекцией Dictionary и

хранит элементы структуры KeyValuePair, сохраняя их упорядоченность по ключу в соответствии с заданным компаратором. Пожалуй, это и все отличия SortedDictionary от Dictionary, даже способы создания, свойства и методы у них одни и те же. Посему материал, приведенный в 10.5.2 и 10.5.3 в равной степени относится к обеим упомянутым коллекциям.

Словарь SortedDictionary имеет смысл использовать, если имеется настоятельная необходимость в постоянной поддержке упорядоченности ключей при частой вставке новых элементов. В противном случае выгоднее отсортировать данные и загрузить их в обычный словарь.

10.7 Для самостоятельного решения

T10.1. Реализуйте дек на основе двух стеков (см. 10.2.1.4). Выполните с деком операции, приведенные в задаче p10010 за исключением попытки чтения данных из пустого дека. Сравните выведенные результаты. Реализация проверки дека на пустоту не требуется.

T10.2. Сформируйте очередь из 25 натуральных чисел в диапазоне от 10 до 99. Выведите значения ее элементов. Осуществите циклический сдвиг элементов влево так, чтобы в полученной очереди максимальный элемент стал первым. Если максимальных элементов несколько, используйте первый из них.

T10.3. Переставьте отрицательные элементы заданного списка в конец, а положительные – в начало списка. После формирования списка нельзя использовать дополнительные последовательности, массивы и иные коллекции.

T10.4. Создайте множество натуральных чисел, не превышающих 100 000. Разбейте его на два подмножества, первое из которых содержит все простые числа исходного множества, а второе — все остальные его члены.

Часть 11

Модули и библиотеки

Самым важным отличием хорошего модуля от плохого является степень, в которой он скрывает [...] детали реализации.

Джошуа Блох

Во Введении было сказано, что одними из первых парадигм программирования были процедурная и возникшая на ее основе парадигма структурного программирования. Обе парадигмы предполагают построение программы из отдельных программных блоков. Появившаяся впоследствии парадигма ***модульного программирования*** предполагает построение таких блоков в виде программных компонент особого вида – ***модулей***. Модуль позволяет разбить текст программы на несколько физических файлов, компилируемых раздельно. Это дает возможность автономно отлаживать и тестировать модули, а также во многом снимает проблему внесения в готовый код непреднамеренных и несанкционированных изменений.

Главную концепцию модуля кратко сформулировал в 1972 г. Д. Парнас: «Для написания одного модуля должно быть достаточно минимальных знаний о тексте другого». Первым языком, полноценно использующим модульную концепцию, был разработанный Н. Виртом в 1975 г. язык Modula-2. Г. Майерс в монографии «Композиционное проектирование приложения» (2009 г.) предложил методологию проектирования модулей.

Но еще задолго появления концепции модулей, на заре становления программирования, при написании программ начали применять ***библиотеки стандартных программ***. Идея использования библиотек состоит во включении в программный код обращений к заранее написанным фрагментам, реализующим некоторые стандартные операции. Математические библиотеки используются для вычисления различных математических функций (таких, как квадратный корень, синус, логарифм). С помощью библиотек ввода-вывода преобразуют вводимые данные в формат, необходимый для вычислений и обеспечивают оформление вывода результатов. Библиотеки математических методов позволяют, к примеру, оперировать матрицами, решать уравнения, находить значения специальных функций. Существуют библиотеки, решающие прикладные задачи в науке, технике и народном хозяйстве.

Библиотека стандартных программ содержит реализацию множества процедур и функций, связанных друг с другом только общей тематикой. Объединяет модули и библиотеки концепция самостоятельной компиляции, отладки и тестирования.

В языке PascalABC.NET и модули, и то, что было выше определено, как библиотеки, оформляются в виде модулей. А библиотеками принято называть самостоятельные программные единицы, динамически загружаемые в память из файлов с расширением .dll (Dynamic-Link Libraries). И если из модуля выбираются только

нужные вызывающей программе компоненты, то библиотека всегда подсоединяется в полном объеме.

11.1 Модули

Модули позволяют разбивать программу на несколько самостоятельных файлов, компилируемых раздельно. В модулях могут описываться различные объекты программы, например, константы, типы, переменные, классы, процедуры, функции. Для того, чтобы модуль можно было использовать, в разделе uses вызывающего модуля (которым может быть и основная программа) нужно указать имя требуемого файла модуля без расширения. Откомпилированный файл модуля (.pcu) должен находиться либо в том же каталоге, что и основная программа, либо в подкаталоге \Lib системного каталога программы PascalABC.NET. Исходный файл модуля (.pas) может находиться либо в том же каталоге, что и основная программа, либо путь к нему явно указывается в секции **uses**.

11.1.1 Структура модуля

В языке PascalABC.NET модуль имеет определенное синтаксическое оформление со следующей структурой:

unit имя_модуля	Заголовок модуля
interface	Раздел интерфейса (необязательный)
код раздела	
implementation	Раздел реализации
код раздела	
initialization	Раздел инициализации (необязательный)
код раздела	
finalization	Раздел финализации (необязательный)
код раздела	
end.	

Если раздел реализации единственный, заголовок **implementation** можно не указывать.

11.1.1.1 Заголовок модуля

Это первая, обязательная строка, открывающая программный код модуля. Она начинается ключевым словом **unit** и объявляет имя модуля, которое **обязано совпадать с именем файла**.

11.1.1.2 Раздел интерфейса

Начинается строкой с ключевым словом **interface**.

Разобраться в том, что такое интерфейс модуля, поможет пример. Допустим, у нас есть электрический чайник. Что можно о нем сказать? Чайник служит для того, чтобы нагревать воду до кипения. Нужно подключить чайник к электрической сети, налить в него воду и нажать кнопку включения. По окончании процедуры нагрева чайник будет автоматически отключен. Поздравляю: вы только что прочитали описание интерфейса чайника – его назначение (нагревать воду), набор операций (греть воду, выключаться) и условия работы (подано рабочее напряжение, налита вода, нажата кнопка включения). Если взять другой чайник, можно не беспокоиться, что вы не сумеете сделать себе чай, потому что интерфейс всех чайников практически одинаков. Даже никогда прежде не имея электрического чайника, но ознакомившись с его интерфейсом, вы будете знать, зачем нужен чайник, и что нужно сделать, чтобы им воспользоваться.

Интерфейс модуля – это договор о том, что необходимо дать модулю для выполнения им заранее оговоренного набора функций. Это лишь внешнее представление о модуле, но иного нам не дано. Теперь должно стать понятно, почему знание интерфейса модуля позволяет выполнить компиляцию вызывающей программной единицы, не имея самого модуля.

Раздел интерфейса содержит объявление всех имен, присутствующих в модуле, которые будут доступны (чаще говорят «видны») в других модулях при его подключении посредством раздела **uses**. Видимыми можно объявить константы, переменные, функции, процедуры, функции, типы, классы и интерфейсы. Раздел интерфейса в свою очередь может начинаться с раздела **uses**, позволяющего подключить к модулю внешние модули и ***пространства имен*** .NET (см. 11.1.1.8). Также, в разделе интерфейса можно указать и ***реализацию***, не создавая с этой целью отдельный раздел, но такая практика не является правильной, поскольку нарушает основной принцип Парнаса.

11.1.1.3 Раздел реализации

Начинается строкой с ключевым словом **implementation**.

Предположим, что вы ознакомились с интерфейсом электрического чайника. Нужны ли вам знания о том, как чайник устроен внутри для того, чтобы им пользоваться? Ответ однозначен: нет. И только если вы конструктор чайника, его изготовитель или мастер по ремонту бытовой техники, это знание становится необходимым. Внутреннее устройство чайника программисты могли бы назвать ***реализацией*** его интерфейса.

Реализация интерфейса модуля на практике означает написание некоторого кода, выполняющего описанный интерфейс. Интерфейс должен быть корректно реализован с точки зрения соответствия имен, типов и значений объявленных в нем объектов.

Раздел реализации содержит программный код всех процедур, функций и методов, объявленных в разделе интерфейса. Если подпрограмма объявлена в разделе

интерфейса, то в разделе реализации после имени подпрограммы допускается не указывать список параметров и содержащие его круглые скобки. Также, в разделе реализации могут присутствовать описания внутренних имен, которые не должны быть видны вне модуля. Именно по этой причине рекомендуется отделять реализацию от интерфейса.

Как и раздел интерфейса, раздел реализации может начинаться с раздела **uses**, служащего для подключения внешних модулей и пространств имен .NET. В случае, когда раздел интерфейса также содержит раздел **uses**, перечисленные в обоих разделах имена не могут пересекаться.

11.1.1.4 Раздел инициализации

Начинается строкой с ключевым словом **initialization**. Может отсутствовать.

Как следует из названия, раздел инициализации служит для задания начальных значений переменным модуля. Программный код этого раздела выполняется раньше, чем код раздела реализации (и/или раздела интерфейса, если там имеется код). В то же время, нельзя однозначно сказать, в какой момент будет выполняться код раздела инициализации по отношению к коду программной единицы, содержащей вызов модуля.

```
unit Demo;

interface

var
  a, b: integer;

function Summa(i1, i2: integer): integer;
implementation

function Summa(i1, i2: integer): integer;
begin
  Result := i1 + i2
end;

initialization
  a := 10;
  b := 20
end.
```

```
uses Demo;

begin
  Println('3 + 2 =', 3 + 2);
  Print(a, b);
  Summa(a, b).Println
end.
```

В приведенном примере мы не знаем, инициализированы ли переменные *a* и *b* в момент исполнения оператора Println('3 + 2 =', 3 + 2). Другое дело, что это знание нам в данном случае не нужно. Но представьте ситуацию, в которой имеется описание uses Demo1, Demo2. Корректно ли обращаться в коде раздела реализации модуля Demo1 к переменным, указанным в интерфейсе модуля Demo2 и получающим значение в его разделе инициализации? Тут будет полезным вспомнить работу Г. Майерса и не писать подобного кода, производящего сильное сцепление двух модулей.

11.1.1.5 Раздел финализации

Начинается строкой с ключевым словом **finalization** и позволяет указать часть программного кода, который выполняется после завершения кода, указанного в разделе реализации. Раздел финализации может отсутствовать.

11.1.1.6 Раздел uses

Раздел подключаемых модулей **uses** состоит из одной или нескольких подряд идущих секций **uses**, каждая из которых имеет вид:

```
uses список_имен;
uses имя in путь_к_файлу\имя.pas;
```

Имена в списке перечисляются через запятую. Это могут быть имена подключаемых внешних модулей PascalABC.NET и/или наименования пространств имен .NET, например:

```
uses System, System.Numerics, Unit13;
```

Здесь Unit13 – имя модуля, представленного в виде исходного текста PascalABC.NET или откомпилированного файла .pcu, System и System.Numerics – имена пространств имен .NET.

В модуле или основной программе, содержащей раздел uses, можно использовать все имена из списка подключаемых модулей PascalABC.NET и пространств имен .NET.

Основное отличие между модулями и пространствами имен .NET состоит в том, что модуль содержит исполняемый код, а пространства имен .NET – лишь перечисление имен. Чтобы использовать код из пространства имен, нужно подключить хранящий его файл при помощи директивы компилятора {$**reference** имя_сборки), где имя_сборки – это имя файла dll, содержащего код Microsoft .NET.

Еще одно важное отличие состоит в том, что в модуле или основной программе невозможно использовать имена, определенные в другом модуле, если не подключить этот модуль посредством раздела **uses**. В то же время, если сборка .NET подключена директивой $**reference**, можно использовать ее имена (указывая перед ними пространство имен) без подключения этого пространства имен в разделе **uses**.

```
begin
  System.Console.WriteLine('Строка 1');
  System.Console.WriteLine('Строка 2');
  System.Console.WriteLine('Строка 3')
end;
```

```
uses System.Console;

begin
  WriteLine('Строка 1');
  WriteLine('Строка 2');
  WriteLine('Строка 3');
end.
```

Какой вариант выбрать – исключительно дело вкуса.

По умолчанию в первой секции раздела uses неявно (т.е. даже если его не указывать) всегда подключается системный модуль PABCSystem, содержащий стандартные константы, типы, процедуры и функции. После него, также неявно, подключаются сборки System.dll, System.Core.dll и mscorlib.dll, содержащие основные .NET-типы.

Могут ли в подключаемых модулях и пространствах имен встретиться одинаковые имена? Безусловно. Что происходит в таких случаях, будет рассмотрено ниже.

Если в подключаемых модулях кроме описаний присутствует исполняемый код, он будет исполнен до первого исполняемого оператора в основной программе. Если в uses перечислено несколько имен, соответствующие модули будут исполняться в порядке, обратном порядку перечисления (от последнего имени к первому).

11.1.1.7 Локальные и глобальные имена

Имена, описанные в некотором модуле, видны в нем, а также во всех подпрограммах и т.д., описанных внутри него. Такие имена называются ***глобальными***. В коде самих подпрограмм могут иметься собственные описания, которые действуют только в пределах этого кода и называются ***локальными***. Аналогичным образом ведут себя блоки: имя, локальное в блоке, будет глобальным для всех вложенных блоков, а вне блока локальная переменная не видна. С точки зрения логики выполнения программного кода можно считать, что локальные объекты вне блока не существуют: они создаются при входе в блок и уничтожаются при выходе из него. Сакральное знание о том, как в действительности происходит работа с памятью, не является для программиста необходимым, если только он не особо озабочен вопросами повышения эффективности работы программы.

11.1.1.8 Пространство имен

Пространство имен модуля – это абстрактное хранилище, с которым ассоциируется все множество **уникальных** имен, объявленных в этом модуле. Имена, объявленные в пространстве имен модуля, считаются глобальными и видны внутри всего этого модуля, образуя глобальное пространство имен. Внутри блоков **begin** … **end** могут быть объявлены так называемые ***внутриблочные переменные***, образующие локальные пространства имен. В модуле могут быть определены функции, процедуры, классы и т.д. Каждое из таких определений может иметь собственные пространства имен. Пространства имен могут вкладываться друг в друга, при этом каждое внешнее пространство имен по отношению ко вложенным будет играть роль глобального.

Если в каком-то месте программного кода будут одновременно видны два одинаковых имени, возникнет коллизия: компилятор не сможет определить, какое из этих имен использовать. Такая ситуация трактуется, как ошибка, делающая невозможной дальнейшую компиляцию программы. С ростом сложности программ вероятность возникновения коллизий возрастает. Свой неприятный вклад могут

внести и пространства имен модулей, особенно, если эти модули написаны разными программистами. При подключении модуля его глобальное пространство имен становится видным во всей программной единице, где выполнено подключение. Поэтому существует запрет на видимость в любом месте программы хотя бы двух одинаковых имен.

11.1.1.9 Область видимости имен

Согласно правилам языка Паскаль, любое имя должно быть описано до момента первого его использования. Имя может быть описано в разделе описаний программной единицы или быть внутриблочным. Некоторые разновидности операторов цикла рекомендуют описывать управляющие переменные в заголовке, например

```
for имя: тип := значение … do …
for var имя := значение … do … // тип выводится автоматически
foreach имя: тип in … do …
foreach var имя in … do … // тип выводится автоматически
```

Имя из раздела описаний некоторой программной единицы, является в ней глобальным и видно в любом ее месте. Внутриблочное имя локально и видно только внутри блока **с момента своего описания** (это очень важный момент). Переменная, описанная в заголовке цикла, считается внутриблочной и видна в пределах действия оператора цикла. Эти правила видимости проверяются на этапе компиляции и позволяют компилятору более эффективно управлять памятью, а программисту – использовать имена, не беспокоясь об их возможном совпадении с именами в других программных единицах.

> В PascalABC.NET принято правило, согласно которому внутриблочные имена не могут совпадать с именами, находящимися в разделе описаний программной единицы или во внешнем блоке.

Не возникает ли коллизия из-за совпадения объявленных в некоторой программной единице имен переменных с именами, которые оказываются видимыми в ней вследствие их указания в интерфейсе подключаемых модулей? Не возникает. Объявленным переменным отдается приоритет. И это вполне логично: разрабатывая код программы, мы не должны озадачиваться именами, которые окажутся видны при подключении внешних модулей, если только в этом нет некоторого специального смысла (значения констант, начальная инициализация и т.д.). Чтобы понимать механизм разрешения коллизий, руководствуйтесь следующим:

> Поиск глобальных имен осуществляется вначале в текущем модуле или основной программе, затем во всех подключенных по **uses** модулях и пространствах имен в порядке, обратном порядку их перечисления (т.е. справа налево) и в последнюю очередь в подключенных директивой $**reference** библиотеках в порядке подключения. При этом считается, что приоритет имеют имена, найденные первыми.

Если необходимо указать имя из конкретного модуля или пространства имен, следует использовать один из следующих форматов:

```
ИмяМодуля.Имя
ИмяПространстваИменNET.Имя
```

В качестве имени модуля может выступать также имя основной программы, если у нее присутствует заголовок **program**. Рассмотрим пример.

```
// p11001
uses p11002, p11003;

function f1(a: integer): integer;
begin
  Result := a * a * a;
end;

begin
  Println('Начало основного модуля');
  for var i := 1 to 2 do
  begin
    var a: T1;
    a.a := 3 * i + 1;
    a.b := 2 * i - 1;
    P1(a);
    f1(a.a).Println;
  end;
  Println;
  var i := 7;
  var a := ArrRandom(i, 10, 99);
  a.Println;
  Println('p11001:', f1(i - 2))
end.
```

```
unit p11002;

interface
type
  T1 = record
    a: integer;
    b: real
  end;
procedure p1(a: T1);
function f1(a: integer): integer;

implementation

procedure p1(a: T1);
begin
  Println(a.a, ' ', a.b)
end;

function f1(a: integer) := a mod 10;

var
  a: integer;

begin
  a := 25;
  Println('Модуль p11002:', f1(a))
end.
```

```
unit p11003;

interface
function f1(a: integer): integer;

implementation

function f1(a: integer) := 10 * a;

begin
  var a := 3;
  Println('Модуль p11003:', f1(a))
end.
```

Слева приведена программа, в которой подключены модули p11002 и p11003; их коды помещены справа. В соответствии с интерфейсом, в модуле p11002 видны тип *T1*, процедура *p1* и *функция f1(**integer**): **integer***, а в модуле p11003 – только функция *f1(**integer**): **integer***. В вызывающей программе также описана функция *f1(**integer**): **integer***. В основной части вызывающей программы имеется цикл, содержащий в заголовке объявление переменной *i*, которая видна только в теле цикла. В теле цикла объявлена внутриблочная переменная *a* типа *T1*; она видна только в блоке. За пределами блока (причем, текстуально **ниже** его – и это очень важно!), имеются объявления переменной *i*, а также массива *a*.

Рассмотрим формирование пространства имен в программе p11001. Его собственное пространство имен P_{p11001} первоначально будет содержать только имя функции *f1*. Далее будет просматриваться модуль p11003 и последним – модуль p11002. Компилятор построит множество, содержащее объединение пространств имен:

$$P_{p11001} = \{\mathrm{f1}_{p11001}\} \cup \{\mathrm{f1}\}_{p11003} \cup \{\mathrm{T1},\ \mathrm{p1},\ \mathrm{f1}\}_{p11002} = \{\mathrm{f1}_{p11001},\ \mathrm{T1}_{p11002},\ \mathrm{p1}_{p11002}\}$$

Коллизия имен не возникает благодаря правилу просмотра модулей при формировании пространства имен. В основной программе будет видна описанная в ней функция *f1*, а также тип *T1* и процедура *p1*, описанные в модуле p11002.

При входе в цикл сформируется внутриблочное пространство имен, содержащее простую переменную *i* типа **integer** и запись *a* типа *T1*. После завершения блока это внутриблочное пространство станет недоступным, а к пространству имен P_{p11001} будут добавлены имена простой переменной *i* типа **integer** и динамического массива *a*, содержащего элементы типа **integer**:

$$P_{p11001} = \{\mathrm{f1}_{p11001},\ \mathrm{T1}_{p11002},\ \mathrm{p1}_{p11002},\ \mathrm{i}_{p11001},\ \mathrm{a}_{p11001}\}$$

Ниже показан пример работы программы p11001.

```
Модуль p11003: 30
Модуль p11002: 5
Начало основного модуля
4   1
64
7   3
343

32 29 12 31 54 49 49
p11001: 125
```

Вначале выполнился код модуля p11003, поскольку он указан последним в разделе **uses**. Следующим выполнился код модуля p11002. И только потом начал выполняться код основной программы p11001. В теле цикла вызывалась функция *f1*. В соответствии с пространством имен это была функция, описанная в основной программе. Эта же функция вызывалась и после выхода из блока.

Поставим эксперимент. В программе p11001 заменим в заголовке функции *f1* тип возвращаемого значения на **real**. Компиляция по-прежнему проходит нормально, и по-прежнему вызывается функция *f1* из p11001. Смена типа возвращаемого значения в основной программе не изменила пространства имен, поскольку возвращаемое функцией значение не входит в параметры, по которым компилятор выбирает подключаемую функцию.

Рассмотрим еще один пример.

```
// p11004
uses p11005, p11006;

begin
  f1(5).Println // 50
end.
```

```
unit p11005;

interface

function f1(a: integer): integer;

implementation

function f1(a: integer) := a mod 10;
end.

unit p11006;

interface

function f1(a: integer): integer;

implementation

function f1(a: integer) := 10 * a;
end.
```

Здесь в строгом соответствии с правилом просмотра подключаемых модулей будет выбрана функция *f1* из модуля p11006, идущего последним в секции **uses**. При указании ***uses*** *p11006, p11005*, будет подключена функция *f1* из p11005 и программа выведет значение 5 (попробуйте!).

Если нужно вызывать функцию из определенного модуля, укажите ее имя вместе с «фамилией» - именем модуля, например, p11005.f1 или p11006.f1. Использовать уточненное имя проще, чем переставлять местами имена подключаемых модулей.

В случае, когда типы параметров в одноименных функциях будут различными, компилятор выполнит перегрузку имен (см. 4.6) и все фактически вызываемые функции будут видны одновременно.

11.1.1.10 Упрощенный синтаксис модуля

Модуль с упрощенным синтаксисом содержит только раздел описаний и/или раздел инициализации. Раздел интерфейса отсутствует, что требует повышенного внимания при использовании таких модулей из-за возможности неожиданно

столкнуться с перегрузкой имен подпрограмм. Такие модули удобно использовать при начальном обучении языку.

```
unit имя_модуля;
  раздел_описаний
end.
```

```
unit имя_модуля;
  раздел_описаний
begin
  раздел_реализации
end.
```

```
unit имя_модуля;
begin
  раздел_реализации
end.
```

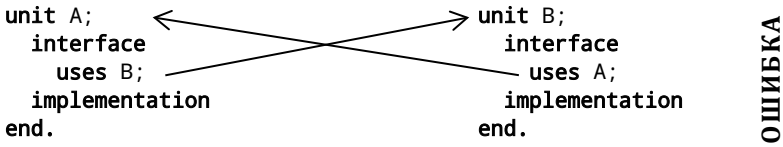
Коды, приведенные в примере из конца предыдущего подраздела, можно существенно сократить:

```
uses p11005, p11006;

begin
  f1(5).Println // 50
end.
```

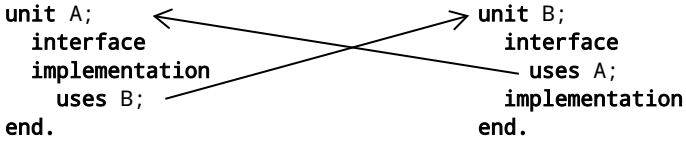
```
unit p11005;

function f1(a: integer) := a mod 10;
end.

unit p11006;

function f1(a: integer) := 10 * a;
end.
```

11.1.2 Циклические ссылки между модулями

Циклические ссылки на модули в интерфейсных частях запрещены. Например, следующая ситуация считается ошибочной и выявляется компилятором:



Циклические ссылки разрешаются в случае, когда хотя бы одна из ссылок находится в части реализации:



11.2 Библиотеки dll

Библиотеки dll обычно содержат группу программ, объединенных какими-либо общими признаками, например, назначением. Библиотека всегда является откомпилированным файлом; к ней можно обращаться из различных программ. Библиотека может быть написана на одном языке программирования, а обращаться к ней можно из программы на другом языке. Файл с библиотекой может быть локаль-

ным и находиться в текущем каталоге, либо глобальным, находящимся в системном каталоге. Глобальными библиотеками могут пользоваться одновременно несколько приложений.

Библиотеки по своему назначению похожи на МОДУЛИ, НО имеют некоторые отличия:

- При создании из модулей исполняемого файла в него будут помещены только те подпрограммы, переменные, типы и константы, которые используются в основной программе. А при компиляции библиотеки в исполняемый файл добавляются все подпрограммы, поскольку неизвестно, какие из них будут востребованы;
- При выполнении программы библиотеки полностью загружаются в оперативную память;
- Библиотеки могут использоваться одновременно несколькими программами.

11.2.1 Структура библиотеки

Библиотека в PascalABC.NET имеет практически ту же структуру, что и МОДУЛЬ.

library имя_библиотеки	Заголовок модуля
interface	Раздел интерфейса (необязательный)
код раздела	
implementation	Раздел реализации
код раздела	
end.	

Как и у модуля, имя библиотеки должно совпадать с именем файла .pas, в котором находится ее исходный код. После компиляции в текущем каталоге будет создан файл .dll, содержащий откомпилированную библиотеку. Имеется также упрощенный синтаксис, который за исключением строки заголовка совпадает с упрощенным синтаксисом для модулей (см. 11.1.1.10).

11.2.2 Подключение библиотек

Для подключения библиотеки к основной программе используется директива компилятора {$**reference** имя_файла}. Имя файла обязательно включает и расширение. Эта директива может быть указана в любом месте программного кода.

```
{$reference ABC1.dll}
{$reference ABC2.dll}
begin
  Writeln(a.GetType)
end.
```

```
library ABC1;
  var a: integer
end.
```

```
library ABC2;
  var a: real
end.
```

Согласно порядку просмотра пространств имен переменная *a* будет иметь тип **integer** (см. 11.1.1.9).

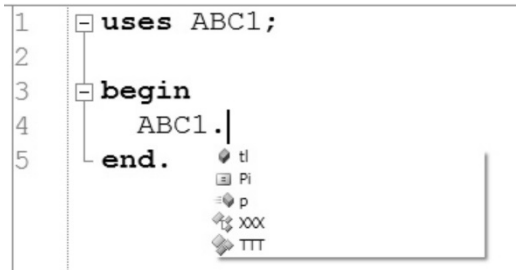
Как и в случае с модулями, при совпадении имен можно воспользоваться уточнением. Например, в приведенном примере можно было писать ABC1.a или ABC2.a

11.3 Документирующие комментарии

При разработке подключаемых модулей и библиотек полезно использовать ***документирующие комментарии***, которые всплывают в подсказках редактора PascalABC.NET при наведении курсора мыши на имя, при открытии скобки после имени подпрограммы и при выборе поля из списка, открывающегося при нажатии точки после имени. Можно документировать заголовки подпрограмм и методов, имена классов, типов, констант и переменных.

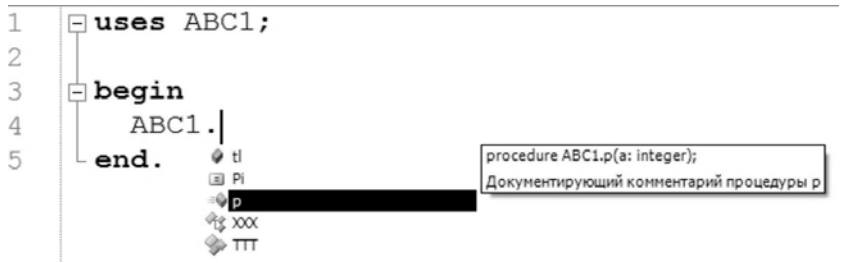
Документирующий комментарий располагается на строчке, предшествующей помечаемому объекту и начинается с трех символов «слэш» (///). Если комментарий содержит несколько строк, каждая из них должна начинаться тремя слэшами. Ниже приведен пример на основе Справки PascalABC.NET:

```
unit ABC1;

interface

const
  /// Константа Pi
  Pi = 3.14;

type
  /// TTT - синоним целого типа
  TTT = integer;

  /// Документирующий комментарий класса XXX
  XXX = class
  end;

/// Документирующий комментарий процедуры p
procedure p(a: integer);

var
  /// Документирующий комментарий переменной tl
  tl: TTT;

implementation

procedure p(a: integer);
begin
end;

end.
```



На приведенном рисунке показан один из способов просмотра интерфейса подключаемого модуля – указание точки после его имени. Значки перед именем позволяют понять, к какой разновидности объектов программы оно относится.

Чтобы получить более подробную информацию, нужно выбрать интересующее поле.



Если возникает надобность изменить заголовок подпрограммы во всплывающей подсказке, в первой строке документирующего комментария после трех слэшей добавляется дефис, например:

```
///- Exclude(var s: set of T; el: T)
/// Удаляет элемент el из множества s
procedure Exclude(var s: TypedSet; el: object);
```

Здесь во всплывающей подсказке будет указан текст

```
procedure Exclude(var s: set of T; el: T);
Удаляет элемент el из множества s
```

Всплывающую подсказку можно скрыть, указав в первой строке документирующего комментария после трех слэшей два дефиса (///--).

11.4 Пример программы

Для некоторого, определенного вводом количества треугольников, заданных случайными координатами вершин, требуется вывести длины сторон и площади. Затем вывести значение максимальной из площадей. Координаты вершин задаются вещественными числами из диапазона [-10; 10] с точностью до 0.1.

В программе используются пользовательские типы, являющиеся записями (**record**). Определяются типы для точки, линии (стороны) и треугольника. Площадь треугольника вычисляется на основе длин его сторон по хорошо известной формуле Герона.

11.4.1 Реализация на основе модулей

В демонстрационных целях модули разделены. Модуль p11008 содержит описания пользовательских типов, модуль p11009 содержит функции для создания и инициализации переменных пользовательских типов, в модуле p11010 собраны подпрограммы, реализующие необходимые вычисления и вывод.

```
// p11007
uses p11008, p11010;

begin
  var КоличествоТреугольников := ReadInteger('Число треугольников =');
  var Треугольники := СгенерироватьТреугольники(КоличествоТреугольников);
  foreach var Треугольник in Треугольники do
    СведенияОТреугольнике(Треугольник);
  Print('Максимальная площадь тругольника');
  Треугольники.Select(Треугольник -> ПлощадьТреугольника(Треугольник))
      .Max.Println
end.
```

```
unit p11008;

type
  /// Точка на плоскости
  TPoint = record
    x: real;
    y: real
  end;

  /// Линия на плоскости
  TLine = record
    A: TPoint;
    B: TPoint
  end;

  /// Треугольник на плоскости
  TTriangle = record
    a: TLine;
    b: TLine;
    c: TLine
  end;

end.
```

```
unit p11009;

interface

uses p11008;
  /// Возвращает точку с заданными координатами x и y
function СоздатьТочку(x, y: real): TPoint;
  /// Возвращает линию, заданную парой точек Pa и Pb
function СоздатьЛинию(Pa, Pb: TPoint): TLine;
  /// Возвращает треугольник, заданный сторонами a, b, c
function СоздатьТреугольник(a, b, c: TLine): TTriangle;

implementation

function СоздатьТочку(x, y: real): TPoint;
begin
  (Result.x, Result.y) := (x, y);
end;

function СоздатьЛинию(Pa, Pb: TPoint): TLine;
begin
  (Result.A, Result.B) := (Pa, Pb);
end;

function СоздатьТреугольник(a, b, c: TLine): TTriangle;
begin
  (Result.a, Result.b, Result.c) := (a, b, c)
end;

end.
```

```
unit p11010;

interface

uses p11008, p11009;

/// Возвращает длину линии
function ДлинаЛинии(L: TLine): real;
/// Возвращает площадь треугольника с указанными сторонами
function ПлощадьТреугольника(p: TTriangle): real;
/// Выводит координаты вершин треугольника
procedure ВыводКоординатВершин(p: TTriangle);
/// Выводит сведения о треугольнике с площадью S
procedure СведенияОТреугольнике(p: TTriangle; S: real := 0);
/// Возвращает массив n треугольников, сгенерированных случайным образом
function СгенерироватьТреугольники(n: integer): array of TTriangle;
```

```
implementation

function ДлинаЛинии(L: TLine) :=
Sqrt(Sqr(L.B.x - L.A.x) + Sqr(L.B.y - L.A.y));

function ПлощадьТреугольника(p: TTriangle): real;
begin
  var ДлинаAB := ДлинаЛинии(p.a);
  var ДлинаBC := ДлинаЛинии(p.b);
  var ДлинаAC := ДлинаЛинии(p.c);
  var Полупериметр := (ДлинаAB + ДлинаBC + ДлинаAC) / 2;
  Result := Sqrt(Полупериметр * (Полупериметр - ДлинаAB) *
      (Полупериметр - ДлинаBC) * (Полупериметр - ДлинаAC))
end;

procedure ВыводКоординатВершин(p: TTriangle);
begin
  Writeln(p.a.A, ', ', p.a.B, ', ', p.b.B)
end;

procedure СведенияОТреугольнике(p: TTriangle; S: real);
begin
  Print('Координаты вершин');
  ВыводКоординатВершин(p);
  Println('Длины сторон треугольника:',
      ДлинаЛинии(p.a), ДлинаЛинии(p.b), ДлинаЛинии(p.c));
  Print('Площадь треугольника:');
  if S = 0 then Println(ПлощадьТреугольника(p))
  else Println(S)
end;

function СгенерироватьТреугольники(n: integer): array of TTriangle;
begin
  Result := new TTriangle[n];
  for var i := 0 to n - 1 do
  begin
    var Координаты := SeqRandom(6, -100, 100)
        .Select(t -> Round(t / 10, 1)).ToArray;
    var Pa := СоздатьТочку(Координаты[0], Координаты[1]);
    var Pb := СоздатьТочку(Координаты[2], Координаты[4]);
    var Pc := СоздатьТочку(Координаты[4], Координаты[5]);
    var Lab := СоздатьЛинию(Pa, Pb);
    var Lbc := СоздатьЛинию(Pb, Pc);
    var Lac := СоздатьЛинию(Pa, Pc);
    Result[i] := СоздатьТреугольник(Lab, Lbc, Lac)
  end
end;

end.
```

Далее приведен пример работы программы:

```
Число треугольников = 3
Координаты вершин (9.9,0.3), (0.2,-1.7), (-1.7,9.9)
Длины сторон треугольника: 9.9040395798886 11.7545735779738 15.057224179775
Площадь треугольника: 58.16
Координаты вершин (-9.7,-5.6), (6.4,9.4), (9.4,7.1)
Длины сторон треугольника: 22.0047722096822 3.78021163428716 22.9368698823532
Площадь треугольника: 41.015
Координаты вершин (3.3,-7.7), (6.8,-1.8), (-1.8,-9.3)
Длины сторон треугольника: 6.860029154457 11.4109596441316 5.34509120595711
Площадь треугольника: 12.245
Максимальная площадь тругольника 58.16
```

В модуле p11009 посредством **uses** подключен модуль p11008. Это позволяет пользоваться объявленными в p11008 типами. В модуле p11010 посредством **uses** подключены модули p11008 и p11009. Такое подключение позволяет использовать объявленные в p11008 типы данных и обращаться к функциям, реализованным в p11009. В основной программе p11007 секция **uses** подключает модуль p11008 для доступа к пользовательским типам и модуль p11010 для вызова определенных там подпрограмм. Модуль p11009 в основной программе подключать не требуется, поскольку обращения к его функциям отсутствуют.

Еще раз хочется отметить, что на практике для подобных задач нет необходимости (да и большого смысла) настолько дробить модули. Вполне разумно слить модули p11008 и p11010 в единый модуль p11008. Конечно, при этом модулю p11009 понадобятся объявления из p11008, а модулю p11008 понадобятся функции из p11009. Возникнет циклическая ссылка (см. 11.1.2). Выход уже указывался: например, в модуле p11008 секцию **uses** с указанием имени модуля p11009 можно переместить в раздел реализации.

Несмотря на то, что в основной программе не подключался модуль p11009, концепция вложенности пространств имен PascalABC.NET позволяет увидеть его интерфейс при редактировании кода. Достаточно лишь набрать в коде основной программы «p11009» и поставить точку. Обратиться к показанным в интерфейсе функциям, безусловно, не получится.

11.4.2 Реализация на основе библиотеки

В целом, программный код может быть практически таким же, как и при использовании модулей, но реализация содержит лишь одну библиотеку. Основная программа имеет имя p11011, в библиотеке p11012 находятся описания пользовательских типов, функции для создания и инициализации переменных этих типов, а также подпрограммы, реализующие необходимые вычисления и вывод. Разбивать библиотеку на части нецелесообразно, да и сложно: любое логически оправданное разбиение библиотеки ведет к возникновению циклических ссылок. Результат работы программы идентичен приведенному в предыдущем разделе за исключением значений данных, поскольку они генерируются случайным образом.

```
// p11011

{$reference p11012.dll}

begin
  var КоличествоТреугольников := ReadInteger('Число треугольников =');
  var Треугольники := СгенерироватьТреугольники(КоличествоТреугольников);
  foreach var Треугольник in Треугольники do
    СведенияОТреугольнике(Треугольник);
  Print('Максимальная площадь тругольника');
  Треугольники.Select(Треугольник -> ПлощадьТреугольника(Треугольник))
      .Max.Println
end.
```

```
library p11012;

interface

type
  /// Точка на плоскости
  TPoint = record
    x: real;
    y: real
  end;

  /// Линия на плоскости
  TLine = record
    A: TPoint;
    B: TPoint
  end;

  /// Треугольник на плоскости
  TTriangle = record
    a: TLine;
    b: TLine;
    c: TLine
  end;

/// Возвращает точку с заданными координатами x и y
function СоздатьТочку(x, y: real): TPoint;
/// Возвращает линию, заданную парой точек Pa и Pb
function СоздатьЛинию(Pa, Pb: TPoint): TLine;
/// Возвращает треугольник, заданный сторонами a, b, c
function СоздатьТреугольник(a, b, c: TLine): TTriangle;

/// Возвращает длину линии
function ДлинаЛинии(L: TLine): real;
/// Возвращает площадь треугольника с указанными сторонами
function ПлощадьТреугольника(p: TTriangle): real;
/// Выводит координаты вершин треугольника
procedure ВыводКоординатВершин(p: TTriangle);
```

```
/// Выводит сведения о треугольнике с площадью S
procedure СведенияОТреугольнике(p: TTriangle; S: real := 0);
/// Возвращает массив n треугольников, сгенерированных случайным образом
function СгенерироватьТреугольники(n: integer): array of TTriangle;

implementation

function СоздатьТочку(x, y: real): TPoint;
begin
  (Result.x, Result.y) := (x, y);
end;

function СоздатьЛинию(Pa, Pb: TPoint): TLine;
begin
  (Result.A, Result.B) := (Pa, Pb);
end;

function СоздатьТреугольник(a, b, c: TLine): TTriangle;
begin
  (Result.a, Result.b, Result.c) := (a, b, c)
end;

function ДлинаЛинии(L: TLine) :=
Sqrt(Sqr(L.B.x - L.A.x) + Sqr(L.B.y - L.A.y));

function ПлощадьТреугольника(p: TTriangle): real;
begin
  var ДлинаAB := ДлинаЛинии(p.a);
  var ДлинаBC := ДлинаЛинии(p.b);
  var ДлинаAC := ДлинаЛинии(p.c);
  var Полупериметр := (ДлинаAB + ДлинаBC + ДлинаAC) / 2;
  Result := Sqrt(Полупериметр * (Полупериметр - ДлинаAB) *
      (Полупериметр - ДлинаBC) * (Полупериметр - ДлинаAC))
end;

procedure ВыводКоординатВершин(p: TTriangle);
begin
  Writeln(p.a.A, ', ', p.a.B, ', ', p.b.B)
end;

procedure СведенияОТреугольнике(p: TTriangle; S: real);
begin
  Print('Координаты вершин');
  ВыводКоординатВершин(p);
  Println('Длины сторон треугольника:',
      ДлинаЛинии(p.a), ДлинаЛинии(p.b), ДлинаЛинии(p.c));
  Print('Площадь треугольника:');
  if S = 0 then Println(ПлощадьТреугольника(p))
  else Println(S)
end;
```

```
function СгенерироватьТреугольники(n: integer): array of TTriangle;
begin
  Result := new TTriangle[n];
  for var i := 0 to n - 1 do
  begin
    var Координаты := SeqRandom(6, -100, 100)
        .Select(t -> Round(t / 10, 1)).ToArray;
    var Pa := СоздатьТочку(Координаты[0], Координаты[1]);
    var Pb := СоздатьТочку(Координаты[2], Координаты[4]);
    var Pc := СоздатьТочку(Координаты[4], Координаты[5]);
    var Lab := СоздатьЛинию(Pa, Pb);
    var Lbc := СоздатьЛинию(Pb, Pc);
    var Lac := СоздатьЛинию(Pa, Pc);
    Result[i] := СоздатьТреугольник(Lab, Lbc, Lac)
  end
end;

end.
```

11.4.3 Реализация на основе модуля и библиотеки

Эта реализация приведена в иллюстративных целях. На практике не следует без необходимости перемешивать модули с библиотеками, особенно когда имеются пользовательские типы данных. При наличии в программе и модулей, и библиотек, принимайте во внимание следующее:

- Планируя состав и структуру программы, определяйте пользовательский тип данных в одном месте и лучше, чтобы этим местом была библиотека;
- Модули просматриваются раньше библиотек, поэтому библиотеки следует предварительно откомпилировать;
- Не подключайте к основной программе модули, которые уже подключены к библиотекам.

```
// p11013

uses p11015;

begin
  var КоличествоТреугольников := ReadInteger('Число треугольников =');
  var Треугольники := СгенерироватьТреугольники(КоличествоТреугольников);
  foreach var Треугольник in Треугольники do
    СведенияОТреугольнике(Треугольник);
  Print('Максимальная площадь тругольника');
  Треугольники.Select(Треугольник -> ПлощадьТреугольника(Треугольник))
      .Max.Println
end.
```

```
library p11014;

interface

type
  /// Точка на плоскости
  TPoint = record
    x: real;
    y: real
  end;

  /// Линия на плоскости
  TLine = record
    A: TPoint;
    B: TPoint
  end;

  /// Треугольник на плоскости
  TTriangle = record
    a: TLine;
    b: TLine;
    c: TLine
  end;

/// Возвращает точку с заданными координатами x и y
function СоздатьТочку(x, y: real): TPoint;
/// Возвращает линию, заданную парой точек Pa и Pb
function СоздатьЛинию(Pa, Pb: TPoint): TLine;
/// Возвращает треугольник, заданный сторонами a, b, c
function СоздатьТреугольник(a, b, c: TLine): TTriangle;

implementation

function СоздатьТочку(x, y: real): TPoint;
begin
  (Result.x, Result.y) := (x, y);
end;

function СоздатьЛинию(Pa, Pb: TPoint): TLine;
begin
  (Result.A, Result.B) := (Pa, Pb);
end;

function СоздатьТреугольник(a, b, c: TLine): TTriangle;
begin
  (Result.a, Result.b, Result.c) := (a, b, c)
end;

end.
```

```
unit p11015;

{$reference p11014.dll}

interface

/// Возвращает длину линии
function ДлинаЛинии(L: TLine): real;
/// Возвращает площадь треугольника с указанными сторонами
function ПлощадьТреугольника(p: TTriangle): real;
/// Выводит координаты вершин треугольника
procedure ВыводКоординатВершин(p: TTriangle);
/// Выводит сведения о треугольнике с площадью S
procedure СведенияОТреугольнике(p: TTriangle; S: real := 0);
/// Возвращает массив n треугольников, сгенерированных случайным образом
function СгенерироватьТреугольники(n: integer): array of TTriangle;

implementation

function ДлинаЛинии(L: TLine) :=
Sqrt(Sqr(L.B.x - L.A.x) + Sqr(L.B.y - L.A.y));

function ПлощадьТреугольника(p: TTriangle): real;
begin
  var ДлинаAB := ДлинаЛинии(p.a);
  var ДлинаBC := ДлинаЛинии(p.b);
  var ДлинаAC := ДлинаЛинии(p.c);
  var Полупериметр := (ДлинаAB + ДлинаBC + ДлинаAC) / 2;
  Result := Sqrt(Полупериметр * (Полупериметр - ДлинаAB) *
      (Полупериметр - ДлинаBC) * (Полупериметр - ДлинаAC))
end;

procedure ВыводКоординатВершин(p: TTriangle);
begin
  Writeln(p.a.A, ', ', p.a.B, ', ', p.b.B)
end;

procedure СведенияОТреугольнике(p: TTriangle; S: real);
begin
  Print('Координаты вершин');
  ВыводКоординатВершин(p);
  Println('Длины сторон треугольника:',
      ДлинаЛинии(p.a), ДлинаЛинии(p.b), ДлинаЛинии(p.c));
  Print('Площадь треугольника:');
  if S = 0 then Println(ПлощадьТреугольника(p))
  else Println(S)
end;
```

```
function СгенерироватьТреугольники(n: integer): array of TTriangle;
begin
  Result := new TTriangle[n];
  for var i := 0 to n - 1 do
  begin
    var Координаты := SeqRandom(6, -100, 100)
        .Select(t -> Round(t / 10, 1)).ToArray;
    var Pa := СоздатьТочку(Координаты[0], Координаты[1]);
    var Pb := СоздатьТочку(Координаты[2], Координаты[4]);
    var Pc := СоздатьТочку(Координаты[4], Координаты[5]);
    var Lab := СоздатьЛинию(Pa, Pb);
    var Lbc := СоздатьЛинию(Pb, Pc);
    var Lac := СоздатьЛинию(Pa, Pc);
    Result[i] := СоздатьТреугольник(Lab, Lbc, Lac)
  end
end;

end.
```

11.5 Стандартные модули в PascalABC.NET

Система программирования на базе языка PascalABC.NET поставляется с набором стандартных модулей, большая часть которых реализует компьютерную графику. Часть этих модулей описана только в Справке, другие имеют отдельные описания и руководства. Объем и формат данной книги не предоставляют возможности подробно останавливаться на стандартных модулях, поэтому предлагаемые сведения неполны, носят общий описательный характер и в своем большинстве взяты из Справки PascalABC.NET.

11.5.1 Модули для работы с графикой

В Справке дается материал по следующим шести стандартным модулям: GraphABC, ABCObjects, GraphWPF, WPFObjects, Graph3D и ABCSprites.

11.5.1.1 Модуль GraphABC

Модуль GraphABC представляет собой простую графическую библиотеку и предназначен для создания несобытийных графических и анимационных программ в процедурном и частично в объектном стиле. Рисование осуществляется в специальном графическом окне, возможность рисования в нескольких окнах отсутствует. Кроме этого, в модуле GraphABC определены простейшие события мыши и клавиатуры, позволяющие создавать элементарные событийные приложения. Основная сфера использования модуля GraphABC - **обучение**.

Начиная с версии 3.3.0.1531 от 30.08.2017 модуль GraphABC объявлен устаревшим; его обновление и поддержка прекращены. Тем не менее, в последующие версии модуль GraphABC по-прежнему будет входить. Вместо GraphABC рекомендуется использовать модуль растровой графики GraphWPF.

11.5.1.2 Модуль GraphWPF

Модуль GraphWPF основан на графической библиотеке WPF и является современной и усовершенствованной версией устаревшего модуля GraphABC. Он представляет собой простую графическую библиотеку и предназначен для создания графических и анимационных программ в процедурном и частично в объектном стиле. Рисование осуществляется в специальном графическом окне, возможность рисования в нескольких окнах отсутствует. Кроме этого, в модуле GraphWPF определены простейшие события мыши и клавиатуры, позволяющие создавать элементарные событийные приложения. Основная сфера использования модуля GraphWPF - **обучение**.

Модуль GraphWPF введен в PascalABC.NET начиная с версии 3.3.0.1531 от 30.08.2017.

11.5.1.3 Модуль ABCObjects

Модуль ABCObjects создан на основе модуля GraphABC. Он реализует векторные графические объекты с возможностью масштабирования, наложения друг на друга, создания составных графических объектов и многократного их вложения друг в друга. Каждый векторный графический объект корректно себя перерисовывает при перемещении, изменении размеров и частичном перекрытии другими объектами. Модуль предназначен для раннего обучения основам объектно-ориентированного программирования, а также для реализации графических и анимационных проектов средней сложности.

11.5.1.4 Модуль WPFObjects

Модуль WPFObjects основан на графической библиотеке WPF и является современной и усовершенствованной версией устаревшего модуля ABCObjects. Он реализует векторные графические объекты с возможностью наложения друг на друга, создания составных графических объектов и многократного их вложения друг в друга. Каждый векторный графический объект корректно себя перерисовывает при перемещении, изменении размеров и частичном перекрытии другими объектами. Модуль предназначен для раннего обучения основам объектно-ориентированного программирования, а также для реализации анимационных и интерактивных проектов средней сложности.

Модуль WPFObjects введен в PascalABC.NET начиная с версии 3.4.2.1782 от 01.09.2018.

11.5.1.5 Модуль Graph3D

Модуль Graph3D основан на библиотеке Helix Toolkit WPF. Он реализует трехмерные графические объекты с возможностью масштабирования, группировки, поворотов, перемещения, создания дочерних объектов. Предназначен для обучения основам объектно-ориентированного программирования и основам 3D-графики.

Позволяет создавать простейшие трехмерные анимации и простейшие интерактивные трехмерные программы и игры.

Модуль Graph3D введен в PascalABC.NET начиная с версии 3.3.0.1531 от 30.08.2017. Для работы требует библиотеки Microsoft .NET 4.5. При работе под операционной системой Windows XP (SP2) возможно возникновение проблем, которые не могут быть устранены из-за прекращения поддержки Windows XP фирмой Microsoft.

11.5.1.6 Модуль ABCSprites

Модуль ABCSprites реализует спрайты - анимационные объекты с автоматически меняющимися кадрами. Спрайты автоматически анимируются в цикле, что управляется специальным таймером из модуля Timers.

11.5.2 Учебные модули

Учебные модули, как следует из их названия, предназначены для решения задач, связанных с обучением. Так, модули Робот и Чертежник, представляют собой реализацию Исполнителей, использующихся на уроках школьной информатики, а также использующихся в электронном задачнике РТ4 (автор – М. Э. Абрамян), входящем в комплект PascalABC.NET. Модуль NumLibABC реализует достаточно мощную библиотеку численных методов, которая может быть использована не только для обучения, но и для выполнения инженерных и научных расчетов.

11.5.2.1 Модуль Робот

Исполнитель Робот, описанный в учебнике А. Г. Кушниренко, Г. В. Лебедева и Я. Н. Зайдельмана «Информатика 7–9 классы», М., 2001, реализован в виде модуля Robot. Исполнитель действует на прямоугольном клеточном поле. Между некоторыми клетками, а также по периметру поля находятся стены. Основная цель Робота – закрасить указанные клетки и переместиться в конечную клетку.

11.5.2.2 Модуль Чертежник

Исполнитель Чертежник, описанный в учебнике А. Г. Кушниренко, Г. В. Лебедева и Я. Н. Зайдельмана «Информатика 7–9 классы», М., 2001, реализован в виде модуля Drawman. Исполнитель предназначен для построения рисунков и чертежей на плоскости с координатами. Чертежник имеет перо, которое он может поднимать, опускать и перемещать. При перемещении опущенного пера за ним остается след.

11.5.2.3 Модуль NumLibABC

Библиотека численных методов NumLibABC в настоящее время в Справке не представлена, но имеет самостоятельное справочное пособие формата PDF. Соответствующий файл с именем NumLibABC.pdf и находится в папке \PascalABC.NET\Doc. В справочном пособии кроме назначения и форматов вызова дается необходимая для понимания вопроса теория, а также приводятся примеры. Библиотека реали-

зована в виде набора классов, поэтому ее исходный текст может использоваться при изучении основ объектно–ориентированного программирования.

NumLibABC – свободно распространяемая библиотека, реализованная в системе программирования PascalABC.NET 3.5 и поставляющаяся вместе с ней, в том числе, в исходном коде. В пакете находятся программы, реализующие различные численные методы посредством классов, а также вспомогательные программы и сопутствующие типы данных.

С помощью пакета NumLibABC можно решать задачи из следующих областей:

- нахождение корней нелинейных уравнений;
- операции с простыми дробями;
- операции с полиномами;
- интерполяция, дифференцирование и аппроксимация данных, заданных в табличном виде;
- операции с векторами и матрицами, решение систем линейных уравнений;
- решение систем дифференциальных уравнений;
- вычисление определенных интегралов;
- задачи оптимизации.

Часть программ переведена в Паскаль на уровне исходного текста из существующих пакетов прикладных программ, таких как SSPLIB (на языке Фортран) или опубликованных в литературе. Другая часть написана на основе алгоритмов, приведенных в различных источниках. В любом случае, для поддержания чистоты открытой лицензии, приводятся ссылки на источник.

Модуль NumLibABC введен в PascalABC.NET начиная с версии 3.3.0.1552 от 20.10.2017.

11.6 Для самостоятельного решения

T11.1. Решения систем n уравнений с n неизвестными при n $\leqslant$ 4 может быть найдено по правилу Крамера. Для n=2 система уравнений имеет следующий вид:

$$\begin{cases} a_{1,1}x + a_{1,2}y = b_1 \\ a_{2,1}x + a_{2,2}x = b_2 \end{cases}$$

По правилу Крамера нужно сначала вычислить главный определитель системы уравнений $\Delta = a_{1,1} \cdot a_{2,2} - a_{2,1} \cdot a_{1,2}$

Если главный определитель системы равен нулю, то использовать правило Крамера нельзя: система уравнений имеет бесконечное множество решений, либо не имеет их вообще. Далее находятся определители по переменным:

$\Delta_x = b_1 \cdot a_{2,2} - b_2 \cdot a_{1,2}$, $\Delta_y = b_2 \cdot a_{1,1} - b_1 \cdot a_{2,1}$

И затем находится решение системы уравнений: $x = \Delta_x / \Delta$, $y = \Delta_y / \Delta$

Для n=3 решение несколько сложнее:

$$\begin{cases} a_{1,1}x + a_{1,2}y + a_{1,3}z = b_1 \\ a_{2,1}x + a_{2,2}y + a_{2,3}z = b_2 \\ a_{3,1}x + a_{3,2}y + a_{3,3}z = b_3 \end{cases}$$

$$\Delta = a_{11}(a_{22} \cdot a_{33} - a_{32} \cdot a_{23}) - a_{12}(a_{21} \cdot a_{33} - a_{31} \cdot a_{23}) + a_{1,3}(a_{21} \cdot a_{32} - a_{31} \cdot a_{22})$$

$$\Delta_x = b_1(a_{22} \cdot a_{33} - a_{32} \cdot a_{23}) - a_{12}(b_2 \cdot a_{33} - b_3 \cdot a_{23}) + a_{1,3}(b_2 \cdot a_{32} - b_3 \cdot a_{22})$$

$$\Delta_y = a_{11}(b_2 \cdot a_{33} - b_3 \cdot a_{23}) - b_1(a_{21} \cdot a_{33} - a_{31} \cdot a_{23}) + a_{1,3}(a_{21} \cdot b_3 - a_{31} \cdot b_2)$$

$$\Delta_z = a_{11}(a_{22} \cdot b_3 - a_{32} \cdot b_2) - a_{12}(a_{21} \cdot b_3 - a_{31} \cdot b_2) + b_1(a_{21} \cdot a_{32} - a_{31} \cdot a_{22})$$

$$x = \frac{\Delta_x}{\Delta},\ y = \frac{\Delta_y}{\Delta},\ z = \frac{\Delta_z}{\Delta}$$

Напишите модуль Kramer, реализующий функцию Solve, возвращающую массив с решениями системы уравнений. В качестве параметров процедуры используйте двумерный массив a с коэффициентами a_{ij} и одномерный массив b со свободными членами уравнений b_i. С помощью написанного модуля найдите решения систем уравнений

$$\begin{cases} 6x - 5y = 23 \\ x + 3y = 8 \end{cases} \qquad \begin{cases} 3x - 2y + 5z = 7 \\ 7x + 4y - 8z = 3 \\ 5x - 3y - 4z = -12 \end{cases}$$

Часть 12

Обработка ошибок в программе

Для меня долгое время было загадкой, как что-то очень дорогое и технологичное может быть столь бесполезным. И вскоре я осознал, что компьютер — это глупая машина, обладающая способностями выполнять невероятно умные вещи, тогда как программисты — это умные люди, у которых талант делать невероятные глупости. Короче, они нашли друг друга.

Билл Брайсон, американский писатель

Тестирование программы может весьма эффективно продемонстрировать наличие ошибок, но безнадежно неадекватно для демонстрации их отсутствия.

Эдсгер Дейкстра

Никогда не выявляйте в программе ошибки, если не знаете, что с ними дальше делать.

Артур Блох.
Законы Мерфи на каждый день

Ошибки, увы, сопровождают любую деятельность человека. Не зря говорится, что не ошибается лишь тот, кто ничего не делает. И программисты здесь не являются приятным исключением. Поскольку ошибки неизбежны, наиболее разумной стратегией будет своевременная подготовка к их обработке.

Среди ошибок в программе можно выделить три категории:

- синтаксические, связанные с некорректным написанием кода;
- логические, вызванные неверным построением алгоритма или неверным пониманием принципов работы языковых конструкций;
- ошибки времени выполнения (run-time error).

Синтаксические ошибки выявляются компилятором. О наличии логических ошибок можно судить по неверному функционированию интерфейса программы или неверным результатам ее работы. Ошибки времени выполнения, также называемые ***исключениями*** (exceptions), возникают во время выполнения программы. По умолчанию исключения обрабатываются исполняющей средой и обычно сопровождаются выдачей диагностических сообщений с последующим завершением программы. Исключения могут возникать при вводе пользователем неправильных данных, делении на ноль, переполнении разрядной сетки, выходе индекса массива за границу, отсутствии ресурса, к которому обращается программа и т.д. PascalABC.NET предоставляет программисту средства, позволяющие отслеживать возникновение исключений и обрабатывать их с тем, чтобы не допускать завершения программы с ошибкой.

12.1 Предотвращение исключений

Полностью предотвратить возможность возникновения исключений нереально – это равнозначно написанию программы, не содержащей ни одной ошибки. Но вполне возможно предусмотреть контроль данных в программе с тем, чтобы не создавались условия для возникновения исключений по конкретным причинам. Например, если проверить значение делителя перед операцией деления, исключение из-за деления на ноль не возникнет. Если перед обращением к файлу проверить его наличие, не возникнет исключения по причине отсутствия файла. Для организации подобных проверок имеются несколько средств и приемов программирования.

Пусть требуется вычислить значение математического выражения

$$\min(r,s),\ \text{где}\ r=\frac{\sqrt{2b-5a}-\sqrt{4a-c}}{\sqrt{3a-b}}, s=\sqrt{\frac{2r-5}{r-4}}$$

Анализ формул показывает, что значения a, b и c должны находиться в определенных областях, таких чтобы под корнями не появлялось отрицательных значений и не происходило деление на ноль. Если допустить, что данные вводятся пользователем с клавиатуры, то в некоторых случаях могут возникнуть проблемы при вычислениях. Давайте убедимся в этом при помощи простой программы.

```
// p12001
begin
  var (a, b, c) := ReadReal3;
  var r := (Sqrt(2 * b - 5 * a) - Sqrt(4 * a - c)) / Sqrt(3 * a - b);
  var s := Sqrt((2 * r - 5) / (r - 4));
  Println(Min(r, s))
end.
```

Выполним программу для набора данных *a*=4, *b*=11, *c*=15, а затем для *a*=4, *b*=11, *c*=17.

```
4 11 15
0.414213562373095

4 11 17
NaN
```

Со вторым набором данных получено значение NaN (Not a Number – не число). Это сигнал о том, что такой набор данных неприемлем. Но чем именно он неприемлем, в каком месте возникла ошибка? Появилось отрицательное число под одним из корней или имело место деление на ноль? На все эти вопросы придется отвечать. А если в программе сотни операторов – как тогда искать ошибку?

12.1.1 Подмена стандартных функций

Первая идея, которая приходит в голову – написать собственную функцию для вычисления квадратного корня, где анализировать аргумент на допустимость. Это не укажет точное место возникновения проблемы, но позволит хоть что-то прояснить.

```
function Sqrt(x: real): real; // p12002
begin
  if x < 0 then Writeln('Sqrt: отрицательный аргумент ', x);
  Result := System.Math.Sqrt(x)
end;

begin
  var (a, b, c) := ReadReal3;
  var r := (Sqrt(2 * b - 5 * a) - Sqrt(4 * a - c)) / Sqrt(3 * a - b);
  var s := Sqrt((2 * r - 5) / (r - 4));
  Println(Min(r, s))
end.
```

Выполним эту программу для набора данных a=4, b=11, c=17, который привел к появлению NaN в результате.

```
4 11 17
Sqrt: отрицательный аргумент -1
NaN
```

Стало чуть лучше. Но лишь чуть: теперь мы точно знаем причину, но не знаем места возникновения ошибки. А вот еще один набор данных, приводящий к появлению NaN:

```
3 9 5
NaN
```

Похоже, тут имело место деление на ноль... Попробуем сделать диагностику и такой причины. Будем делить при помощи собственной функции Divide.

```
function Sqrt(x: real): real; // p12003
begin
  if x < 0 then Writeln('Sqrt: отрицательный аргумент ', x);
  Result := System.Math.Sqrt(x)
end;

function Divide(a, b: real): real;
begin
  if Abs(b) <= 1e-17 then Writeln('Деление на ноль');
  Result := a / b
end;

begin
  var (a, b, c) := ReadReal3;
  var r := Divide(Sqrt(2 * b - 5 * a) - Sqrt(4 * a - c), Sqrt(3 * a - b));
  var s := Sqrt(Divide(2 * r - 5, r - 4));
  Println(Min(r, s))
end.
```

Проверим работу программы с наборами данных *a*=4, *b*=11, *c*=17 и *a*=3, *b*=9, *c*=5:

```
4 11 17
Sqrt: отрицательный аргумент -1
Деление на ноль
NaN

3 9 5
Деление на ноль
NaN
```

Оказывается, деление на ноль возникает в обоих случаях, но все же: где именно? Если мы отдадим пользователю такую программу, он вряд ли обрадуется подобным сообщениям.

12.1.2 Системная процедура Assert

Для целей отладки может оказаться полезной системная процедура Assert, позволяющая установить причину и место возникновения ошибки.

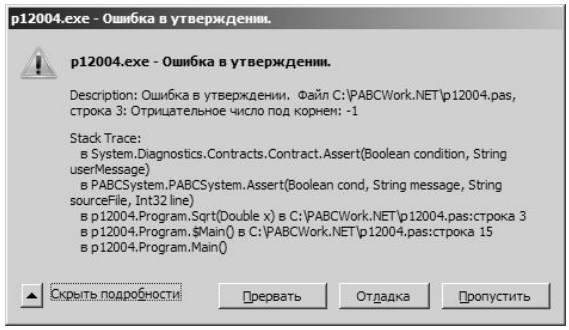
```
procedure Assert(условие);
procedure Assert(условие; текст_сообщения);
```

Если указанное «условие» не выполняется (значением логического выражения, понимаемого под «условием» будет False), то в специальном окне будет выведено диагностическое сообщение и стек вызовов подпрограмм. Также будет выведен «текст_сообщения», если он присутствует в вызове.

Модифицируем процедуру p12003, заменив диагностирующие сообщения на обращения к Assert.

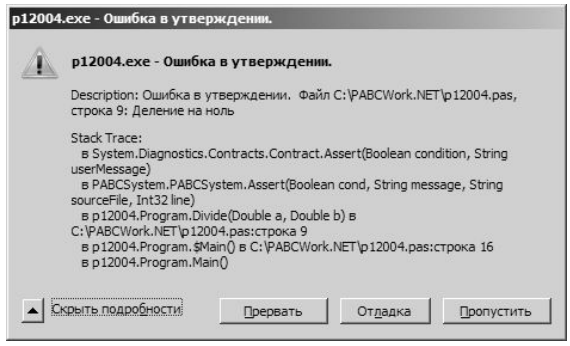
```
function Sqrt(x: real): real;// p12004
begin
  Assert(x >= 0, 'Отрицательное число под корнем: ' + x.ToString);
  Result := System.Math.Sqrt(x)
end;

function Divide(a, b: real): real;
begin
  Assert(Abs(b) > 1e-17, 'Деление на ноль');
  Result := a / b
end;

begin
  var (a, b, c) := ReadReal3;
  var r := Divide(Sqrt(2 * b - 5 * a) - Sqrt(4 * a - c), Sqrt(3 * a - b));
  var s := Sqrt(Divide(2 * r - 5, r - 4));
  Println(Min(r, s))
end.
```

Проверим работу программы с набором данных *a*=4, *b*=11, *c*=17. После ввода данных появляется диагностическое окно Assert.



Зафиксирована ошибка в файле p12004.pas. Утверждение Assert в строке 3 кода сообщило, что под корнем получилось отрицательное число -1. Просмотр стека вызовов (списка модулей в порядке, обратном их вызову) в последней строке показывает, что ошибка произошла в модуле p12004.pas на строчке 15. Это оператор, вычисляющий значение переменной r.

Для продолжения работы с программой выбираем кнопку «Отладка». Всплывает еще одно окно Assert. На этот раз произошло деление на ноль. И мы определяем по стеку вызовов, что это произошло в файле p12004.pas на строчке 16, где вычисляется значение переменной s. Снова выбираем кнопку «Отладка» и программа завершается, выводя результат




```
4  11  17
NaN
```

Конечно, такое средство как Assert, не годится для встраивания в приложения, которые делаются для пользователей. Но при отладке или поиске ошибок в программе оно может оказаться достаточно удобным.

С помощью Assert можно получить детальный отчет о ходе выполнения программы. Для этого придется модифицировать ее код, обращаясь к Assert во всех «подозрительных» местах.

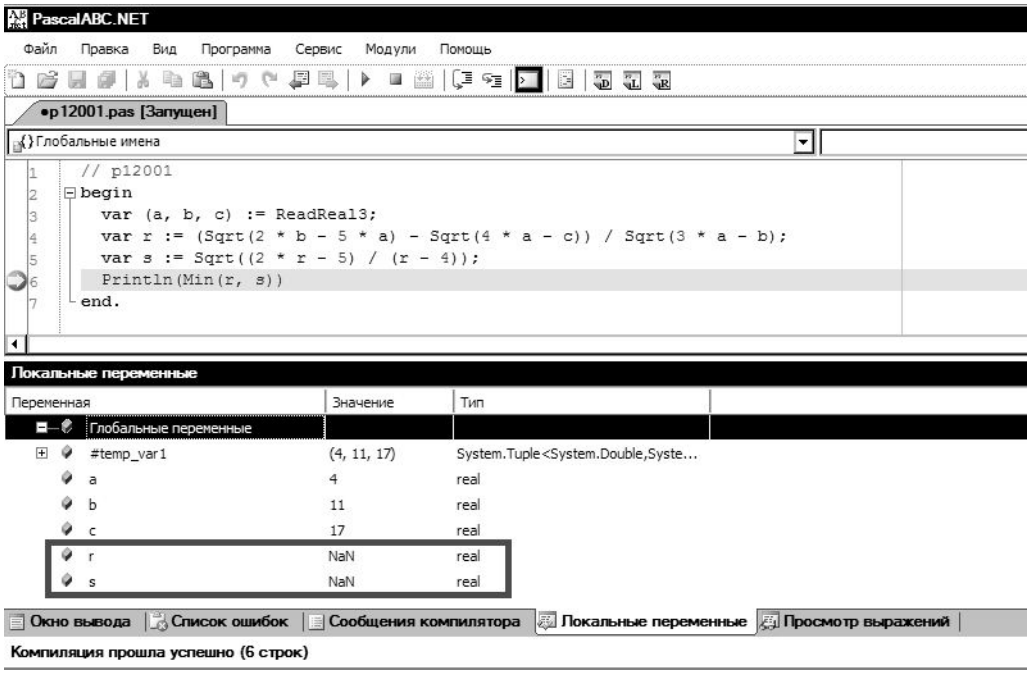
```
begin // p12005
  var (a, b, c) := ReadReal3;
  Assert(2 * b - 5 * a >= 0, 'r: Отрицательное значение под первым
корнем');
  Assert(4 * a - c >= 0, 'r: Отрицательное значение под вторым корнем');
  Assert(3 * b - b >= 0, 'r: Отрицательное значение под третьим корнем');
  Assert(Abs(3 * b - b) > 1e-17, 'r: деление на ноль');
  var r := (Sqrt(2 * b - 5 * a) - Sqrt(4 * a - c)) / Sqrt(3 * a - b);
  Assert(Abs(r - 4) > 1e-17, 's: деление на ноль');
  Assert((2 * r - 5) / (r - 4) >= 0, 's: Отрицательное значение под корнем');
  var s := Sqrt((2 * r - 5) / (r - 4));
  Println(Min(r, s))
end.
```

Запуск этой программы для набора данных a=4, b=11, c=17 приведет к выдаче сообщений

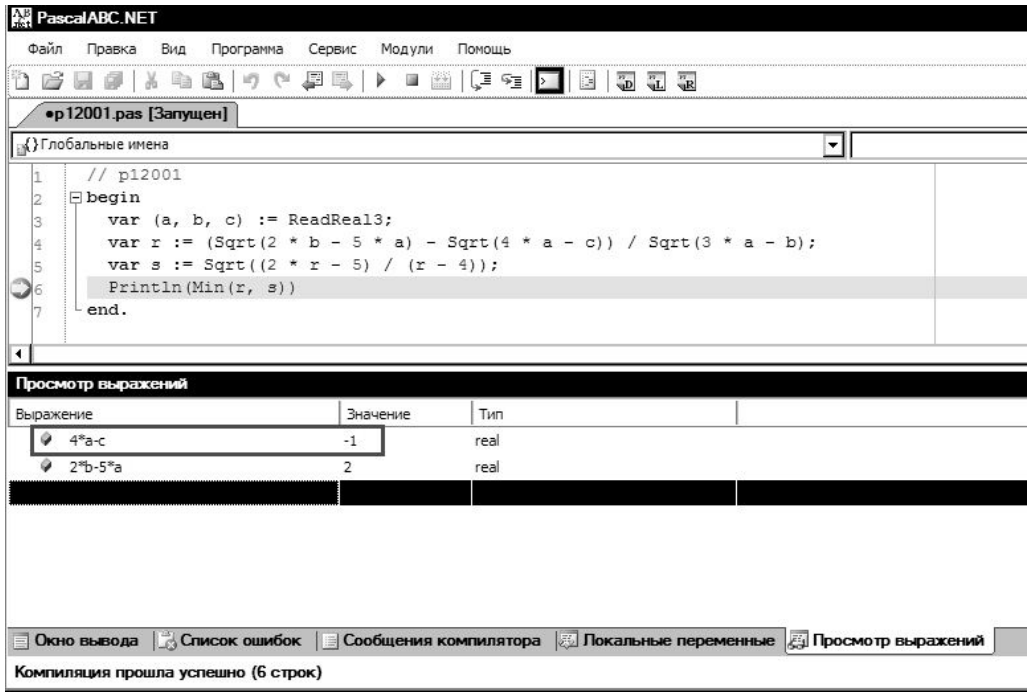
```
r: Отрицательное значение под вторым корнем
s: деление на ноль
```

12.1.3 Использование IDE-отладчика

Интегрированная среда обработки (IDE) PascalABC.NET позволяет вести отладку выполняющейся программы в реальном времени. Такой способ поиска причины и места ошибки может использоваться только программистом. Он требует понимания структуры отлаживаемой программы и логики выполнения ее фрагментов. Можно указать точки прерывания (строки исходной программы, перед выполнением которых будет сделана остановка), в которых затем просматривать значения переменных, проделывать некоторые вычисления и продолжать выполнение программы до следующей точки прерывания или построчно.



В приведенном примере показана остановка на строке 6 программы p12001. В окне «Локальные переменные» видно, что значения r и s представлены как NaN. Можно перейти в окно «Просмотр выражений» и проделать там вычисления выражений, которые при вычислении r и s могли привести к такому результату. Это подкоренные выражения, которые не должны быть отрицательными и делители, которые не должны равняться нулю. Поскольку s зависит от r, достаточно найти место проблемы, возникшей при вычислении r. Будем поочередно вводить подкоренные выражения. Во втором выражении получено значение -1, которое и является местом возникновения проблемы. На этом отладку можно завершить.



С точки зрения PascalABC.NET в программе ошибок не возникало. В арифметике вещественных чисел .NET значение NaN используется с целью не допускать возникновения исключений из-за некорректных значений аргументов математических функций. Другое дело, что такое значение не устраивает пользователей. К сожалению, обратной стороной этого «удобства» является невозможность использовать самые мощные средства отладки и подавления ошибок, имеющиеся в PascalABC.NET, поскольку они основаны на обработке возникающих исключений.

12.2 Обработка исключений

Что произойдет в программе p12001 (см. 12.1.1), если пользователь введет некорректные данные? Например, укажет в числе запятую вместо точки. Возникнет исключение, стандартной реакцией на которое будет выдача сообщения об ошибке с последующим завершением программы:

```
p12001.pas(2): Ошибка времени выполнения: Входная строка имела неверный формат.
```

Как бороться с таким неправильным вводом? Существуют несколько способов. Например, можно принять данные как строку, проанализировать ее на корректность и затем либо превратить в числовые данные, либо выдать свое сообщение об ошибке и запросить повторный ввод. PascalABC.NET также предлагает универсальное решение, основанное на перехвате возникшего исключения и его последующей обработке программным путем.

12.2.1 Виды исключений

Всякий раз, когда возникает ошибка, система реагирует на нее путем генерации исключения. Имеется заранее определенный набор ***стандартных*** исключений. Программист может также определить собственные ***пользовательские*** исключения. Возникшее исключение может быть перехвачено и обработано программным путем. Если этого не сделать, программа будет завершена с ошибкой. Для перехвата и обработки исключения в PascalABC.NET используется оператор **try** … **except**.

12.2.2 Оператор try … except

Оператор имеет следующий формат:

```
try
   защищаемый блок
except
   блок обработки исключений
end;
```

Защищаемый блок (далее – также блок try) – это набор операторов, для которых будет перехвачено любое возникшее исключение. Блок обработки исключений (далее – также блок except) определяет, как именно следует обработать перехваченное исключение. Если исключение будет обработано, то следующим выполняется оператор, указанный непосредственно после **try** … **except**. Это понятно: работа оператора завершена, далее выполняется следующий за ним. В случае, когда возникшее исключение не обработано (например, его обработка не была предусмотрена в блоке except), программа будет завершена с ошибкой. Если исключение не возникает, блок except просто игнорируется.

Возможен вариант, когда при обработке исключения в блоке except возникнет еще одно исключение. В этом случае выполнение операторов блока except будет прервано, а исключение, которое обрабатывалось этим блоком, будет считаться необработанным.

Операторы **try** … **except** могут вкладываться друг в друга. В таких случаях при возникновении исключения в блоках except управление будет передано объемлющему (внешнему) оператору **try** … **except**.

Блок except может содержать либо последовательность операторов, разделенных точкой с запятой, либо последовательность обработчиков исключений, каждый из которых может иметь вид

```
on тип do оператор;
on имя: тип do оператор;
```

Здесь «тип» - тип исключения, производный от стандартного класса Exception, а «имя» используют для доступа к свойствам .Message и .StackTrace (см. 12.2.3). Для стандартных исключений типы известны; они являются потомками класса .NET System.SystemException и могут быть найдены в соответствующей документации. В Справке приводится часть этих типов.

- System.OutOfMemoryException - недостаточно памяти для выполнения программы;
- System.StackOverflowException - переполнение стека (как правило, при многократных вложенных вызовах подпрограмм);
- System.AccessViolationException - попытка доступа к защищенной памяти;
- System.ArgumentException - неверное значение параметра подпрограммы;
- System.ArithmeticException - базовый класс всех арифметических исключений. Его наследники:
  - System.DivideByZeroException - целочисленное деление на 0;
  - System.OverflowException - переполнение при выполнении арифметической операции или преобразования типов;
- System.FormatException - неверный формат параметра (например, при преобразовании строки в число);
- System.IndexOutOfRangeException - выход за границы диапазона изменения индекса массива;
- System.InvalidCastException - неверное приведение типов;
- System.NullReferenceException - попытка вызвать метод для нулевого объекта или разыменовать нулевой указатель;
- System.IO.IOException - ошибка ввода-вывода. Наследники:
  - System.IO.IOException.DirectoryNotFoundException - каталог не найден;
  - System.IO.IOException.EndOfStreamException - попытка чтения за концом потока;
  - System.IO.IOException.FileNotFoundException - файл не найден.

В 12.1.2 упоминалась проблема, связанная с некорректным вводом данных. Теперь можно предложить ее решение.

```
begin // p12006
  var a, b, c: real;
  var ВводУспешен: boolean;
  repeat
    try
      (a, b, c) := ReadReal3;
      ВводУспешен := True;
    except
      on System.FormatException do
      begin
        Writeln('Ошибка ввода, повторите');
        ВводУспешен := False
      end
    end
  until ВводУспешен;
  var r := (Sqrt(2 * b - 5 * a) - Sqrt(4 * a - c)) / Sqrt(3 * a - b);
  var s := Sqrt((2 * r - 5) / (r - 4));
  Println(Min(r, s))
end.
```

В цикле **repeat** ... **until** осуществляется ввод данных до тех пор, пока значение переменной ВводУспешен не станет истинным. В случае, если при вводе не возникнет ошибки, будут выполнены все операторы блока try. При возникновении

исключения из-за ошибки ввода произойдет переход к выполнению блок except. Ошибке ввода соответствует тип System.FormatException, поэтому будет выведено информационное сообщение, а переменная ВводУспешен получит значение False. После завершения работы оператора **try** … **except** будет проверено условие завершения цикла.

В блоке except для каждого исключения может быть указан свой обработчик **on**. Типы исключений просматриваются в порядке их записи и выбирается нужный. Обработанное исключение при необходимости можно снова возбудить при помощи оператора **raise** (см. 12.2.3.2). Если ни один обработчик не подойдет, исключение будет считаться необработанным.

Все исключения, для которых не указаны обработчики, можно обработать в ветке **else** блока except. Если такая ветка присутствует, она указывается в последовательности обработчиков последней.

В приведенном выше примере рассмотрен случай обработки исключения единственного класса, связанного с некорректным преобразованием формата. В случае возникновения исключения иного класса, оно не будет обработано. Если считать, что иных исключений не может возникнуть, программу можно упростить.

```
begin // p12007
  var a, b, c: real;
  var ВводУспешен: boolean;
  repeat
    try
      (a, b, c) := ReadReal3;
      ВводУспешен := True;
    except
      Writeln('Ошибка ввода, повторите');
      ВводУспешен := False
    end
  until ВводУспешен;
  var r := (Sqrt(2 * b - 5 * a) - Sqrt(4 * a - c)) / Sqrt(3 * a - b);
  var s := Sqrt((2 * r - 5) / (r - 4));
  Println(Min(r, s))
end.
```

Такой упрощенный подход не требует знаний о том, какое именно исключение было сгенерировано.

12.2.3 Поиск причины и места возникновения ошибки

В 12.1.1.3 был рассмотрен способ локализации ошибки при помощи IDE-отладчика. Там же отмечено, что такой способ непригоден для конечного пользователя. На компьютере вообще может отсутствовать PascalABC.NET. В подобных случаях может помочь выдача отладочной информации в log-файл. Под log-файлами понимают обычные текстовые файлы, в которые программным путем помещается

информация о ходе выполнения программы. Рассмотрим пример программы, помещающей протокол работы в log-файл.

```
begin // p12008
  try
    var lf := OpenWrite('MyLog.txt');
    try
      var n := ReadInteger('Введите n:');
      var (a, b, c) := ReadReal3('Введите a, b, c:');
      Println('Результат', Trunc(a * b * c) div n);
    except
      on error: System.Exception do
      begin
        var s1 := error.StackTrace;
        var s2 := error.Message;
        lf.Writeln(s2);
        lf.Writeln(s1);
        $'Строка {s1.ToArray.Last}: {s2}'.Println
      end
    end;
    lf.Close;
    Println('Программа завершена')
  except
    Writeln('Не могу создать лог-файл');
  end
end.
```

При ошибках пользователя исключение может возникнуть вследствие одной из трех причин: ввод некорректного значения, невозможности преобразовать сумму трех величин типа **real** к типу **integer** и целочисленное деление на ноль. При генерации любого исключения записываем диагностику в log-файл и выводим сообщение для пользователя. Ниже приводятся примеры работы программы.

```
Введите n: 2.4
Строка 7: Входная строка имела неверный формат.
Программа завершена
```

```
Введите n: 5
Введите a, b, c: 1.5 73 2,18
Строка 7: Входная строка имела неверный формат.
Программа завершена
```

```
Введите n: 0
Введите a, b, c: 13 26.3 -1.3e-2
Строка 9: Попытка деления на нуль.
Программа завершена
```

```
Введите n: 37
Введите a, b, c: 1e12 2e19 3.5e+7
Строка 9: Значение было недопустимо малым или недопустимо большим для Int32.
Программа завершена
```

В программе использован тип Exception – от него наследуются все остальные типы, связанные с исключениями, поэтому он является универсальным обработчиком.

Свойство Message хранит стандартное сообщение об ошибке, свойство StackTrace – стек вызовов.

Чтобы зафиксировать возникшее исключение, а затем сбросить его, сохранив возможность обратиться к деталям этого исключения в дальнейшем, можно сохранить исключение во внешней по отношению к оператору **try** ... **except** переменной. Тип переменной для сохранения исключения должен быть System.Exception или производным от него.

```
begin // p12009
  var err1, err2: System.Exception;
  try
    var i := 0;
    Println(10 div i);
  except
    on err: System.Exception do
    begin
      'Перехват исключения в Try1'.Println;
      err1 := err
    end
  end;
  Println('Продолжение работы');
  try
    var i := 0;
    Println(10 div i)
  except
    on err: System.Exception do
    begin
      Println('Перехват исключения в Try2');
      err2 := err
    end;
  end;
  Println('Возникали исключения:');
  err1.StackTrace.Println;
  err1.Message.Println;
  err2.StackTrace.Println;
  err2.Message.Println
end.
```

В результате работы программы будет получен следующий вывод:

```
Перехват исключения в Try1
Продолжение работы
Перехват исключения в Try2
Возникали исключения:
   в Program1.Program.$Main() в C:\PABCWork.NET\P12009.pas:строка 5
Попытка деления на нуль.
   в Program1.Program.$Main() в C:\PABCWork.NET\P12009.pas:строка 16
Попытка деления на нуль.
```

Можно еще отметить, что имя программы, в которой было сгенерировано исключение (в данном случае p12009) находится в свойстве .Source возбужденного исключения.

12.2.3.1 Оператор try … finally

В определенных ситуациях может возникать потребность выполнить некоторые действия независимо от того, возникает исключение, или нет. Для таких случаев можно использовать оператор

```
try
  операторы
finally
  операторы
end;
```

Независимо от того, возникло ли исключение в блоке операторов, расположенном между ключевыми словами **try** и **finally** (далее – блок try), будут выполняться операторы из блока, находящегося между ключевыми словами **finally** и **end** (далее – блок finally). После выполнения блока finally исключение, если оно возникло, считается необработанным.

Зачем нам в программе необработанное исключение? Конечно для того, чтобы его обработать своими средствами! Оставить исключение необработанным - намеренно устроить аварийное завершение программы и дать пользователю лишнюю возможность небезосновательно предположить, откуда растут ваши руки. Мы помним, что исключение обрабатывается в операторе **try** … **except**. И теперь вам должно стать понятно, для чего «пряморукие» программисты оператор **try** … **finally** всегда оборачивают в оператор **try** … **except**.

Но если вы пожелаете отмахнуться от возникшего исключения, как от назойливой мухи, попросту сбросив его, это можно сделать как с помощью **try** … **finally**, так и с помощью **try** … **except**.

```
try
  операторы1
finally
  операторы2
end;
```

```
try
  операторы1
except
end;
операторы2;
```

Приводимый ниже пример взят из справочника Micrisoft по C# для оператора try – finally, слегка модифицирован и портирован в PascalABC.NET. В блоке **try** … **except** основной программы происходит вызов процедуры TryCast. Эта процедура возбуждает исключение вследствие попытки привести строку s с недопустимыми в записи числа символами к переменной i типа **integer**. Используется промежуточная переменная obj типа **object** с тем, чтобы ошибка не выявилась на стадии компиляции. После возбуждения исключения выполнение операторов в блоке try прерывается, выполняется блок finally и управление передается блоку except основной программы. Возникшее исключение запоминается в переменной ex, после чего выполняется блок в секции **on**.

```
procedure TryCast; // p12010
begin
  var i := 123;
  var s := 'Некоторая строка';
  var obj: object := s;
  try
    i := integer(obj);
    Println('Строка в конце блока try');
  finally
    Println($'В блоке finally процедуры TryCast i = {i}');
  end
end;


begin
  try
    TryCast
  except
    on ex: System.Exception do
    begin
      $'Перехват исключения {ex.GetType()} после блока finally'.Println;
      ex.Message.Println;
      ex.StackTrace.Println
    end
  end
end.
```

Программа выводит следующие строки:

```
В блоке finally процедуры TryCast i = 123
Перехват исключения System.InvalidCastException после блока finally
Заданное приведение является недопустимым.
   в p12010.Program.TryCast() в C:\PABCWork.NET\p12010.pas:строка 7
   в p12010.Program.$Main() в C:\PABCWork.NET\p12010.pas:строка 16
```

В Справке приводятся фрагменты программ, демонстрирующие применение оператора **try** ... **finally**. Так, при записи в файл, аварийное завершение программы может привести к утере части информации, находящейся в буфере. Оператор **try** ... **finally** позволяет гарантированно закрыть файл, обеспечив сохранение данных.

12.2.3.2 Оператор raise

Оператор **raise** служит для возбуждения исключения. Он имеет два формата:

```
raise;
raise объект;
```

Здесь «объект» – объект класса System.Exception или производный от него. Обычно программист определяет собственные типы исключений, для чего нужно создать класс, являющийся наследником класса System.Exception. Как создавать такие классы, рассказывается в следующей части (см. 13.11). Также можно указать имя переменной из секции on, которое является именем объекта, связанного с обрабатываемым исключением.

Оператор **raise** без параметров можно использовать только внутри блока **except**. Он служит для повторного возбуждения обрабатываемого исключения.

Возбужденное посредством оператора **raise** исключение в свойстве .StackTrace в некоторых случаях возвращает номер строки программы со следующим выполняемым оператором. Не удивляйтесь – это известная программистскому сообществу проблема, связанная с некорректностью в файлах .pdb, создаваемых компиляторами в .NET-среде для целей отладки.

В программе p12011 показано применение оператора **raise** в целях тестирования. В соответствии с введенным значением возбуждается определенное исключение, которое затем обрабатывается и возбуждается повторно. Используются вложенные операторы **try** ... **except**.

```
begin // p12011
  var er: System.Exception;
  try
    try
      Println('Укажите тип возбуждаемого исключения');
      case ReadInteger('0 - деление, 1 - индекс') of
        0: raise new System.DivideByZeroException;
        1: raise new System.IndexOutOfRangeException;
        else Println('Введено недопустимое значение')
      end;
      Println('Завершение try')
    except
      on e1: System.DivideByZeroException do
      begin
        Print('Обработчик деления на ноль в строке');
        e1.StackTrace.ToArray.Last.Println;
        er := e1;
        raise;
      end;
      on e2: System.IndexOutOfRangeException do
      begin
        Print('Обработчик выхода индекса за границу в строке');
        e2.StackTrace.ToArray.Last.Println;
        er := e2;
        raise er;
      end
    end;
  except
    on System.Exception do
    begin
      Println(er.GetType);
      Println(er.StackTrace)
    end
  end
end.
```

Как можно убедиться из приводимых протоколов работы, программа работает правильно. В некоторых случаях имела место описанная некорректность с выдачей номера строки в **raise**.

```
Укажите тип возбуждаемого исключения
0 - деление, 1 - индекс 0
Обработчик деления на ноль в строке 8
System.DivideByZeroException
   в p12011.Program.$Main() в C:\PABCWork.NET\p12011.pas:строка 18

Укажите тип возбуждаемого исключения
0 - деление, 1 - индекс 1
Обработчик выхода индекса за границу в строке 9
System.IndexOutOfRangeException
   в p12011.Program.$Main() в C:\PABCWork.NET\p12011.pas:строка 25

Укажите тип возбуждаемого исключения
0 - деление, 1 - индекс 2
Введено недопустимое значение
Завершение try
```

Часть 13

Объектно-ориентированное программирование

У меня есть кошка по имени Трэш. [...] если бы я пытался продать ее (по крайней мере, специалисту по компьютерам), я бы [...] сказал, что она объектно-ориентированная.

Р. Кинг, исполнительный директор CBS

ООП предоставляет вам множество способов замедлить работу ваших программ

Патрик Киллелиа,
автор книги «Тюнинг веб-сервера»

Я придумал термин «объектно-ориентированный», и я уверяю вас, что не имел в виду C++

Алан Кей, создатель ООП

История объектно-ориентированного программирования (далее – ООП) насчитывает уже полвека. Сам термин «объектно-ориентированное программирование» был введен Аланом Кеем в 1972 году для языка Smalltalk-72 (его он и имел в виду в своей известной цитате, вынесенной в заголовок). И все же, до сих пор нет общепринятого единого взгляда на эту технологию программирования. Примерно с 80-х годов термин «объектно-ориентированный» настолько вошел в моду, что его стали навешивать на любые, вновь разрабатываемые языки, чтобы успешно продвигать их. В результате принципы ООП, заложенные в ранних языках стали подвергаться различного рода искажениям и дополнениям, что и породило нынешние расхождения во взглядах.

В парадигме структурного программирования – предшественнице ООП – внимание сосредоточено на разработке алгоритма. Первоосновой являются подпрограммы, которые передают друг другу данные, т.е. данные являются вторичными. В объектно-ориентированной парадигме подпрограммы (они называются ***методами***) объединяются с данными в единую структуру – ***класс***. Классы выстраиваются в определенную иерархию. На базе классов создаются ***объекты***, между которыми и происходит взаимодействие. Смена парадигмы на практике означает смену характера взаимодействия: вместо отношений «подпрограмма – подпрограмма» возникает отношение «объект – объект».

13.1 Базовые понятия

Как и любая технология программирования, ООП использует определенный набор понятий и именующих их терминов. К сожалению, в научном мире существует достаточно много расхождений в терминологии, когда речь идет об общих концепциях. В период становления терминов свою лепту внесли также переводчики иностранной литературы. В результате чрезвычайно сложно найти две книги разных авторов, в которых используется единая терминология, пока речь не пойдет о конкретном языке программирования. Чтобы не усугублять ситуацию, здесь

преимущественно будет использоваться терминология, принятая в книге Мэтта Вайсфельда (Matt Weisfeld) «Объектно-ориентированное мышление». – СПб.; Питер, 2014. До тех пор, пока речь не пойдет о PascalABC.NET.

13.1.1 Классы и объекты

Объекты реального мира характеризуются своими данными и поведением. Например, данными объекта Автомобиль могут быть марка, цвет, год выпуска, госномер, сведения о владельце. Поведение автомобиля – это его способность разгоняться, передвигаться, менять скорость, поворачивать, тормозить. С этой точки зрения объект – это некоторая сущность, в которой одновременно имеются данные и поведение.

В терминологии ООП данные называют ***атрибутами***, а поведение объектов содержится в ***методах***. Объекты представляют собой «кирпичики», из которых строится программа. Они взаимодействуют друг с другом, посылая ***сообщения***, обращающиеся к тем или иным методам.

Общение объектов друг с другом потребовало наличия у объекта двух видов поведения: сообщения значений своих атрибутов и изменения значений атрибутов, в соответствии с полученными сообщениями. С этой целью каждый атрибут может снабжаться собственными методами для сообщения значения или установки значения. Исторически сложилось так, что операцию получения значения в программировании принято называть английским словом get (получить), а операцию изменения значения называют set (установить). Соответственно, процедуры получения и установки значений атрибутов часто называют ***геттерами (getter)*** и ***сеттерами (setter)***. Вы можете встретиться с этими терминами в различной литературе, а также при общении с программистами. Геттеры и сеттеры поддерживают одну из важных концепций ООП – сокрытие данных в объекте. Они не позволяют другим объектам напрямую манипулировать скрытыми данными. В настоящее время по установившейся терминологии атрибуты, имеющие сеттеры и/или геттеры, называют просто ***свойствами объекта***.

Некоторые атрибуты объекта могут быть доступны сообщениям напрямую. Например, цвет и госномер автомобиля видны на расстоянии, а злоумышленник может ночью самостоятельно раскрасить автомобиль или снять с него госномер. Такие незащищенные атрибуты, не имеющие ни сеттеров, ни геттеров, носят название ***полей объекта***.

Классы и объекты являются самыми главными понятиями ООП. Классы – это «чертежи», по которым изготавливаются объекты-кирпичики. У Стива Макконнелла (Steve McConnell) в книге «Совершенный код» приводится такое сравнение: «Класс можно рассматривать как форму для выпечки булочек, а объекты – как сами булочки».

Класс предназначен для того, чтобы хранить описания, необходимые для создания объектов и создавать такие объекты. Причем не любые, а со строго определенными параметрами. Например, на основе чертежа скворечника можно делать только объекты одного типа – скворечники. Разного размера, по-разному раскрашенные, даже из разного материала. Но все они будут скворечниками. И никак не собачьими конурами или гаражами. Если наш класс – курица, то мы будем получать из яиц цыплят, но не крокодильчиков или черепашек. Итак, класс – это основа для создания объектов.

В PascalABC.NET класс является разновидностью типа. Можно использовать готовый класс или описать собственный в разделе **type** с заголовком ИмяКласса = **class**. Объекты, созданные на основе некоторого класса, будут иметь тип, совпадающий с именем этого класса.

Построенные скворечники могут отличаться атрибутами – цветом, размером, материалом и т.д. Атрибуты - вторичные характеристики, по которым мы отличаем один скворечник от другого (или не сможем различить, если все будет одинаковым). Когда мы проектируем собственный класс, то обязательно должны решить, каким набором атрибутов будут обладать объекты, порождаемые этим классом, какие атрибуты сделать полями, а какие – свойствами. Атрибуты в объектах играют примерно такую же роль, как параметры в подпрограммах.

Атрибуты – это данные, которые объект «знает». В русском языке их можно описать при помощи существительных и прилагательных. Большинство объектов кроме наличия данных еще что-то «умеют». Умения описываются в русском языке глаголами, а в ООП они называются методами. В структурном программировании методам соответствуют подпрограммы. Например, объект класса *Собака* (обычно так говорят вместо более длинной фразы «Объект, созданный на основе класса *Собака*») может иметь методы *Сидеть*, *Лежать*, *Голос* и другие. К объекту можно обратиться с просьбой вызывать (выполнить) тот или иной метод и это делается при помощи сообщений. Как и набор атрибутов, нужный набор методов необходимо спроектировать при создании класса.

В языке PascalABC.NET класс состоит из полей, методов и свойств, каждое из которых в отдельности называется ***членом класса***.

Между классами и записями намного больше сходств, чем различий. Но все же различия есть:

- Классы имеют предка System.Object, а записи унаследованы от System.ValueType;
- Описываемая в программе запись не может быть унаследована и от нее нельзя наследовать;
- Классовые переменные имеют ссылочный тип, а записи имеют размерный тип;
- По умолчанию члены класса имеют модификатор доступа **internal**, а поля записи – **public**;
- При вызове конструктора записи инициализируются ее поля, но объект в памяти не создается.

13.1.1.1 Описание класса

В общем случае описание класса имеет следующую структуру

```
type
  имя_класса = class
    секция1
    секция2
    …
  end;
```

В теле класса может иметься произвольное количество секций, располагающихся в произвольном порядке. Каждая секция содержит ***модификатор доступа***, после которого следуют поля. Далее идут методы и свойства с доступом, определяемым модификатором, указанным для секции. В первой секции модификатор доступа может отсутствовать, в этом случае подразумевается модификатор **internal**.

```
модификатор доступа
  описания полей
  объявления или описания методов и описания свойств
```

Каждое поле, метод или свойство класса имеет модификатор доступа, задающий правила его видимости. В PascalABC.NET существуют четыре вида модификаторов доступа:

- **public** – открытый;
- **private** – закрытый;
- **protected** – защищенный;
- **internal** – внутренний.

К члену класса, имеющему модификатор доступа **public**, можно обратиться из любого места программы – это действительно «открытый» модификатор. Члены класса с модификатором **private** доступны только внутри методов этого класса. Члены класса с модификатором **protected** доступны внутри методов этого класса и всех его производных классов. Члены класса с модификатором **internal** доступны внутри ***.NET-сборки*** (здесь и далее под термином .NET-сборка подразумевается все множество файлов, необходимых для генерации .exe или .dll-файла). Кроме того, **private** и **protected** члены видны отовсюду в пределах модуля, в котором определен класс.

> В пределах модуля, в котором описан класс, его члены будут видны независимо от значения своего модификатора. За пределами модуля члены класса с модификатором **public** будут видны везде, с модификатором **internal** – в пределах .NET-сборки, в которую войдет модуль, с модификатором **protected** – только в кодах его производных классов, а с модификатором **private** совсем не будут видны.

Если класс является ***производным*** (также используется термин ***унаследованным*** - см. 13.2), то после ключевого слова **class** в круглых скобках указывается имя ***базового*** класса, за которым может следовать список поддерживаемых

интерфейсов (см. 13.9). Перед ключевым словом **class** может быть указано ключевое слово **sealed**, запрещающее наследовать от этого класса.

Отметьте, что внутри подпрограмм описывать классы запрещено.

Рассмотрим пример программы, полученной минимальной переделкой кода p07001 (см. 7.1.2).

```
// p13001
type
  Дробь = sealed class
  public
    Числитель, Знаменатель: integer;

    constructor(a, b: integer);
    begin
      (Числитель, Знаменатель) := (a, b)
    end;

    constructor(a: integer);
    begin
      (Числитель, Знаменатель) := (a, 1)
    end;

    function Out := $'({Числитель}/{Знаменатель})';

  end;

begin
  var a := new Дробь(3, 11); // стиль C#
  var b := new Дробь(13); // стиль C#
  var c := Дробь.Create(-1143, 65434); // стиль Object Pascal
  var d := Дробь.Create; // стиль Object Pascal
  Writeln(a.Out, NewLine, b.Out, NewLine, c.Out, NewLine, c.Out)
end.
```

Описан класс Дробь, имеющий два поля (Числитель и Знаменатель) и три метода. Все члены класса имеют модификатор доступа **public**. Наследование от этого класса запрещено атрибутом **sealed**. Схожесть описаний класса **class** и записи **record** из программы p07001 сомнений не вызывает. Ниже приведен результат работы программы.

```
(3/11)
(13/1)
(-1143/65434)
(0/0)
```

13.1.1.2 Конструкторы и создание объекта

Создание объекта в языке PascalABC.NET тесно связано с понятием ***конструктора***. Под конструктором понимают особую функцию в описании класса, имеющую фиксированное имя Create, которое можно не указывать. Конструктор создает в динамической памяти объект, инициализирует его поля, и возвращает

ссылку типа класс на этот объект. При описании конструктора вместо ключевого слова **function** используется ключевое слово **constructor**, а тип возвращаемого значения опускается.

Как и любая функция, конструктор допускает параметры, значения которых могут использоваться для инициализации полей. Как и любую функцию, конструктор из одного оператора можно записать в «короткой» однострочной форме. Можно описать несколько конструкторов, каждый из которых имеет свой набор параметров – нужный конструктор будет выбран по их фактическому количеству, типу и порядку следования. В PascalABC.NET всегда автоматически создается разновидность конструктора, не имеющая параметров. Такой конструктор инициализирует все поля значениями по умолчанию (числовые – нулевыми значениями, строковые – пустыми строками, логические – значением False).

Пример описания конструктора был приведен выше. В программе p13001 переменная d для создания объекта вызывает конструктор без параметров – именно тот, который был создан автоматически. Это пример того, как не нужно программировать. В самом деле, создать дробь, в знаменателе которой записан ноль, означает создать себе в будущем проблему. Хочется, чтобы по умолчанию создавалась дробь с нулевым значением? Напишите собственный конструктор без параметров, в котором числитель будет равен нулю, а знаменатель – единице:

```
// p13002
type
  Дробь = class
  public
    Числитель, Знаменатель: integer;

    constructor(a, b: integer);
    begin
      (Числитель, Знаменатель) := (a, b)
    end;

    constructor(a: integer) := (Числитель, Знаменатель) := (a, 1);

    constructor := (Числитель, Знаменатель) := (0, 1);

    function Out := $'({Числитель}/{Знаменатель})';

end;

begin
  var a := new Дробь(3, 11); // стиль C#
  var b := new Дробь(13); // стиль C#
  var c := Дробь.Create(-1143, 65434); // стиль Object Pascal
  var d := Дробь.Create; // стиль Object Pascal
  Writeln(a.Out, NewLine, b.Out, NewLine, c.Out, NewLine, d.Out);
end.
```

Если имя параметра в описании конструктора совпадает с именем поля, перед именем поля нужно использовать уточнение Self (см. 13.8.1), например: Self.alpha := alpha.

Создать объект можно двумя способами.

- **new** Имя_класса или **new** Имя_класса(Список_параметров);
- Имя_класса.Create или Имя_класса.Create(Список_параметров).

Использование **new** – это вызов конструктора в стиле языка C#. С помощью обращения к конструктору по его имени Create объекты создаются в языке Object Pascal. В приведенной программе продемонстрированы оба способа.

Чаще всего создание объекта происходит в операторе присваивания с тем, чтобы получить переменную со ссылкой на созданный объект:

```
var a := new Дробь(3, 11);
var b: Дробь;
b := Дробь.Create(-1143, 65434);
```

13.1.1.3 Три кита объектно-ориентированного программирования

В древности считали, что Земля стоит на трех китах, плавающих в Мировом океане. Точнее, на трех слонах, которые стоят на морской черепахе, а вот она уже стоит на трех китах. Как звали тех китов, уже никто и не помнит. ООП тоже опирается на «трех китов» и их имена известны: ***инкапсуляция***, ***наследование, полиморфизм***. Познакомимся с каждым из них.

- Инкапсуляция (англ. encapsulation) в программировании означает концепцию объединения данных и методов работы с ними в единую упаковку (капсулу). В PascalABC.NET под инкапсуляцией понимается принцип реализации класса, в соответствии с которым члены класса, используемые исключительно для реализации самого класса, не видны вне его. Пользователю предоставляется только интерфейс класса;
- Наследование (англ. inheritance) – это механизм, позволяющий описать новый класс на основе существующего (родительского) класса. При этом члены родительского класса наследуются классом-потомком. Наследование говорит о том, что **можно наследовать** члены класса;
- Полиморфизм (англ. polymorphism) в программировании – это возможность предоставлять один и тот же интерфейс для различных типов данных. Полиморфизм сообщает: в классах-потомках при необходимости **можно изменить** унаследованные методы.

Некоторые авторы в своих работах утверждают, что «китов» на самом деле четыре, имея в виду ***абстракцию***. Идея абстракции состоит в выделении наиболее существенных членов класса с целью создания на их основе интерфейсов.

Рассмотрим в деталях программу, иллюстрирующую многие особенности ООП.

```
// p13003
type
  Животное = class
  public
    Имя: string;
    constructor(Имя: string) :=
        Self.Имя := Имя;
  procedure Голос; virtual :=
      Println('...');
end;

Кошка = class(Животное)
public
  procedure Голос; override :=
      'Мяу!'.Println;
end;

Собака = class(Животное)
public
  Порода: string;
  constructor(Имя, Порода: string);
  begin
    inherited Create(Имя);
    Self.Порода := Порода
  end;
  procedure Голос; override;
  begin
    'Гав!'.Println
  end;
end;

Корова = class(Животное)
public
  procedure Голос; override :=
      'Му-у-у'.Println;
end;

begin
  var Хозяйство := new List<Животное>;
  Хозяйство.Add(new Собака('Шарик',
     'Двортерьер'));
  Хозяйство.Add(new Кошка('Мурка'));
  Хозяйство.Add(new Животное('Жорик'));
  Хозяйство.Add(new Корова('Милка'));
  Println('У нас в хозяйстве весело!');
  foreach var Зверушка in Хозяйство do
  begin
    Print(Зверушка.GetType.Name,
        Зверушка.Имя+':');
    Зверушка.Голос
  end
end.
```

Вывод программы выглядит следующим образом:

```
У нас в хозяйстве весело!
Собака Шарик: Гав!
Кошка Мурка: Мяу!
Животное Жорик: ...
Корова Милка: Му-у-у
```

В классе *Животное* модификатор **public** делает открытыми поле *Имя*, конструктор и метод *Голос*. Они будут видны везде. Конструктор принимает один параметр – *Имя* и заносит его в открытую переменную *Имя*. Поскольку внутри конструктора параметр *Имя* имеет приоритет перед глобальной переменной *Имя*, то для обращения к ней используется уточнение **Self**. В заголовке метода *Голос*, оформленного в виде процедуры, имеется ключевое слово **virtual**, объявляющее метод виртуальным (см. 13.3.1). Виртуальный метод – это некоторая заготовка, вместо которой впоследствии будет вызван требуемый метод.

Класс *Кошка* является потомком класса *Животное* и наследует от него поле *Имя*, конструктор и метод *Голос*, поскольку все они являются открытыми. Вместо виртуального метода *Голос* будет вызван метод *Голос* класса *Кошка*, о чем свидетельствует ключевое слово **override** (см. 13.2).

Класс *Собака* тоже потомок класса *Животное*. Он также наследует поле *Имя*, конструктор и метод *Голос*. У этого класса появилось второе поле – *Порода*. Определен собственный конструктор, принимающий два параметра. Для инициализации поля *Имя* в нем вызывается конструктор класса-предка. Вместо виртуального метода *Голос* будет вызван метод *Голос* класса *Собака*.

Класс *Корова* построен аналогично классу *Кошка*.

В основной программе определен список List с именем *Хозяйство*. Элементы этого списка имеют тип *Животное*, т.е. являются объектами класса *Животное*. В список добавляются четыре элемента – собака, кошка, некоторое животное и корова. Элементы с унаследованным типом (собака, кошка и корова) совместимы с элементами базового класса, поэтому проблем с их размещением в списке List не возникает. После завершения добавления для каждого элементов производится вывод.

Инкапсуляция в примере представлена инициализацией поля *Имя* в классах-потомках без обращения к нему.

Наследование представлено достаточно наглядно. От базового класса *Животное* наследуются классы *Собака*, *Кошка* и *Корова*, получающие поле *Имя*, конструктор класса и виртуальный метод *Голос*.

В примере также показан полиморфизм: в классах-потомках переопределен метод *Голос*, а в классе *Собака* введено дополнительное поле *Порода* и для его инициализации переопределен конструктор класса. Один и тот же метод *Голос* в зависимости от класса дает различный результат – так полиморфизм проявляет себя на практике.

13.1.1.4 Опережающие и циклические ссылки в классах

Если в объявлении класса поместить ссылку на объект, принадлежащий к другому классу, который объявлен ниже по тексту программы, при компиляции возникнет ошибка «неизвестное имя». Схожая ситуация для подпрограмм преодолевалась использованием ключевого слова **forward** (см. 4.5). С классами проблема решается аналогично – путем их опережающего объявления.

```
type
  A = class
    fa: B;
  end;

  B = class
    fb := 2;
  end;

begin
  var oa := new A;
  var ob := new B;
  oa.fa := ob;
  oa.fa.fb.Print
end.
```

НЕВЕРНО

```
type
  B = class;

  A = class
    fa: B;
  end;

  B = class
    fb := 2;
  end;

begin
  var oa := new A;
  var ob := new B;
  oa.fa := ob;
  oa.fa.fb.Print
end.
```

ВЕРНО

Опережающее объявление содержит только заголовок описания класса, за которым следует точка с запятой. В представленном примере компилятор «видит», что тип *B* – это некий класс, поэтому при просмотре тела класса *A* проблем с типом *B* уже не возникает. Аналогичным образом поступают, когда имеются циклические ссылки, например, описание класса *A* ссылается на класс *B*, а описание класса *B* ссылается на класс *A*.

```
// p13004
type
  B = class;

  A = class
    fa: B;
  end;

  B = class
    fb := 2;
    fc: A;
  end;

begin
  var oa := new A;
  var ob := new B;
  oa.fa := ob;
  oa.fa.fb.Println;
  ob.fc := oa;
  ob.fc.fa.fb.Println
end.
```

Попробуйте самостоятельно разобраться, почему при выполнении программы два раза выводится двойка.

Если в теле класса A используется ссылка на класс B, описанный текстуально ниже, для класса B делается опережающее объявление. При этом в теле класса A допускается только ссылаться на тип B. Попытки инициализировать объект класса B в конструкторе или методах, а также сослаться на какие-то его поля вызовут ошибку компиляции. Компилятор лишь понимает, что B – это класс, но и только.

13.1.2 Поля класса

Поля класса ассоциируются с обычными переменными и служат для тех же целей. Модификатор доступа распространяется на все поля, описанные в секции. В PascalABC.NET описание поля отличается от описания обычной переменной лишь отсутствием ключевого слова **var**. Разрешено автовыведение типа в случае совмещения описания поля с его с инициализацией. В каждой секции описание полей всегда предшествует описанию или объявлению методов и свойств.

При создании объекта его поля, если они не инициализированы явно, будут инициализированы автоматически. При этом числовые поля инициализируются нулевыми значениями, символьные – пустыми строками, логические – значением **False**, а ссылочные – значением **nil**. Инициализация может проводиться как в конструкторе, так и непосредственно при описании – в последнем случае код инициализации неявно вставляется в начало всех конструкторов класса. Это понятно: объект создается конструктором и проводить инициализацию его полей больше негде.

После того, как объект создан, можно обращаться к его полям при помощи точечной нотации. Такой записью вы пользовались много раз. Указывается имя объекта, за которым следует имя поля, например, *Объект.Класс.*

Поле может быть ***статическим***. Под статическими (иначе – ***классовыми***) полями понимаются поля, которые связаны не с объектом некоторого класса, а с классом в целом. С такими полями вы тоже знакомились, например, **integer**.MaxValue, где **integer** – имя класса. Это не оговорка – именно класса, несмотря на то, что мы все время говорим о типе. Но вспоминайте: класс – это тоже тип. И (вот он – момент истины!) в PascalABC.NET все базовые типы – это типы Microsoft .NET, но имеющие другие имена. Поэтому все время знакомства с PascalABC.NET вы использовали ООП, быть может, даже не задумываясь об этом.

В отличие от статических полей, остальные поля называются ***экземплярными***, т.е. связанными не с классом, а с его конкретным экземпляром – объектом. Чтобы сделать поле статическим, перед его именем нужно поместить ключевое слово **static**. Для совместимости с Delphi вместо **static** указывается ключевое слово **class**.

```
type
  МойКласс = class
  public
    a: integer;
    static b: real;
    class c := 1;
  end;

begin
  var x := new МойКласс;;
  Println(x.a, МойКласс.b, МойКласс.c)
end.
```

Как будет показано дальше, статическими могут быть и сами классы, и различные их члены, а не только поля. Статические поля не создаются в каждом объекте. Они создаются один раз и после этого доступны всем объектам, создаваемым на основе этого класса.

```
// p13005
type
  МойКласс = class
  public
    a: integer := 2;
    static b: real;
  end;

  Потомок = class(МойКласс)
    constructor;
    begin
      b := Sqrt(a);
    end;
  end;

begin
  МойКласс.b := -0.18;
  Println(МойКласс.b);
  var x := new МойКласс;;
  Println(x.a, МойКласс.b);
  var y := new Потомок;
  Потомок.b.Println
end.
```

Но какая нужда в том, чтобы делать поля статическими? Статические поля – хорошая альтернатива глобальным переменным. К ним можно обратиться и вне класса, причем объекты этого класса можно не создавать (первая и вторая строки основной программы в приведенном примере). В статических полях можно хранить начальные установки, различного рода счетчики, коды ошибок и т.д. Кроме того, статические поля экономят память, потому что не дублируются в каждом объекте подобно экземплярным полям.

13.1.3 Методы класса

Методы класса описываются в виде обыкновенных функций и процедур, так что писать методы вы уже умеете. Нужно только учитывать некоторые особенности, которые налагает ООП.

На область видимости метода влияет модификатор доступа секции, в которую этот метод помещен (см. 13.1.1.1). Метод может быть статическим; в этом случае он

принадлежит всему классу, а не отдельному объекту. Чтобы сделать метод статическим, нужно перед словом **function** или **procedure** указать **static**.

Статические методы могут работать только со статическими полями.

Для обращения к статическому методу нужно указать имя класса и через точку имя метода. Статические методы можно вызывать без создания объекта. Это позволяет создавать «классы-обертки». Например, если имеется некоторая старая библиотека, а хочется пощеголять ООП-стилем, можно обернуть ее в класс, запрещающий наследование (от греха подальше) и каждую подпрограмму сделать статическим методом. Сам класс можно поместить в отдельный модуль и откомпилировать его.

```
type
  СуперБиб = static class
    static function КвадратноеУравнение(a, b, c: real): (real, real);
    begin
      var x1, x2: real;
      // реализация
      Result := (x1, x2);
    end;

    static function ДетерминантМатрицы(A: array[,] of real): real;
    begin
      // реализация
    end;

  end;

begin
  var (x1, x2) := СуперБиб.КвадратноеУравнение(3, -2, 7.5);
  Println(x1, x2)
end.
```

Конечно же, статические методы придуманы не для того, чтобы тешить старую гвардию программистов обертками. Как и статические поля, они не тиражируют себя в объектах, экономя память и не засоряя объекты ненужным мусором. Можно описать в классе статический конструктор (он ведь тоже метод), который будет гарантированно вызван перед первым созданием объекта этого класса.

13.1.4 Свойства класса

Свойства отличаются от полей тем, что при обращении к ним выполняется какое-либо действие. Для описания свойств используется ключевое слово **property**. Общий формат записи свойства таков:

```
property имя: тип read ПриЧтении write ПриЗаписи;
```

Здесь *имя* указывает имя свойства, *тип* определяет тип свойства, *ПриЧтении* – действие, которое будет выполняться при попытке прочитать значение свойства, *ПриЗаписи* – действие, которое будет выполняться при попытке изменить значе-

ние свойства. Отсутствие фразы **read** ПриЧтении или **write** ПриЗаписи соответственно означает запрет попытки прочитать значение свойства или изменить его.

В качестве элемента ПриЧтении можно указать:
- имя функции, предназначенной для чтения значения свойства;
- имя поля соответствующего типа;
- выражение соответствующего типа.

В качестве элемента ПриЗаписи можно указать:
- имя процедуры, предназначенной для изменения значения свойства;
- имя поля соответствующего типа;
- оператор.

Обычно свойство связывается с определенным полем, скрытым внутри класса. Свойства позволяют получать значения таких полей и даже изменять их, причем делают этот процесс контролируемым. Например, указание write ИмяМетода при попытке изменить значение свойства может вызывать метод, изменяющий связанные с этим свойством значения. Так, если свойство *Сторона* хранит длину стороны квадрата, при изменении значения длины можно автоматически пересчитать площадь и периметр квадрата, обеспечив хранение в полях их актуальных значений. Можно поступить иначе: при попытке получить значение площади предварительно вычислить ее – такой прием полезен при продолжительных вычислениях. В самом деле, зачем что-то долго вычислять, если оно, быть может, не будет востребовано?

Функция, предназначенная для чтения свойства, должна вернуть требуемое значение с типом, указанным в описании, либо совместимым с ним по присваиванию. Такие функции обычно помещают в секции с модификаторами доступа **private** или **protected**. Если функция просто возвращает значение некоторого поля, можно вместо нее указать имя этого поля. В PascalABC.NET доступны ***расширенные свойства***, которые в случае чтения являются произвольными выражениями, разрешенными для правой части оператора присваивания и имеющими необходимый тип.

Процедура, предназначенная для записи свойства, должна иметь один параметр требуемого типа. Ее назначение обычно состоит в том, чтобы использовать значение параметра для модификации поля и (возможно) попутно выполнить еще какие-то действия. Такие процедуры обычно помещают в секции с модификаторами доступа **private** или **protected**. Если процедура лишь меняет значение некоторого поля, можно вместо нее указать имя этого поля. В PascalABC.NET доступны расширенные свойства, которые в случае записи являются выражениями, присваивающими полю нужное значение: при этом значение свойства доступно через служебное имя **value**.

Если объявляется некоторое свойство с именем A, то компилятор автоматически создаст методы Get_A и Set_A. Учитывайте это при выборе имен.

Рассмотрим иллюстрирующую программу p13006, вычисляющую некоторые параметры треугольника, заданного длинами сторон.

В классе *Треугольник* открыты только конструктор и три свойства (*a, b, c*), ассоциированные с длинами сторон треугольника. При создании объекта вызывается конструктор, присваивающий начальные значения закрытым полям *aa*, *bb* и *cc* минуя свойства *a*, *b* и *c*. Далее из конструктора запрашивается расчет параметров треугольника и вывод результатов. Затем вызывающая программа трижды изменяет значения свойств и при этом каждый раз вызываются соответствующие процедуры, указанные в описании свойств после write. Теперь уже каждая процедура после изменения значения поля для заданного свойства запрашивает расчет параметров треугольника и вывод результатов.

```
*** после вызова конструктора ***
Длины сторон: 3 4 5
Периметр 12, площадь 6.000
Радиус вписанной окружности 1.000
Радиус описанной окружности 2.500

*** после oT.c := 5.1 ***
Длины сторон: 3 4 5.1
Периметр 12.1, площадь 5.995
Радиус вписанной окружности 0.991
Радиус описанной окружности 2.552

*** после oT.b := 6.7 ***
Длины сторон: 3 6.7 5.1
Периметр 14.8, площадь 7.240
Радиус вписанной окружности 0.978
Радиус описанной окружности 3.540
Длины сторон: 18 6.7 5.1

*** после oT.a := 18 6.7 5.1
Треугольник не существует
```

Конечно, тут не ощущается смысл использования модификаторов **public** и **private**, и каждый раз при вводе точки после имени oT в основной программе вы видите в раскрывающемся списке перечень всех полей, методов и свойств. Но оформите описание класса отдельным модулем – и в вызывающей программе будут видны только члены класса, имеющие модификатор **public**.

```
// p13006
type
  Треугольник = class
  private
    aa, bb, cc, r1, r2,
    P, S, p2: real;

    procedure Вычислить;
    begin
      P := aa + bb + cc;
      p2 := P / 2;
      S := Sqrt(p2 * (p2 - aa) *
          (p2 - bb) * (p2 - cc));
      r1 := S / p2;
      r2 := aa * bb * cc / (4 * S)
    end;

    procedure Обнулить;
    begin
      (P, p2, S, r1, r2) :=
          (0, 0, 0, 0, 0)
    end;

    procedure SetAA(v: real);
    begin
      aa := v;
      ВывестиРезультаты;
      if not ВозможенТреугольник
      then Обнулить
    end;

    procedure SetBB(v: real);
    begin
      bb := v;
      ВывестиРезультаты;
      if not ВозможенТреугольник
      then Обнулить
    end;

    procedure SetCC(v: real);
    begin
      cc := v;
      ВывестиРезультаты;
      if not ВозможенТреугольник
      then Обнулить
    end;

    function ВозможенТреугольник :=
        (aa + bb > cc) and
        (aa + cc > bb) and
        (bb + cc > aa);

    procedure ВывестиРезультаты;
    begin
      Println('Длины сторон:', aa, bb, cc);
      if ВозможенТреугольник then
      begin
        Вычислить;
        $'Периметр {P}, площадь {S:f3}'.Println;
        Writeln('Радиус вписанной окружности ',
            r1:0:3);
        Writeln('Радиус описанной окружности ',
            r2:0:3)
      end
      else
        Println('Треугольник не существует')
    end;

  public
    constructor(pa: real := 3.0; pb: real := 4.0;
        pc: real := 5.0);
    begin
      aa := pa;
      bb := pb;
      cc := pc;;
      ВывестиРезультаты;
      if not ВозможенТреугольник then Обнулить
    end;

    property a: real read aa write SetAA;
    property b: real read bb write SetBB;
    property c: real read cc write SetCC;

  end;

  begin
    var oT := new Треугольник;
    oT.c := 5.1;
    oT.b := 6.7;
    oT.a := 18;
  end.
```

Если вы знакомы с тригонометрией, то возможно захотите добавить в класс способность находить углы треугольника. Для этого достаточно ввести три расширенных свойства, при чтении которых будут вычисляться необходимые выражения.

Пусть *УголA*, *УголB* и *УголC* – углы, лежащие против сторон *a*, *b* и *c* соответственно. Формулы для расчета этих углов хорошо известны.

```
property УголA: real read RadToDeg(ArcCos((bb * bb + cc * cc - aa * aa) /
    (2 * bb * cc)));
property УголB: real read RadToDeg(ArcCos((aa * aa + cc * cc - bb * bb) /
    (2 * aa * cc)));
property УголC: real read RadToDeg(ArcCos((bb * bb + aa * aa - cc * cc) /
    (2 * bb * aa)));
```

Теперь для получения значения нужного угла в градусах достаточно обратиться к соответствующему свойству. В таком стиле можно оформлять вычисления значений, которые нужны только по конкретным запросам, а не делаются автоматически при смене исходных значений. Конечно, не следует злоупотреблять использованием вычисляемых полей, если планируется создавать большое количество экземпляров класса (например, тысячи или десятки тысяч треугольников), поскольку это может внести серьезное замедление в работу программы.

Рассмотрим случай использования свойства для поля, которое хранится в одном виде, а читается и принимает новое значение в другом. Пусть в классе есть некоторое закрытое поле *Угол*, хранящее значение угла в радианной мере. Введем свойство *УголВГрадусах*, позволяющее работать в более привычной мере углов – градусной.

```
property УголВГрадусах: real read RadToDeg(Угол) write Угол := DegToRad(value);
```

У свойств есть одна особенность, о которой не следует забывать: свойство не может быть var-параметром подпрограммы, т.е. подпрограмма не может менять значение свойства, полученного как параметр. Это ограничение проверяется при компиляции. Если возникает необходимость передать свойство в подпрограмму для его изменения, придется воспользоваться вспомогательной переменной.

```
procedure Sqt(var t: real);
begin
  t := Sqrt(t);
end;

begin
  var oT := new Треугольник;
  Sqt(oT.c);
end.
```

НЕВЕРНО

```
procedure Sqt(var t: real);
begin
  t := Sqrt(t);
end;

begin
  var oT := new Треугольник;
  var p := oT.c;
  Sqt(p);
  oT.c := p;
end.
```

ВЕРНО

Подобно полю, свойство может быть объявлено статическим путем добавления ключевого слова **static** перед **property**. Статические свойства не могут в своих сеттерах и геттерах обращаться к экземплярным полям, поскольку они принадлежат всему классу, а экземплярные поля принадлежат конкретному объекту, который может на момент обращения и не существовать. Статические свойства не

присутствуют в самом объекте, но каждый объект может к ним обращаться через имя класса. К ним также можно обращаться, не создавая объектов. И еще, статические члены класса (как и статические поля) не загромождают общее пространство имен.

Свойство может быть объявлено виртуальным (см. 13.3.1), для чего после его описания добавляется ключевое слово **virtual**.

13.1.5 Индексные свойства

До сих пор мы использовали преимущественно поля базовых типов. Но поле может иметь любой тип, в том числе быть массивом. Индексные свойства ведут себя так же, как и поля-массивы. Как и свойства, рассмотренные выше, индексные свойства, позволяют получать значения полей и менять такие значения, попутно выполняя некоторые действия. Общий формат записи индексного свойства таков:

```
property имя[ОписаниеИндексов]: тип read ПриЧтении write ПриЗаписи;
```

Здесь *имя* указывает имя индексного свойства, *тип* определяет его тип, *ПриЧтении* – действие, которое будет выполняться при попытке прочитать значение индексного свойства, *ПриЗаписи* – действие, которое будет выполняться при попытке изменить значение индексного свойства. Отсутствие фразы **read** ПриЧтении или **write** ПриЗаписи соответственно означает запрет попытки прочитать значение или изменить его.

Простейшее *ОписаниеИндекса* может иметь вид [индекс: тип], где *индекс* – имя индексной переменной. Если индекса два и они имеют одинаковый тип, то используют запись вида [индекс1, индекс2: тип]. В отличие от массива, тип индекса может быть любым, в том числе, строковым. Строковые индексы позволяют достаточно просто организовывать ассоциативные массивы.

В качестве элемента ПриЧтении можно указать:

- имя функции, предназначенной для чтения значения индексного свойства;
- выражение соответствующего типа.

В качестве элемента ПриЗаписи можно указать:

- имя процедуры, предназначенной для изменения значения индексного свойства;
- оператор;
- имя поля.

Индексные свойства используются в работе с полями, имеющими ссылочный тип и являющимися контейнерами, например, массивами. Начинающие программисты могут совершить ошибку, решив что ссылочное свойство, для которого в описании отсутствует write, гарантирует только чтение и, тем самым, страхует данные от изменения. Вспомните, что в PascalANC.NET можно передавать значение, ссылку и копию ссылки. Передача значения (копирование) гарантирует невозможность изменения исходных данных, передача ссылки, наоборот, обеспечивает возмож-

ность изменить значение, а передача копии ссылки не дает возможности подменить сам контейнер, но никак не может запретить менять его содержимое. К примеру, если индексное свойство A связать с целым массивом и запретить его изменение, то будет невозможно, прямо или косвенно, выполнить присваивание вида A := B, но возможность изменить значение A[i] присваиванием останется.

В иллюстративных целях дан пример, в котором класс определен в модуле p13007, а основная программа оформлена отдельной программной единицей p13008 с тем, чтобы обеспечить защиту членов класса с модификатором **private**.

```
unit p13007;

type
  SortByInsert = class

  private
    ma: array of integer;
    procedure Sort;
    begin
      for var i := 1 to ma.High do
        if ma[i - 1] > ma[i] then
        begin
          var (p, j) := (ma[i], i - 1);
          while (j >= 0) and (ma[j] > p) do
             (ma[j + 1], j) := (ma[j], j - 1);
          ma[j + 1] := p
        end;
    end;

  public
    constructor(pa: array of integer);
    begin
      ma := pa;
      Sort
    end;
    property ia[i: integer]: integer
    read ma[i]
    write
      begin
        ma[i] := value;
        Sort
      end;
    default;

    property a: array of integer
        read ma;

  end;
end.
```

```
uses p13007; // p13008

begin
  var a := ArrRandom(15, -50, 50);
  Println('Исходный массив');
  a.Println;
  var oT := new SortByInsert(a);
  Println('Массив после сортировки');
  oT.a.Println;
  $'Заменим восьмой по порядку элемент ({oT[7]}) нулем'.Println;
  oT[7] := 0;
  Println('Новое состояние элементов массива');
  oT.a.Println;
  Println('Обнулим третий элемент через свойство без write');
  oT.a[2]:=0;
  Println('Последнее состояние элементов массива');
  oT.a.Println;
end.
```

Обращение к свойству oT.a используется для работы с ним, как с единым массивом. Обращение oT[7] демонстрирует работу с индексным свойством, назначенным для класса по умолчанию. Попытка обращения к свойству oT.a для изменения элементов массива присваиванием будет пресечена компилятором, а вот модификация отдельного элемента окажется разрешена. Обратите внимание, что после модификации элемента массива посредством oT.a[2]:=0 сортировка не выполняется

13.1.6 Автоклассы и автосвойства

С первого взгляда ***автокласс*** – это разновидность класса «для ленивых». На самом деле – удобная вещь! При минимуме затрат на описание вы получаете полноценный класс. Достаточно лишь перед словом **class** добавить описатель **auto**.

```
type
  МойАвтокласс = auto class
  private
    f1: integer;
  public
    f2: real;
  end;

begin
  var a := new МойАвтокласс(18, 3.5);
  Writeln(a) // выводится (18,3.5)
end.
```

Для автокласса автоматически генерируется конструктор, позволяющий при создании объекта инициализировать **все** его поля. Кроме этого, генерируется метод .ToString, позволяющий привести к строке значения всех полей. Именно этот метод неявно используется при выводе значений посредством процедуры Write. Автоклассы **не могут** быть унаследованными (потеряется смысл инкапсуляции).

Чтобы понять, насколько хорошо вы усвоили материал об автоклассах, попробуйте определить, что произойдет и почему, если поле *f1* описать как

```
static f1: integer;
```

Автосвойство автоматически создает в классе закрытое поле, предоставляет к нему доступ по чтению-записи и позволяет инициализировать это поле при создании объекта. Чтобы определить автосвойство, используется один из следующих форматов:

```
auto property ИмяСвойства: ТипСвойства;
auto property ИмяСвойства: ТипСвойства := Выражение;
auto property ИмяСвойства := Выражение; // тип выводится автоматически
```

Закрытые поля именуются по формату #PFieldN, где N = 1, 2, … – порядковый номер автосвойства в описании. Это имя может понадобиться внутри класса для непосредственного обращения к закрытому полю.

Для чего нужны автосвойства? Ведь разницы между использованием поля и автосвойства не видно. На самом деле, автосвойства - это способ быстро и коротко определять свойства. Если даже вы не видите смысла использовать в своей задаче свойства, обходясь полями, это не означает, что в будущем ситуация не изменится. Например, если понадобится внести какой-то контроль значений или контроль доступа к полю, придется переделывать существенную часть кода. Либо сделать свойства полями, а затем вновь завести поля, придумать им имена и связать поля со свойствами. Большей части этой работы можно избежать, если сразу использовать автосвойства вместо полей.

13.2 Наследование

Цель наследования состоит в том, чтобы не дублировать код в реализации каждого класса, а наследовать его. Функционально наследование предполагает унаследование интерфейса (заголовков методов).

Механизм наследования позволяет описывать новые (унаследованные) классы на основе базовых (родительских) классов. Классы-потомки наследуют от родительского класса все его члены: поля, методы и свойства. В процессе наследования можно доопределить новые члены класса и/или переопределить наследуемые методы и свойства. Ранее уже было сказано о том, как описывается унаследованный класс (см. 13.1.1.1). Его члены наследуются без каких-либо дополнительных описаний.

Чтобы доопределить член класса, достаточно добавить его описание в унаследованный класс. В переопределяемом методе можно обратиться к методу из базового класса при помощи вызова вида

```
inherited ИмяМетода;
```

В случае совпадения имени метода и параметров с описанными в базовом классе, достаточно указать только ключевое слово **inherited**.

Такое обращение может потребоваться, чтобы не дублировать код из базового класса. Пример был показан в программе p13003 (см. 13.1.1.3) для конструктора класса *Собака*.

Для конструкторов используются более сложные правила наследования. Если в классе-потомке конструкторы не описаны, то они наследуются из родительского класса. В случае, если описать хотя бы один конструктор, ни один из конструкторов родительского класса унаследован не будет. В PasacalABC.NET конструктор с модификатором **protected** и без параметров всегда неявно присутствует в любом описании класса. При вызове конструктора из родительского класса, обращение к нему в теле конструктора класса-потомка должно указываться первым оператором (вы помните, что конструктор всегда имеет имя Create – оно может понадобиться для записи с **inherited**). Если такого обращения нет, по умолчанию из родительского класса будет вызван конструктор без параметров. Если в родительском классе конструктор по умолчанию отсутствует (например, класс вызывается из модуля, созданного не в PascalABC.NET), компилятор фиксирует ошибку. Отметьте, что при вызове конструктора из родительского класса создания объекта не происходит, потому что этот конструктор вызывается как процедура, а не как функция, возвращающая объект.

Приведенные правила наследования отличаются от принятых в Object Pascal, где конструкторы наследуются всегда, что может оказаться причиной потери совместимости кода.

Рассмотрим пример наследования конструкторов, приведенный в Справке.

Определен родительский класс *A*, состоящий из поля *i* и конструктора с одним параметром *i*. Также, класс содержит конструктор по умолчанию, не содержащий параметров. От класса *A* наследуется класс *B*, состоящий из поля *j*, конструктора без параметров и конструктора с двумя параметрами (*i*, *j*).

Класс *B* унаследует от родительского класса *A* поле *i* и конструктор с одним параметром. Конструктор без параметров (по умолчанию) унаследован не будет, поскольку он перекрывается конструктором, указанным явно. При вызове конструктора с двумя параметрами вначале вызывается унаследованный конструктор с одним параметром, инициализирующий поле *i*, а затем будет инициализировано поле *j*.

Определение класса *C* пустое, но он является потомком класса *B*, поэтому унаследует поля *i* и *j*, а также все три конструктора. Поэтому, например, оператор x := **new** C; создаст объект x класса C, в котором поле *i* будет иметь значение 0 (его инициализирует .NET), а поле *j* – значение 1.

```
type
  A = class
    i: integer;
    constructor Create(i: integer);
    begin
      Self.i := i;
    end;
  end;
  B = class(A)
    j: integer;
    constructor Create;
    begin
      j := 1;
    end;
    constructor Create(i, j: integer);
    begin
      inherited Create(i);
      Self.j := j;
    end;
  end;
  C = class(B)
  end;
```

Что будет унаследовано от автокласса, в котором можно себе позволить не определять ничего, кроме списка полей, а необходимый конструктор строится автоматически? Все, что в нем имеется. Поля и конструктор, их инициализирующий. А также конструктор без параметров, определенный в автоклассе по умолчанию. И это еще раз подтверждает полноценность автоклассов.

13.2.1 Принцип подстановки Лисков

В 1987 году Барбара Лисков (Barbara Liskov) сформулировала один из важнейших принципов наследования, известный теперь как LSP – Liskov Substitution Principle (принцип подстановки Лисков). В вольном изложении он звучит так: «Должна быть возможность вместо базового типа подставить любой его подтип». На практике реализация этого принципа означает, что поведение объектов любого унаследованного класса не должно входить в противоречие с поведением объектов базового класса. Что дает выполнение этого требования? Во-первых, не теряется функциональность базового класса при использовании его потомков. Например, если объекты базового класса имеют способность летать, ее будут иметь объекты любых унаследованных классов. Во-вторых, это делает всегда возможным разместить ссылку на объект унаследованного типа в переменной базового типа. В то же время невозможно обратное: разместить ссылку на объект базового типа в переменной унаследованного типа. Такая ошибка фиксируется компилятором.

13.2.2 Перегрузка операций

Кроме переопределения методов и конструкторов, PascalABC.NET допускает переопределение операций над типами данных, определяемыми пользователем. Вво-

дить собственные знаки операций не разрешается, можно только переопределять существующие, кроме @, ^, **as**, **is** и **new**. О перегрузке бинарных операций, не возвращающих значения (+=, -=, *=, /=) будет сказано отдельно. Можно также перегружать операции приведения типа. Перегружать ранее перегруженные операции не разрешается.

Для некоторого типа данных T, являющегося классом или записью, перегрузка операции осуществляется посредством статической функции-метода с заголовком

```
static function operator знак_операции(параметры): тип_результата;
```

Здесь *параметры* – это описание одного параметра для унарной операции и двух параметров для бинарной.

> Хотя бы для одного параметра тип данных должен совпадать с типом класса или записи, в котором определяется перегрузка.

Рассмотрим пример перегрузки арифметических операций для класса, предназначенного для работы с простыми дробями.

Класс *Дробь* содержит два поля – *Числитель* и *Знаменатель*. Знаменатель инициализируется единицей, что дает возможность определять в конструкторе целые числа, которые можно рассматривать, как частный случай дроби с единичным знаменателем. Перегружены четыре операции – унарный минус, сложение, умножение и деление. Также, перегружен стандартный метод ToString (обратите внимание на наличие **override**), который определен в «прародителе всех классов» - классе Object и используется для вывода значений полей в определенном там же методе Write.

В примере вычисляется значение дроби

$$\frac{\frac{3}{4}-\frac{2}{5}}{\frac{3}{4}+\frac{2}{5}}=\frac{\frac{15-8}{4\times5}}{\frac{15+8}{4\times5}}=\frac{\frac{7}{20}}{\frac{23}{20}}=\frac{7\times20}{23\times20}=\frac{140}{460}=\frac{7}{23}$$

Вычисленное значение 140/460 можно сократить до 7/23, но программа p13009 выполнять сокращение дробей не умеет. Для сокращения дроби нужно найти НОД числителя и знаменателя и затем выполнить их деление на НОД.

Более полный класс для работы с простыми дробями произвольной точности можно найти, например, в библиотеке NumLibABC (см. 11.5.2.3) – там он имеет имя Fraction.

```
// p13009
type
  Дробь = class
  public
    Числитель: integer;
    Знаменатель := 1;

    constructor(Чис, Знам: integer) :=
        (Числитель, Знаменатель) := (Чис, Знам);
    constructor(Чис: integer) := Числитель := Чис;
    static function operator -(a: Дробь) :=
        new Дробь(-a.Числитель, a.Знаменатель);
    static function operator +(a, b: Дробь) :=
        new Дробь(a.Числитель * b.Знаменатель +
          b.Числитель * a.Знаменатель,
          a.Знаменатель * b.Знаменатель);
    static function operator -(a, b: Дробь) :=
        new Дробь(a.Числитель * b.Знаменатель -
          b.Числитель * a.Знаменатель,
          a.Знаменатель * b.Знаменатель);
    static function operator *(a, b: Дробь) :=
        new Дробь(a.Числитель * b.Числитель,
            a.Знаменатель * b.Знаменатель);
    static function operator /(a, b: Дробь) :=
        new Дробь(a.Числитель * b.Знаменатель,
            b.Числитель * a.Знаменатель);
    function ToString: string; override :=
        $'{Числитель}/{Знаменатель}';
end;

begin
  var a := new Дробь(3, 4);
  var b := new Дробь(-2, 5);
  var c := -(a + b) / (a - b);
  Writeln(c) // -140/460 – дробь не сокращена!
end.
```

Для перегрузки операторов приведения типа используются статические функции, у которых вместо имени указывается **operator implicit** или **operator explicit** для случая неявного или явного приведения соответственно. При вызове такого оператора используется имя типа или класса.

Перегрузка бинарных операций, не возвращающих значения (+=, -=, *=, /=) осуществляется с помощью статической процедуры-метода, первый параметр которой передается по ссылке и имеет тип записи или класса, в котором определяется данная операция, а второй параметр передается по значению и совместим по присваиванию с первым.

Программа p13010 – это пример, иллюстрирующий перегрузку оператора приведения типа и бинарной операции +=. В данном случае кортеж из двух целочисленных значений приводится к объекту класса Дробь.

```
// p13010
type
  Дробь = class
  public
    Числитель: integer;
    Знаменатель := 1;
    constructor(Чис, Знам: integer) :=
        (Числитель, Знаменатель) := (Чис, Знам);
    constructor(Чис: integer) := Числитель := Чис;
    static function operator -(a: Дробь) :=
        new Дробь(-a.Числитель, a.Знаменатель);
    static function operator +(a, b: Дробь) :=
        new Дробь(a.Числитель * b.Знаменатель +
          b.Числитель * a.Знаменатель,
          a.Знаменатель * b.Знаменатель);
    static function operator -(a, b: Дробь) :=
        new Дробь(a.Числитель * b.Знаменатель -
          b.Числитель * a.Знаменатель,
          a.Знаменатель * b.Знаменатель);
    static function operator *(a, b: Дробь) :=
        new Дробь(a.Числитель * b.Числитель,
            a.Знаменатель * b.Знаменатель);
     static function operator /(a, b: Дробь) :=
        new Дробь(a.Числитель * b.Знаменатель,
            b.Числитель * a.Знаменатель);
    function ToString: string; override :=
        $'{Числитель}/{Знаменатель}';
    static function operator explicit
        (a:(integer, integer)): Дробь :=
        new Дробь(a[0], a[1]);
    static procedure operator += (var a: Дробь; b: Дробь) :=
         a := a + b;
end;
begin
  var a := Дробь((3, 4));
  var b := new Дробь(-2, 5);
  var c := (a + b) / (a - b);
  c += a * b;
  Writeln(c) // 40/9200
end.
```

Операции можно также перегружать при помощи методов расширения. При этом не указывается, что метод является статическим (ключевое слово **static** или **class**), но в конце заголовка добавляется ключевое слово **extensionmethod**. Подробнее об этом см. 13.8.2.

13.3 Полиморфизм

Под полиморфизмом понимается способность переменной базового класса вызывать методы того из классов, связанных наследованием, объект которого в момент вызова содержится в этой переменной. Полиморфизм проявляется в случае, когда для нескольких объектов (например, элементов класса-контейнера) требуется выполнение одного и того же действия и в зависимости от своего класса каждый объект это действие может выполнять по-своему.

Для реализации полиморфизма в базовом классе определяются так называемые ***виртуальные*** методы, которые затем переопределяются в каждом унаследованном классе, реализуя необходимое поведение. В программе p13003 (см. 13.1.1.3) таковым является процедура-метод *Голос*.

13.3.1 Виртуальные методы

Рассмотрим незначительно модифицированный пример из Справки.

```
// p13011
type
  Базовый = class
  public
    procedure Вывод := Println('Базовый класс');
  end;

  Унаследованный = class(Базовый)
  public
    procedure Вывод := Println('Унаследованный класс');
  end;

begin
  var a: Базовый := new Унаследованный;
  a.Вывод
end.
```

Переменная *a* принадлежит родительскому классу *Базовый* и хранит ссылку на объект класса-потомка *Унаследованный*. При обращении к методу *Вывод* будет вызвана процедура из класса *Базовый*, поскольку компилятору явно указан тип переменной *a*.

Говорят, что если решение о связывании имени метода с его определением (телом метода) принимается на этапе компиляции программы, имеет место ***раннее связывание***. Существует также ***позднее связывание***, при котором решение о связи имени метода с его телом принимается во время выполнения программы в зависимости от реального объекта, на который ссылается переменная базового класса.

В случае позднего связывания метод называется ***виртуальным*** и для него указывается модификатор **virtual**. Подпрограммы-методы, которые связываются с

именем переменной, в каждом классе объявляются с модификатором **override**, а сама переменная получает название ***полиморфной***.

Использование виртуального метода совершенно незначительно поменяет код программы, но теперь уже будет вызван метод *Вывод* из класса *Унаследованный* вследствие позднего связывания:

```
// p13012
type
  Базовый = class
  public
    procedure Вывод; virtual := Println('Базовый класс');
  end;

  Унаследованный = class(Базовый)
  public
    procedure Вывод; override := Println('Унаследованный класс');
  end;

begin
  var a: Базовый := new Унаследованный;
  a.Вывод
end.
```

Тип класса, который хранится в переменной базового класса при позднем связывании, называют ***динамическим типом***.

Виртуальный метод в родительском классе и все методы, переопределяющие его в классах потомках, объединяются в ***цепочку виртуальности***. Если в каком-либо класса-потомке цепочку нужно разъединить, вместо модификатора **override** используется модификатор **reintroduce**. При необходимости начать новую цепочку в классе-потомке, указывают сразу и **virtual** (начало цепочки виртуальности), и **reintroduce**. Наличие модификатора **reintroduce** не обязательно, оно лишь подавляет выдачу предупреждения компилятора о том, что следует указывать или **virtual**, или **override**, если Вы о них забудете или намеренно не напишите.

По поводу «цепочки виртуальности» в Borland Delphi, откуда и пришел модификатор **reintroduce**, было написано немало. Разобраться в этой теме проще всего на примере. Приведенная программа содержит цепочку наследований классов, в которой каждый следующий ниже по тексту класс является потомком предшествующего. В базовом классе *Родитель* определен виртуальный метод *Вывод*. В каждом классе-потомке метод *Вывод* перегружен (указан модификатор **override**). Выражение Self.GetType.Name возвращает имя класса, в теле которого оно указано – это важно.

```
// p13013
type
  Родитель = class
  public
    procedure Вывод; virtual :=
        Println(Self.GetType.Name + ': Родитель');
  end;

  Дочка = class(Родитель)
  public
    procedure Вывод; override :=
        Println(Self.GetType.Name + ': Дочка');
  end;

  Внучка = class(Дочка)
  public
    procedure Вывод; override :=
        Println(Self.GetType.Name + ': Внучка');
  end;

  Правнучка = class(Внучка)
  public
    procedure Вывод; override :=
        Println(Self.GetType.Name + ': Правнучка');
  end;

begin
  Seq&<Родитель>(new Дочка, new Правнучка, new Родитель, new Внучка)
      .Foreach(b -> begin b.Вывод end)
end.
```

Программа выведет следующие строки:

```
Дочка: Дочка
Правнучка: Правнучка
Родитель: Родитель
Внучка: Внучка
```

В каждом объекте метод *Вывод* показал имя класса, в теле которого он был вызван, а затем идентифицировал себя. В каждом классе-потомке метод *Вывод* оказался ожидаемо перегружен.

Теперь прервем цепочку виртуальности для метода *Вывод* в классе *Внучка*. Удалим в заголовке метода модификатор **override**.

```
Внучка = class(Дочка)
  public
    procedure Вывод := Println(Self.GetType.Name + ': Внучка');
  end;
```

Отсутствие модификатора **override** прервет цепочку наследований. Появится еще один метод *Вывод*, имеющий такой же заголовок, как метод *Вывод* в родительском классе *Дочка*. Если бы мы имели объект типа *Внучка*, для него был бы вызван его

собственный метод *Вывод*. Соответственно, для объекта типа *Дочка* будет вызван метод *Вывод* из описания класса *Дочка* и т.д. Но в цикле **forech** переменная *b* имеет тип *Родитель*, поэтому будет вызываться метод *Вывод* именно этого класса, перегруженный для всех потомков в классе *Дочка* . Это еще не все. Убрав модификатор **override**, мы одновременно лишили класс-потомок *Правнучка* возможности перегрузить унаследованный метод *Вывод*, поэтому компилятор зафиксирует ошибку («Нет метода для переопределения»). А по поводу определения метода *Вывод* в классе *Внучка* компилятор даст предупреждение «Ожидался модификатор **override** либо **reintroduce**». Так зачем нужен модификатор **reintroduce**? Всего лишь для того, чтобы подавить сообщение компилятора о своем отсутствии. Больше он не делает ничего.

Немного поправим наш код:

```
Внучка = class(Дочка)
  public
    procedure Вывод; reintroduce :=
        Println(Self.GetType.Name + '- Внучка');
  end;

Правнучка = class(Внучка)
  public
    procedure Вывод :=
        Println(Self.GetType.Name + '- Правнучка');
  end;
```

Вывод будет ожидаем:

```
Дочка: Дочка
Правнучка: Дочка
Родитель: Родитель
Внучка: Дочка
```

Мы прервали цепочку виртуальности на классе *Внучка* и для всех потомков этого класса выполняется метод *Вывод* из класса-предка *Дочка*. Именно эту картину мы и наблюдаем: три строки с «дочкой».

Что изменится, если в классе *Внучка* добавить модификатор **virtual** с тем, чтобы перегрузить метод *Вывод* в классах-потомках, а в классе *Правнучка* вернуть модификатор **override**?

```
Внучка = class(Дочка)
  public
    procedure Вывод; virtual; reintroduce :=
        Println(Self.GetType.Name + '- Внучка');
  end;

Правнучка = class(Внучка)
  public
    procedure Вывод; override :=
        Println(Self.GetType.Name + '- Правнучка');
  end;
```

Догадались? Правильно, в данной программе – ничего! От того, что мы добавили модификатор **virtual**, потомки класса *Внучка* получили возможность наследовать метод *Вывод* и перегружать его. Но цепочка виртуальности по-прежнему прервана, и по-прежнему переменная *b* в цикле **foreach** имеет тип *Родитель*. А значит, по-прежнему будет вызываться метод *Вывод* из класса *Дочка*.

Более того, мы вообще можем сделать классы Внучка и Правнучка пустыми – и это тоже ничего не изменит:

```
Внучка = class(Дочка)
end;

Правнучка = class(Внучка)
end;
```

Почему? Потому что все равно метод *Вывод* будет наследоваться из класса *Дочка*.

Прежде чем разрывать цепочку виртуальности подумайте – а оно вам действительно так уж надо?

> Перегружающий метод не может иметь модификатор с меньшей видимостью, чем у перегружаемого виртуального метода.

Аналогично методам, виртуальными можно объявлять сеттеры и геттеры свойств. В этом случае их обычно помещают не в закрытую (**private**) секцию, а в защищенную (**protected**). Также, можно объявить виртуальными и перегрузить сами свойства.

13.3.2 Абстрактные методы и классы

Абстрактными называются методы, содержащие только заголовок и служащие для переопределения в классах-потомках. Фактически, это виртуальные методы, лишенные параметров и тела. Вместо ключевого слова **virtual** указывается ключевое слово **abstract**. В классах-потомках для методов, предназначенных для перегрузки, обязательно указывается **override**.

Классы, содержащие абстрактные методы, также называются абстрактными. Можно сделать класс абстрактным явно, даже если он не содержит абстрактных методов. С этой целью при описании класса перед словом **class** указывается ключевое слово **abstract**:

```
type AC = abstract class
```

Абстрактные классы используются исключительно в качестве родительских, поскольку создавать объекты на их основе невозможно.

Пусть нужно создать класс, который может вычислять и выводить площади прямоугольников, заданных длинами сторон. Ниже дается пример такой реализации.

```
// p13014
type
  Прямоугольник = auto class
    Длина: real;
    Ширина: real;
    function Площадь: real := Длина * Ширина;
    procedure Вывести := $'{Длина} x {Ширина} = {Площадь}'.Println;
  end;

begin
  var L := new List<Прямоугольник>;
  L.Add(new Прямоугольник(12, 15.5));
  L.Add(new Прямоугольник(18.2, 9.4));
  L.Add(new Прямоугольник(15, 15));
  foreach var p in L do
    p.Вывести
end.
```

В процессе работы программы выведет следующие строки:

```
12 x 15.5 = 186
18.2 x 9.4 = 171.08
15 x 15 = 225
```

В последней строке найдена площадь прямоугольника, имеющего одинаковые длины сторон, т.е. квадрата. Для квадратов разумно задавать длину только одной стороны, поэтому опишем еще один класс *Квадрат*. Поскольку квадрат отличается от прямоугольника лишь равенством сторон, логично сделать класс *Квадрат* потомком класса *Прямоугольник*. Наследование позволит поместить объекты обоих классов в контейнер с типом родительского класса. Конструктор класса *Квадрат* будет принимать длину стороны и заносить ее в оба унаследованных от класса *Прямоугольник* поля. Функция для вычисления площади и процедура вывода также будут унаследованы. Пример такой реализации дается в программе p13015. Она выводит те же строки, что и предыдущая программа.

```
// p13015
type
  Прямоугольник = auto class
    Длина: real;
    Ширина: real;
    function Площадь: real := Длина * Ширина;
    procedure Вывести := $'{Длина} x {Ширина} = {Площадь}'.Println;
  end;

  Квадрат = class(Прямоугольник)
    constructor(Сторона: real);
    begin
      Длина := Сторона;
      Ширина := Сторона;
    end;
  end;
```

```
begin
  var L := new List<Прямоугольник>;
  L.Add(new Прямоугольник(12, 15.5));
  L.Add(new Прямоугольник(18.2, 9.4));
  L.Add(new Квадрат(15));
  foreach var p in L do
    p.Вывести
end.
```

Каковы недостатки такой реализации? Во-первых, объекты типа *Квадрат* хранят избыточную информацию – два поля *Длина* и *Ширина* вместо одного поля *Сторона*. Поля открытые, поэтому их значения можно менять. Посмотрим, что будет, если написать следующий код:

```
begin
  var L := new List<Прямоугольник>;
  L.Add(new Прямоугольник(12, 15.5));
  L.Add(new Прямоугольник(18.2, 9.4));
  L.Add(new Квадрат(15));
  L[2].Ширина := 14;
  foreach var p in L do
    p.Вывести
end.
```

L[2] – это последний элемент списка, т.е. объект класса *Квадрат*. Но теперь он превратился в прямоугольник, что некорректно. Конечно, в рамках одной маленькой задачи это не принесло вреда. Но в более сложных проектах такие метаморфозы недопустимы.

После попытки сделать новый класс *Квадрат* стало даже хуже, чем было! Ведь если поменять, как было показано выше, длину или ширину, то квадрат перестанет быть квадратом, а функция *Площадь* вернет старое значение. На ум приходит фраза «Хотели, как лучше, а получилось, как всегда». Чтобы предотвратить подобные инциденты, нужно создавать класс *Квадрат*, у которого будет только одно поле *Сторона*. Такая модифицированная программа приведена ниже.

```
// p13016
type
  Прямоугольник = auto class
    Длина: real;
    Ширина: real;
    function Площадь: real; virtual := Длина * Ширина;
    procedure Вывести; virtual :=
        $'{Длина} x {Ширина} = {Площадь}'.Println;
  end;

  Квадрат = class(Прямоугольник)
    Сторона: real;
    constructor(Сторона: real);
    begin
      Self.Сторона := Сторона;
    end;
    function Площадь: real; override := Sqr(Сторона);
    procedure Вывести; override :=
        $'{Сторона} x {Сторона} = {Площадь}'.Println;
  end;

begin
  var L := new List<Прямоугольник>;
  L.Add(new Прямоугольник(12, 15.5));
  L.Add(new Прямоугольник(18.2, 9.4));
  L.Add(new Квадрат(15));
  foreach var p in L do
    p.Вывести
end.
```

Пришлось вспомнить о виртуальных методах. Но иначе невозможно обеспечить полиморфизм! Иначе не удастся, имея переменную *p* с типом *Прямоугольник*, вызывать для случая ссылки на объект класса *Квадрат* нужные методы.

От проблемы получения и вывода неверной площади квадратов мы избавились. А вот проблема, связанная с наличием у объекта класса *Квадрат* пустых полей *Длина* и *Ширина* (да, инициализированных нулями, но от этого не легче) никуда не делась. В более крупных проектах это может сыграть злую шутку (и обычно происходит такое очень некстати). Всему виной неверная абстракция. Это в школьной геометрии квадрат является частным случаем прямоугольника, а в ООП квадрат – это никак не наследник прямоугольника. Конечно, мы можем попытаться спасти ситуацию, закрыв поля и введя вместо них открытые свойства, у которых в поведении по записи (в сеттерах) указать нужные модификации полей – мы же недаром столько всего изучили! Но, во-первых, это сильно усложнит код, а во-вторых сделает классы более «тяжелыми», содержащими совершенно лишние члены. И, скорее всего, лишние поля все равно будут где-то болтаться.

Попробуем наследовать классы Прямоугольник и Квадрат от некоторого третьего класса, содержащего общности обоих классов-потомков. Назовем этот класс *Фигура*. А какие там могут быть общности, если у квадрата задается длина стороны, а у

прямоугольника – длина и ширина? Но ведь мы сказали: «абстракция». Абстрактные методы нас спасут! В них не надо озадачиваться параметрами и прочими деталями, они лишь говорят: «Такой метод у класса есть». Они – протокол, декларация, интерфейс.

```
// p13017
type
  Фигура = class
    function Площадь: real; abstract;
    procedure Вывести; abstract;
  end;

  Прямоугольник = class(Фигура)
    Длина: real;
    Ширина: real;
    constructor(Длина, Ширина: real);
    begin
      Self.Длина := Длина;
      Self.Ширина := Ширина
    end;
    function Площадь: real; override := Длина * Ширина;
    procedure Вывести; override :=
        $'{Длина} x {Ширина} = {Площадь}'.Println;
  end;

  Квадрат = class(Фигура)
    Сторона: real;
    constructor(Сторона: real);
    begin
      Self.Сторона := Сторона;
    end;
    function Площадь: real; override := Sqr(Сторона);
    procedure Вывести; override :=
        $'{Сторона} x {Сторона} = {Площадь}'.Println;
  end;

begin
  var L := new List<Фигура>;
  L.Add(new Прямоугольник(12, 15.5));
  L.Add(new Прямоугольник(18.2, 9.4));
  L.Add(new Квадрат(15));
  foreach var p in L do
    p.Вывести
end.
```

Вот же оно – счастье программиста! Теперь у нас имеется расширяемый набор классов, потому что от абстрактного класса *Фигура* можно наследовать любые классы, для которых вычисляется площадь. Хоть криволинейные контуры! Это никак не затронет кода остальных классов, чего мы все время пытались добиться. Свершилось! Давайте на радостях добавим класс для кругов.

```
// p13018
type
  Фигура = class
    function Площадь: real; abstract;
    procedure Вывести; abstract;
  end;

  Прямоугольник = class(Фигура)
    Длина: real;
    Ширина: real;
    constructor(Длина, Ширина: real);
    begin
      Self.Длина := Длина;
      Self.Ширина := Ширина
    end;
    function Площадь: real; override := Длина * Ширина;
    procedure Вывести; override :=
        $'{Длина} x {Ширина} = {Площадь}'.Println;
  end;

  Квадрат = class(Фигура)
    Сторона: real;
    constructor(Сторона: real);
    begin
      Self.Сторона := Сторона;
    end;
    function Площадь: real; override := Sqr(Сторона);
    procedure Вывести; override :=
        $'{Сторона} x {Сторона} = {Площадь}'.Println;
  end;

  Круг = class(Фигура)
    Радиус: real;
    constructor(Радиус: real);
    begin
      Self.Радиус := Радиус;
    end;
    function Площадь: real; override := Pi * Sqr(Радиус);
    procedure Вывести; override :=
        $'Pi x {Радиус}^2 = {Площадь}'.Println;
  end;

begin
  var L := new List<Фигура>;
  L.Add(new Прямоугольник(12, 15.5));
  L.Add(new Прямоугольник(18.2, 9.4));
  L.Add(new Квадрат(15));
  L.Add(new Круг(23.27));
  foreach var p in L do
    p.Вывести
end.
```

Осталось проверить работу программы:

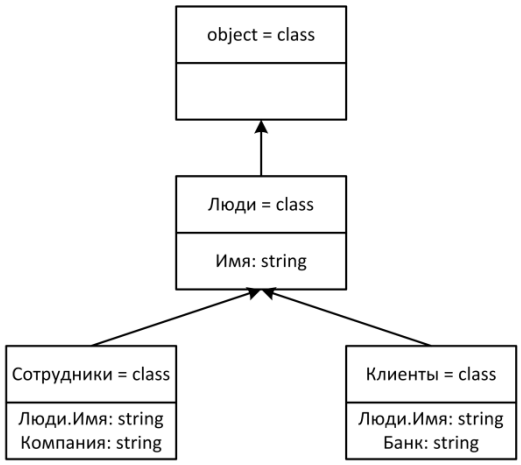
```
12 x 15.5 = 186
18.2 x 9.4 = 171.08
15 x 15 = 225
Pi x 23.27^2 = 1701.15011661103
```

13.4 Операции as и is

Пусть имеется переменная некоторого типа в цепочке унаследованных классов. Можно ли привести переменную к некоторому другому типу в этой цепочке?

Приведение типа потомка к родительскому (апкаст) возможно всегда. Приведение родительского типа к типу потомка (даункаст) возможно не всегда.

Рассмотрим пример. Класс *Люди* имеет двух потомков – класс *Сотрудники* и класс *Клиенты*. У него также имеется неявный родитель – класс *object*, от которого ведут наследование все классы PascalABC.NET и который описывать не нужно.



Класс *Люди* имеет открытое поле *Имя*, хранящее имя человека. Это логично: и *Сотрудники*, и *Клиенты* всегда имеют имя, поэтому нет нужды организовывать такое поле в каждом классе, проще его унаследовать.

У класса *Сотрудники* имеется открытое поле *Компания* для наименования компании, в которой работает сотрудник.

Класс *Клиенты* содержит открытое поле *Банк* с наименованием банка, в котором у клиента открыт счет.

Имеются две цепочки наследований:

- *object* ← *Люди* ← *Сотрудники*;
- *object* ← *Люди* ← *Клиенты*.

Эти цепочки делают возможными следующие апкасты: *Люди* ← *Сотрудники*, *Люди* ← *Клиенты*, *object* ← *Люди*, *object* ← *Сотрудники*, *object* ← *Клиенты*. Почему апкасты всегда допустимы? Любой потомок содержит в себе все, что содержит родитель, но в дополнение также может содержать еще что-то. При апкасте это «дополнение» отсекается.

```
// p13019
type
  Люди = auto class
    Имя: string;
  end;

  Сотрудники = class(Люди)
    Компания: string;
    constructor(Имя, Компания: string) :=
        (Self.Имя, Self.Компания) :=(Имя, Компания);
  end;

  Клиенты = class(Люди)
    Банк: string;
    constructor(Банк, Имя: string) :=
        (Self.Банк, Self.Имя) :=(Банк, Имя);
  end;

begin
  var Сотрудник := new Сотрудники('Шура','Рога и копыта');
  var Человек: Люди := Сотрудник; // от Сотрудники к Люди
  //Println($'Человек: {Человек.Имя}, {Человек.Компания}');
  Println($'Человек: {Человек.Имя}');
  Println($'Сотрудник: {Сотрудник.Имя}, "{Сотрудник.Компания}"');
end.
```

Создается объект класса *Сотрудники*, в его поле *Имя* записывается строка «Шура», в поле *Компания* – строка «Рога и копыта». Ссылка на созданный объект помещается в переменную *Сотрудник*, а сама переменная получает тип *Сотрудники*. Далее описывается переменная *Человек* с типом *Люди*. Приведение типа *Сотрудники* к типу *Люди* в данном случае является апкастом. Апкаст происходит путем присваивания – помещения ссылки из переменной *Сотрудник* в переменную *Человек*. После такой операции обе переменные будут ссылаться на один и тот же объект. «Отсечение», о котором говорилось выше, происходит на логическом уровне и выражается в том, что из переменной *Человек* поле *Компания* доступно не будет по причине своего отсутствия и в этом классе, и в его предках (*object*). Поле *Компания* описано в классе-потомке *Сотрудник*, о существовании которого класс-родитель даже не догадывается. Почему? Так класс – он же всего лишь «чертеж», а не объект. Чертеж один раз создают и впоследствии не меняют. Но могут брать за основу для новых чертежей – это и есть наследование. Закомментированная строка в программе содержит пример обращения к несуществующему полю. Снимите комментарий и убедитесь, что пресловутое «отсечение» имело место: *Человек* не видит поле *Компания*.

Апкаст, показанный в примере – случай ***неявного приведения типа***. Тип левой части компилятор приводит к типу правой части. Существует также операция ***явного приведения типа***, которая имеет вид

```
ТипДляПриведения(ПриводимаяПеременная)
```

Например, для явного апкаста мы могли бы записать

```
var Человек := Люди(Сотрудник).
```

Необходимым условием выполнимости операции **приведения типа** является принадлежность типа приводимой переменной и типа, к которому происходит приведение, к одной цепочке наследования. В случае **апкаста** это условие одновременно является и **достаточным**. Нарушение необходимого условия фиксируется компилятором.

Какая разница между использованием явного и неявного приведения? Невозможность неявного приведения обнаруживает компилятор. Невозможность явного приведения обнаруживается во время выполнения программы, после чего генерируется исключение «Не удалось привести тип объекта "xxx" к типу "yyy"».

Теперь рассмотрим ситуацию с даункастом. Поскольку объект, принадлежащий к родительскому классу, не имеет никакой информации о потомках, в общем случае даункаст невозможен. Не может появиться в таком объекте поле, метод или свойство, потому что просто неоткуда ему появиться. И все же, в некоторых случаях даункаст сделать можно.

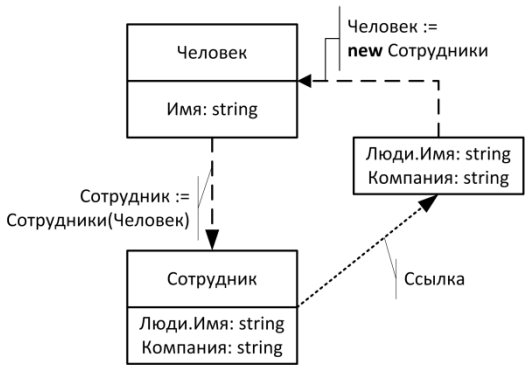
Еще раз: что произойдет, если мы сделаем апкаст и сохраним в переменной родительского типа ссылку на объект унаследованного класса? При доступе к объекту через такую переменную нам не будут видны члены, отсутствующие в родительском классе. Но в самом объекте они будут. Можно ли теперь сделать приведение назад, к унаследованному классу, если мы знаем, что переменная ссылается на объект, изначально принадлежащий к этому же, унаследованному классу?

Ответ на вопрос получим экспериментальным путем. Допустим, с помощью неявного апкаста объект *Сотрудник* класса *Сотрудники* был приведен к типу *Люди*. Результат приведения помещен в переменную *Человек*, имеющую тип *Люди*. Теперь попробуем выполнить явный даункаст, приведя объект, на который ссылается переменная *Человек*, к типу *Сотрудники*. Результат поместим в переменную *Сотрудник*, имеющую тип *Сотрудники*.

```
// p13020
type
  Люди = auto class
    Имя: string;
  end;

  Сотрудники = class(Люди)
    Компания: string;
    constructor(Имя, Компания: string) :=
        (Self.Имя, Self.Компания) :=(Имя, Компания);
  end;

  Клиенты = class(Люди)
    Банк: string;
    constructor(Банк, Имя: string) :=
        (Self.Банк, Self.Имя) :=(Банк, Имя);
  end;

begin
  var Сотрудник: Сотрудники;
  var Человек: Люди;
  Человек := new Сотрудники('Шура','Рога и копыта');
  Сотрудник := Сотрудники(Человек);
  Println($'Человек: {Человек.Имя}');
  Println($'Сотрудник: {Сотрудник.Имя}, "{Сотрудник.Компания}"');
end.
```

Посмотрим, что выведет эта программа.

```
Человек: Шура
Сотрудник: Шура, "Рога и копыта"
```



Логически, переменная *Человек* не имеет поля *Компания*, поэтому после сохранения в ней объекта типа *Сотрудники* значение этого поля через переменную *Человек* недоступно. После приведения типа переменной *Человек* к типу *Сотрудники*, имеющему поле *Компания*, доступ к этому полю появляется. В реальности созданный в памяти объект не изменяется. Вначале ссылка на него размещается в переменной *Человек*, а затем копируется в переменную *Сотрудник*. Обе переменные указывают на один и тот же объект, но из переменной *Сотрудник* поле *Компания* видно, а из переменной *Человек* – нет.

Можно ли здесь было воспользоваться неявным приведением типа при даункасте, написав Сотрудник := Человек ? Такая операция, как мы знаем, для ссылочного типа выполняется путем копирования ссылки, а ведь нам нужно именно это. Но

нет: компилятор считает такое присваивание ошибочным и выдает сообщение «Нельзя преобразовать тип Люди к Сотрудники». И правильно, потому что в общем-то случае нельзя, а типы переменных предварительно были описаны. Явное приведение типа откладывает решение вопроса о возможности проведения этой операции до момента выполнения, когда ситуация прояснится. И либо все получится, либо сгенерируется исключение. Утешительно звучит, не правда ли?

> Даункаст при неявном приведении типа невозможен и запрещен на уровне компиляции программы.

Как же избежать генерации исключения по ошибке при явном даункасте или преодолеть ее последствия? Обработка посредством **try** – **catch** – громоздкое решение. Можно попробовать сопоставление с образцом (pattern matching), с котором мы будем знакомиться позднее. Но самый простой вариант – использовать придуманные как раз для таких случаев операции **as** и **is**.

13.4.1 Операция is

Операция is позволяет проверить, имеет ли переменная указанный динамический тип. Конструкция *a* **is** *Tun* возвращает **True**, если переменную *a* можно привести к *Tun* и **False** в противном случае. Оба операнда должны иметь классовый тип.

Используя операцию **is**, можно написать код, защищенный от возникновения исключения при даункасте:

```
if Человек is Сотрудники then
begin
  Сотрудник := Сотрудники(Человек);
  Println($'Человек: {Человек.Имя}');
  Println($'Сотрудник: {Сотрудник.Имя}, "{Сотрудник.Компания}"')
end;
```

Имеется также расширенная запись операции *a* **is** *Tun* (**var** *b*), в которой при возврате **True** происходит приведение *a* к *b*. Это позволяет написать более короткий код:

```
if Человек is Сотрудники (var Сотрудник) then
begin
  Println($'Человек: {Человек.Имя}');
  Println($'Сотрудник: {Сотрудник.Имя}, "{Сотрудник.Компания}"')
end;
```

Не пытайтесь писать что-то вроде *10* ***is real***: компилятор зафиксирует ошибку, поскольку типы обоих операндов должны совпадать или быть связаны цепочкой наследований, да и вообще 10 – это литерал, а не классовая переменная. Но можно написать

```
begin
  var a: object;
  a := 10;
  Println(a is real); // False
end.
```

13.4.2 Операция as

Допустим, что с помощью операции **is** мы убедились в возможности безопасного даункаста. Теперь следует позаботиться о безопасности самого процесса приведения. Специально с этой целью в PascalABC.NET включена операция **as** следующего формата:

```
Переменная as ТипКласса
```

Оператор **as** позволяет рассматривать объект одного класса (*Переменная*), как принадлежащий другому классу (*ТипКласса*) и совместимый по присваиванию. Если такое преобразование возможно, то приведение выполняется и операция возвращает ссылку на приведенный объект. В противном случае операция вернет **nil**. Операция **as** делает возможным **неявный даункаст**.

```
// p13021
type
  Люди = auto class
    Имя: string;
  end;

  Сотрудники = class(Люди)
    Компания: string;
    constructor(Имя, Компания: string) :=
        (Self.Имя, Self.Компания) :=(Имя, Компания);
  end;

  Клиенты = class(Люди)
    Банк: string;
    constructor(Банк, Имя: string) :=
        (Self.Банк, Self.Имя) :=(Банк, Имя);
  end;
```

```
begin
  var Сотрудник: Сотрудники;
  var Человек: Люди;
  Человек := new Сотрудники('Шура','Рога и копыта');
  if Человек is Сотрудники then
  begin
    Сотрудник := Сотрудники(Человек);
    Println($'Человек: {Человек.Имя}');
    Println($'Сотрудник: {Сотрудник.Имя}, "{Сотрудник.Компания}"')
  end;
  Человек := new Клиенты('СкруджБанк', 'МакДак');
  var Клиент: Клиенты := Человек as Клиенты;
  if Клиент <> nil then
    Println($'Клиент: {Клиент.Имя}, "{Клиент.Банк}"')
  else
    Println($'Человек: {Человек.Имя}')
end.
```

Приведенная программа выводит следующие строки:

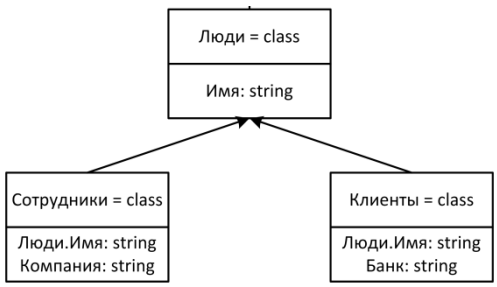
```
Человек: Шура
Сотрудник: Шура, "Рога и копыта"
Клиент: МакДак, "СкруджБанк"
```

У операции **as** есть еще одно применение – обращение из объектов одного класса к членам совместимого по присваиванию класса. Так, для приведенного выше примера можно написать

```
Человек := new Клиенты('СкруджБанк', 'МакДак');
if Человек is Клиенты then
  Println((Человек as Клиенты).Банк)
```

13.5 Сопоставление с образцом

Сопоставление с образцом (англ. Pattern matching) – это метод, в котором действия выполняются в зависимости от результата сопоставлении некоторого объекта с заданным образцом. Из-за внешней схожести структуры оператора сопоставления с оператором **case**, можно встретить разговорный термин «case по классам».



Еще раз обратимся к рассмотренному выше примеру. В переменной типа *Люди* вследствие апкаста могут находиться объекты как типа *Сотрудники*, так и типа *Клиенты*. Если создать контейнер и наполнить его элементами типа *Люди*, при последующем просмотре содержимого этого контейнера может понадобиться знание о том, с каким конкретно объектом мы имеем дело. Нас может выручить операция **is**.

```
begin
  var ВсеВместе := new List<Люди>;
  ВсеВместе.Add(new Сотрудники('Шура', 'Рога и копыта'));
  ВсеВместе.Add(new Клиенты('СкруджБанк', 'МакДак'));
  ВсеВместе.Add(new Клиенты('ЛунтикБанк', 'Лошарик'));
  foreach var Каждый in ВсеВместе do
    if Каждый is Сотрудники then
      Println(Каждый.Имя, 'С')
    else
      if Каждый is Клиенты then
        Println(Каждый.Имя, 'К')
end.
```

Что же, возможное решение найдено. Но... а если у нас будет три класса? Пять? Десять? Придется писать чудовищное нагромождение условных операторов. И кто же потом согласится назвать это пресловутым «современным программированием»? Конечно, выход есть, если вы имеете дело с собственной задачей, которую пишете с нуля. Добавьте в каждом классе поле порядкового типа, идентифицирующее класс. И потом пишите **case**. Но если у вас работа с готовыми объектами, такой подход не годится.

Для организации подобных ветвлений и существует метод сопоставления с образцом на основе оператора **match**.

```
match ИмяПеременной with
  Класс1(var Имя1): Оператор1;
  Класс2(var Имя2): Оператор2;
  . . .
  КлассN(var ИмяN): ОператорN;
end
```

Есть большое сходство с **case**, не так ли? Переменная, для которой указывается *ИмяПеременной* содержит ссылку на объект. Для каждого имени класса указывается параметр (*Имя1*, *Имя2*, ...). Это формальные имена, сродни именам в лямбда-выражениях, и они могут совпадать. Изюминка в том, что *ИмяХ* выступает в *ОператореХ* в качестве объекта *КлассаХ*. А *Оператор*, конечно же, может быть и блоком. Ключевое слово **var** разрешается опускать.

```
begin
  var ВсеВместе := new List<Люди>;
  ВсеВместе.Add(new Сотрудники('Шура', 'Рога и копыта'));
  ВсеВместе.Add(new Клиенты('СкруджБанк', 'МакДак'));
  ВсеВместе.Add(new Клиенты('ЛунтикБанк', 'Лошарик'));
  foreach var Каждый in ВсеВместе do
    match Каждый with
      Сотрудники(v): Println(v.Имя, v.Компания);
      Клиенты(v): Println(v.Имя, v.Банк);
    end
end.
```

Мощное, наглядное и красивое решение.

При сопоставлении с образцом имеется возможность указывать фразу с ключевым словом **when**, определяющую условие, при истинности которого будет выполняться указанный в ветке оператор:

```
match ИмяПеременной with
  Класс1(var Имя1) when Условие1: Оператор1;
  Класс2(var Имя2) when Условие2: Оператор2;
  . . .
  КлассN(var ИмяN) when УсловиеN: ОператорN;
end
```

13.6 Деконструкторы

Деконструктор – это метод-процедура с фиксированным именем Deconstruct. Этот метод можно включить в класс с тем, чтобы получить значения желаемого набора его полей в операторах **is** и **match**. В автоклассах метод Deconstruct создается автоматически. Все параметры метода должны указываться с ключевым словом **var**.

Чтобы понять, в каких случаях деконструкторы могут быть полезны, можно воспринимать их, как своего рода синтаксический сахар для операций **is** и **match**.

Вот так могла бы выглядеть программа p13021 (см. 13.4.2), использующая деконструктор.

```
// p13022
type
  Люди = auto class
    Имя: string;
  end;

  Сотрудники = class(Люди)
    Компания: string;
    constructor(Имя, Компания: string) :=
        (Self.Имя, Self.Компания) := (Имя, Компания);
    procedure Deconstruct(var Имя, Компания: string) :=
        (Имя, Компания) := (Self.Имя, Self.Компания);
  end;

  Клиенты = class(Люди)
    Банк: string;
    constructor(Банк, Имя: string) :=
        (Self.Банк, Self.Имя) := (Банк, Имя);
  end;
```

```
begin
  var Человек: Люди := new Сотрудники('Шура','Рога и копыта');
  if Человек is Сотрудники(var Имя, var Компания) then
  begin
    Println($'Человек: {Человек.Имя}');
    Println($'Сотрудник: {Имя}, "{Компания}"')
  end
end.
```

А для задачи из 13.5 можно было бы написать примерно такое решение:

```
// p13023
type
  Люди = auto class
    Имя: string;
  end;

  Сотрудники = class(Люди)
    Компания: string;
    constructor(Имя, Компания: string) :=
        (Self.Имя, Self.Компания) := (Имя, Компания);
    procedure Deconstruct(var Имя, Компания: string) :=
        (Имя, Компания) := (Self.Имя, Self.Компания);
  end;

  Клиенты = class(Люди)
    Банк: string;
    constructor(Банк, Имя: string) :=
        (Self.Банк, Self.Имя) := (Банк, Имя);
    procedure Deconstruct(var Имя, Банк: string) :=
        (Банк, Имя) := (Self.Банк, Self.Имя);
  end;

begin
  var ВсеВместе := new List<Люди>;
  ВсеВместе.Add(new Сотрудники('Шура', 'Рога и копыта'));
  ВсеВместе.Add(new Клиенты('СкруджБанк', 'МакДак'));
  ВсеВместе.Add(new Клиенты('ЛунтикБанк', 'Лошарик'));
  foreach var Каждый in ВсеВместе do
    match Каждый with
      Сотрудники(Имя, Компания): Println(Имя, Компания);
      Клиенты(Имя, Банк): Println(Имя, Банк);
    end
end.
```

Программа выведет следующие строки:

```
Шура Рога и копыта
МакДак СкруджБанк
Лошарик ЛунтикБанк
```

Деконструкторы могут быть добавлены к уже имеющимся классам при помощи **методов расширения** (см. 13.8.1, программа p13027). Использовать в написании программ деконструкторы или нет – решать вам.

13.7 Анонимные классы

Как известно, лямбда-выражения позволяют формировать функции и процедуры по мере возникновения надобности и без их предварительного описания. Подобным же образом, анонимные классы предназначены для создания объектов классов, для которых описание не создавалось. Такие классы не имеют имени (поэтому они и называются анонимными), зато имеют набор полей. Объект, созданный на основе анонимного класса, будет содержать набор открытых полей, которые перечисляются в конструкторе класса.

Объект на основе анонимного класса можно создать, используя оператор вида

```
var Имя := new class(поле1 := значение1, поле2 := значение2, …);
```

Если переменные, имена которых должны быть указаны для формирования имен полей в создаваемом объекте, предварительно были определены и получили значения, то можно использовать более короткий формат записи:

```
var поле1 := значение1;
var поле2 := значение2;
var Имя := new class(поле1, поле2, …);
```

В случае, когда при записи имени переменной используется точечная нотация, именем поля станет самое последнее (правое) из имен, разделенных точками.

Анонимный класс можно использовать для заполнения контейнеров объектами, но сравнивать элементы такого контейнера будет проблематично из-за того, что при сравнении объектов сравниваются ссылки на них, а не содержимое.

```
begin
  var L := new List<object>;
  L.Add(new class(x := 3.5, y := -2.8));
  L.Add(new class(x := -4.1, y := 3.11));
  L.PrintLn // (3.5,-2.8) (-4.1,3.11)
end.
```

Беда здесь в том, что элементом списка *L[i]* будет объект типа Object и доступ к его бывшим полям x и y будет невозможен, поэтому полезность такого использования анонимных классов очень сомнительна.

Анонимный класс можно использовать, чтобы одной строкой ввести в программу и инициализировать несколько переменных. Например, вот так можно компактно записать программу решения квадратного уравнения:

```
begin
  var k := new class(a := 3.5, b := -4.2, c := -2.18);
  var d := k.b ** 2 - 4 * k.a * k.c;
  if d > 0 then
  begin
    d := Sqrt(d);
    $'x1={(-k.b-d)/2/k.a}, x2={(-k.b+d)/2/k.a}'.Println
  end
  else
    if d = 0 then
      $'{-k.b/2/k.a}'.Println
    else
      Println('нет корней')
end.
```

В данном случае будет получен результат

```
x1=-0.391391518451284, x2=1.59139151845128
```

Еще один пример использования анонимных классов посвящается тем, кто не любит работать с индексами кортежей. Вначале посмотрим, как выглядит программа поиска наиболее удаленной от начала координат точки, записанная без применения объектно-ориентированного программирования.

```
begin
  var Точки := new List<(integer, integer)>;
  loop ReadInteger('Укажите количество точек:') do
    Точки.Add(Random2(-20, 20));
  Точки.Println;
  var Дальняя := Точки.Select(Точка ->
      (Точка, Точка[0] ** 2 + Точка[1] ** 2)).
      MaxBy(Точка -> Точка[1]);
  Writeln('Точка (', Дальняя[0][0], ', ', Дальняя[0][1],
      ') имеет максимальное удаление от начала координат ',
      Sqrt(Дальняя[1]))
end.
```

Просто «кортеж на кортеже»! Доходит до использования двойного индекса (наподобие *Дальняя[0][0]*), что никак не прибавляет программе прозрачности. Контейнер List хранит двухэлементные кортежи, представляющие собой координаты x и y точек. Элементы контейнера проецируются в последовательность двухэлементных кортежей. Первым элементом результирующего кортежа будет исходный кортеж, а вторым – квадрат расстояния от начала координат. Переменная Дальняя хранит в виде кортежа сведения о максимально удаленной точке.

Посмотрим теперь, что можно сделать с помощью классов.

```
// p13024
type
  Точка = auto class
    x, y: integer;
    constructor :=
        (x, y) := Random2(-20, 20);
    static function Случайная := new Точка;
end;

begin
  var Точки := new List<Точка>;
  var КоличествоТочек := ReadInteger('Укажите количество точек:');
  loop КоличествоТочек do
    Точки.Add(Точка.Случайная);
  Точки.Println;
  var Дальняя := Точки.Select(Точка ->
      new class(x := Точка.x, y := Точка.y,
      Дальность := Sqr(Точка.x) + Sqr(Точка.y))).
      MaxBy(Точка -> Точка.Дальность);
  Writeln('Точка (', Дальняя.x, ', ', Дальняя.y,
      ') имеет максимальное удаление от начала координат ',
      Sqrt(Дальняя.Дальность))
end.
```

Обе программы дают выдачу одинакового вида:

```
Укажите количество точек: 7
(9,-11) (17,3) (-20,-2) (-13,-11) (-8,-8) (-8,20) (8,3)
Точка (-8, 20) имеет максимальное удаление от начала координат 21.540659228538
```

Автокласс *Точка* позволил размещать в контейнере объекты, содержащие два поля с информативными именами. Почему именно автокласс? Чтобы не писать метод для вывода полей класса. Статическая функция позволила убрать в тело класса малоинформативное ***new*** *Точка*, заменив его более наглядным *Точка.Случайная*. Абстрактный класс, использованный при проецировании, позволил работать с полями *x*, *y* и *Дальность* вместо элементов кортежа достаточно сложной структуры. Длиннее получилось? Ну что же, не всегда «краткость – сестра таланта».

13.8 Методы расширения

Любой существующий тип данных, независимо от того, является ли он пользовательским или базовым, хранится ли в стандартной библиотеке Microsoft .NET или во внешней библиотеке .dll, можно ***расширить***, добавив ***методы расширения***. Использование методов расширения особенно актуально в случаях, когда недоступен исходный код класса, реализующего тип данных.

Метод расширения (также используется термин ***расширение***) – дополнительный, определяемый пользователем метод, имеющий модификатор **extensionmethod**. Первый параметр в заголовке метода обязательно должен иметь имя **Self**, а его тип

должен принадлежать расширяемому типу. При встраивании расширения в цепочку точечной нотации считается, что такой параметр является значением выражения, находящегося левее предшествующей имени расширения точке. Сохраняется также возможность обращения к методу более традиционным путем, указывая имя метода с последующим набором всех параметров.

13.8.1 Переменная Self

Неоднократно упоминалось, что **Self** – это некоторое имя, которое компилятор «знает» и ассоциирует с объектом. Переменная с именем **Self** определена внутри любого метода, не являющегося статическим. Она является ссылкой на объект, который вызывает данный метод, т.е. подменяет собой имя объекта, которого на этапе описания класса еще не создано. Например, когда в конструкторе имя параметра совпадает с именем поля класса, переменная **Self** позволяет уточнить, где именно используется имя поля.

```
type
  A = class
    i: integer;
    constructor (i: integer);
    begin
      Self.i := i;
    end;
  end;
```

В приведенном выше примере при обращении к конструктору создается объект, имя которого будет ассоциировано с переменной **Self**. Поэтому обращение ***Self.i*** означает поле класса, а не параметр конструктора.

Переменная Self играет важную роль при написании методов, позволяя встраивать их вызов в цепочки с точечной нотацией. В этом случае Self понимается как все, что находится по цепочке с вызовом левее имени метода.

```
type
  ПростаяДробь = class
  public
    Числитель, Знаменатель: integer;
    constructor(Числ, Знам: integer);
    begin
      (Числитель, Знаменатель) := (Числ, Знам)
    end;
  end;

procedure Вывод(Self: ПростаяДробь); extensionmethod;
begin
  $'{Self.Числитель}/{Self.Знаменатель}'.Println
end;

begin
  (new ПростаяДробь(-5, 19)).Вывод // -5/19
end.
```

В приведенном примере показано, как используется Self при написании метода, предназначенного для использования в точечной нотации. В данном случае она подменяет собой параметр процедуры.

Рассмотрим еще один пример.

```
// p13025
type
  Точка = auto class
  public
    x, y: real;
  end;

function Расстояние(Self, Точка2: Точка): real; extensionmethod;
begin
  Result := Sqrt(Sqr(Точка2.x - Self.x) + Sqr(Точка2.y - Self.y))
end;

begin
  var p1 := new Точка(-5, 10.6);
  var p2 := new Точка(2.7, 0);
  p1.Расстояние(p2).Println // 13.1015266286032
end.
```

Методы расширения являются глобальными и по этой причине не могут быть описаны в теле класса. Обязательное требование для **Self** иметь тип расширяемого класса позволяет ассоциировать тип этого параметра с расширяемым классом. В приведенном примере **Self** имеет тип *Точка*, поэтому для переменной *p1*, имеющей такой же тип, метод *Расстояние* будет доступен.

Третий пример иллюстрирует расширение для **integer** – базового класса PascalABC.NET. Здесь используется еще один известный принцип ООП, называемый Принципом Открытости/Закрытости (OCP – Open/Close Principle). В вольном изложении он говорит, что к классу можно добавлять новые методы без доступа к его исходному коду. Фактически, мы наследуем от класса и меняем унаследованное.

```
// p13026
function Cube(Self: integer): integer;
    extensionmethod := Sqr(Self)*Self;
begin
  5.Cube.Println; // 125
end.
```

Четвертый пример показывает, как можно добавить деконструктор нужного вида в базовый класс **real**.

```
// p13027
procedure Deconstruct(Self: real; var Целая: integer; var Дробная: real);
    extensionmethod := (Целая, Дробная) :=
        (Trunc(Self), Round(Frac(Self),14));

begin
  foreach var r in Seq(-18.345, 0.0, 14.71293) do
  begin
    if r is real(var Цел, var Дроб) then
      $'{r} = {Цел} + {Дроб}'.Println;
    match r with
      real(Цел, Дроб) when r < 0: $'{r} = {Цел} - {-Дроб}'.Println;
      real(Цел, Дроб) when r = 0: r.Println;
      real(Цел, Дроб) when r > 0: $'{r} = {Цел} + {Дроб}'.Println
    end
  end
end.
```

Вывод результатов показывает, что вариант с is требует доработки.

```
-18.345 = -18 + -0.345
-18.345 = -18 - 0.345
0 = 0 + 0
0
14.71293 = 14 + 0.71293
14.71293 = 14 + 0.71293
```

Методы расширения не могут быть виртуальными. Если имя метода расширения совпадет с именем обычного метода, то будет вызван обычный метод.

13.8.2 Перегрузка операций посредством расширения

В отличие от методов расширений, при перегрузке операций использование имени Self не требуется. Условия, которые необходимо соблюдать при перегрузке операций описаны в 13.2.2. Посмотрите, как будет выглядеть при перегрузке операций посредством расширения вариант программы p13009:

```
// p13028
type
  Дробь = class
  public
    Числитель: integer;
    Знаменатель := 1;

    constructor(Чис, Знам: integer) :=
        (Числитель, Знаменатель) := (Чис, Знам);

    constructor(Чис: integer):= Числитель := Чис;

    function ToString: string; override :=
        $'{Числитель}/{Знаменатель}';

end;

function operator -(a: Дробь); extensionmethod :=
    new Дробь(-a.Числитель, a.Знаменатель);

function operator +(a, b: Дробь); extensionmethod :=
    new Дробь(a.Числитель * b.Знаменатель +
    b.Числитель * a.Знаменатель,
    a.Знаменатель * b.Знаменатель);

function operator *(a, b: Дробь); extensionmethod :=
    new Дробь(a.Числитель * b.Числитель,
    a.Знаменатель * b.Знаменатель);

function operator /(a, b: Дробь); extensionmethod :=
    new Дробь(a.Числитель * b.Знаменатель,
    b.Числитель * a.Знаменатель);

begin
  var a := new Дробь(3, 4);
  var b := new Дробь(-2, 5);
  var c := (a + b) / (a + (-b));
  Writeln(c) // 140/460
end.
```

Вы можете выбрать любой из способов перегрузки операторов. Перегрузка при помощи расширения не требует менять код в теле класса.

13.9 Интерфейсы

Давайте немного вспомним модули PascalABC.NET. Создавая модуль, мы имели возможность отделить в его разделах интерфейс от реализации. Это давало возможность скрыть реализацию от посторонних глаз, но при этом показать интерфейс. Интерфейс, как договор о том, что необходимо передать модулю для выполнения заранее оговоренного набора функций.

В ООП интерфейс – это тоже договор, точнее протокол, определяющий набор методов и свойств, которые должны быть реализованы в каждом классе, поддерживающем данный интерфейс. Класс, о котором сказано, что он реализует некоторый интерфейс, должен полностью реализовывать каждый его компонент. Если хотя бы один компонент интерфейса не будет реализован, компилятор зафиксирует ошибку. И это отличный способ контроля используемых классов в больших проектах, особенно когда в них задействовано несколько разработчиков. Класс, реализующий интерфейс, может объявить методы абстрактными и передать их реализацию классам-потомкам.

Интерфейс – замечательный (и, к тому же, единственный) способ объединить классы, не унаследованные от общего предка (кроме, конечно, System.Object). Он позволяет выполнить объединение по формальному признаку – наличию одинаковых свойств и/или методов. Например, что объединяет диван, чемодан и маленькую собачонку? Как считал С. Я. Маршак, – дама, которая их сдавала в багаж. Ведь это были предметы, предназначенные для перевозки. Их общий интерфейс должен описывать вес и габариты. Занеся предметы в некоторый контейнер и впоследствии их перебрав, мы получим возможность оценить общую стоимость перевозки согласно некоторым тарифам. Вспомните задачу p13017 (см. 13.3.2): там мы пытались «скрестить ужа с ежом», разыскивая общее у прямоугольника, заданного шириной и высотой, и квадрата, заданного стороной. Выход из положения нашелся в наследовании обеих фигур от абстрактного класса. Но, имея соответствующие знания, мы могли бы вместо наследования описать и затем реализовать в каждом классе интерфейс, обеспечивающий вычисление площади.

Реализовывать в создаваемых классах интерфейс могут заставить обстоятельства. В 10.6 уже упоминалось, что для включения в коллекции объекты должны поддерживать определенные методы интерфейсов Microsoft .NET с именами IComparable или IComparer. Там же был дан пример написания простейшего компаратора, реализующего требуемый интерфейс. Сейчас есть хороший повод еще раз вернуться к этому примеру: ваших знаний ООП уже достаточно, чтобы его понять. А заодно – посмотреть пример реализации интерфейса в классе.

13.9.1 Описание интерфейса

В PascalABC.NET интерфейс является набором данных ссылочного типа. Он описывается в секции **type** в формате, похожем на описание класса, только вместо слова **class** указывается **interface**:

```
ИмяИнтерфейса = interface
  Объявление_методов_и_свойств
end;
```

Имя интерфейса традиционно начинают с прописной латинской буквы I (произносится «ай», поскольку именно так эта буква называется в английском алфавите), чтобы использовать некоторую игру слов. В английском языке I означает местоимение «я», поэтому интерфейс с именем IComparable (comparable – англ. сравнимый, сопоставимый) как бы сообщает нам: «Я сравним». В Microsoft .NET имеется

большое количество стандартных интерфейсов, но все же постарайтесь выбирать имена, не совпадающие с ними. На достаточно популярных интерфейсах IComparable, IComparer и IClonable мы остановимся отдельно.

Интерфейс аналогичен абстрактному классу, содержащему абстрактные методы. Для каждого метода описывается его заголовок, для каждого свойства – возвращаемый тип и модификаторы доступа read/write. Поскольку интерфейс не может содержать реализацию (т.е. операторы, реализующие члены интерфейса), в нем нельзя определять поля и статические члены. Для членов интерфейса не допускается указывать модификатор видимости. По умолчанию используется модификатор **public**: интерфейс всегда должен быть открытым. Приведенный в качестве примера интерфейс *IStrange* для некоторого обобщенного типа T содержит два метода и два свойства.

```
type
  IStrange<T> = interface
    function M1(p: T): T;
    procedure M2(p1: T; var P2: T);
    property P1: T read write;
    property P2: T read;
  end;
```

Класс может наследовать один и более интерфейсов. Если в нескольких интерфейсах методы или свойства одинаковы, они сливаются в один. Чтобы указать интерфейсы, которые наследует класс, в заголовке описания класса их имена нужно заключить в круглые скобки и перечислить после ключевого слова **class**. Если класс имеет предка, перечисление имен интерфейсов следует за именем предка, отделяясь от него запятой:

```
Класс1 = class(IComparable)
Класс2 = class(IComparable, IEnumerable)
Класс3 = class(Предок, IComparable, IEnumerable)
```

13.9.2 Реализация интерфейса в классе

Если класс реализует интерфейс, он обязан открыто (в секции с модификатором видимости **public**) реализовать все его методы и свойства. Нарушение этого требования обнаруживается на этапе компиляции с выдачей сообщения вида «Класс *ИмяКласса* не реализует метод *ЗаголовокМетода* интерфейса *ИмяИнтерфейса*». Аналогичное сообщение выдается и для свойств.

Поскольку интерфейс фактически является абстрактным классом, реализация интерфейса подобна написанию класса-потомка для абстрактного класса. Рассмотрим реализацию на примере программы p13018 (см. 13.3.2). Там решается задача вычисления площадей для некоторого набора прямоугольников и квадратов. С этой целью был написан абстрактный класс *Фигура*, от которого наследовались классы других геометрических объектов. Посмотрим, как эту же программу можно реализовать с помощью интерфейса.

```
// p13029
type
  IФигура = interface
    procedure Вывести;
    property Площадь: real read;
  end;

  Прямоугольник = auto class(IФигура)
    private
      Длина: real;
      Ширина: real;
    public
      procedure Вывести :=
          $'{Длина} x {Ширина} = {Площадь}'.Println;
      property Площадь: real read Длина * Ширина;
  end;

  Квадрат = auto class(IФигура)
    private
      Сторона: real;
    public
      procedure Вывести :=
          $'{Сторона} x {Сторона} = {Площадь}'.Println;
      property Площадь: real read Sqr(Сторона);
  end;

  Круг = auto class(IФигура)
    private
      Радиус: real;
    public
      procedure Вывести :=
         $'Pi x {Радиус}^2 = {Площадь}'.Println;
      property Площадь: real read Pi * Sqr(Радиус);
  end;

begin
  var L := new List<IФигура>;
  L.Add(new Прямоугольник(12, 15.5));
  L.Add(new Прямоугольник(18.2, 9.4));
  L.Add(new Квадрат(15));
  L.Add(new Круг(23.27));
  foreach var p in L do
    p.Вывести
end.
```

Если не принимать в расчет «более правильное» отнесение полей в классах к закрытым и использование автоклассов, позволившее убрать описание конструкторов, разница с программой p13018 окажется несущественной. Если вы умеете наследовать от абстрактного класса, реализация интерфейсов особого труда не составит.

13.9.3 Наследование интерфейсов

Интерфейсы, как и классы, можно наследовать. В отличие от классов, у которых может быть только один предок (в PascalABC.NET множественное наследование от классов не поддерживается), разрешено унаследовать несколько интерфейсов.

```
type
  IPoint = interface
    property x: real read;
    property y: real read;
  end;

  IPrint = interface
    procedure Print;
  end;

  IPrintPoint = interface(IPoint,IPrint)
  end;

begin

end.
```

При множественном наследовании возможен случай, когда в двух или более интерфейсах встречаются одноименные методы или свойства. Это не вызывает проблемы: одинаковые члены сливаются в один.

```
// p13030
type
  IКорова = interface
    procedure Хвост;
  end;

  ICамолет = interface
    procedure Хвост;
  end;

  Демонстратор = class(IКорова, ICамолет)
  public
    procedure Хвост := begin Println('У меня есть хвост') end;
  end;

begin
  var D := new Демонстратор;
  D.Хвост;
end.
```

Здесь все очевидно – обычный вызов метода. А как быть, если для каждого интерфейса одноименный метод должен вызывать свою процедуру? В классе, наследующем интерфейсы, нужно описать оба метода, но ведь они одноименные! Сделать необходимые описания несложно: всего лишь нужно указать имя интерфейса, а затем, через точку, имя метода. А при вызове придется делать явное приведение объекта к типу интерфейса. Посмотрите на примере, как это происходит.

```
//p13031
type
  IКорова = interface
    procedure Хвост;
  end;

  ICамолет = interface
    procedure Хвост;
  end;

  Демонстратор = class(IКорова, ICамолет)
  public
    procedure IКорова.Хвост;
    begin
      Println('Корова вертит хвостом');
    end;

    procedure ICамолет.Хвост;
    begin
      Println('Горизонтальный руль расположен на хвосте самолета');
    end;
  end;

begin
  var D := new Демонстратор;
  IКорова(D).Хвост;
  ICамолет(D).Хвост;
end.
```

13.9.4 Некоторые стандартные интерфейсы .NET

В пространстве имен System находится множество стандартных интерфейсов. Некоторые из интерфейсов существуют в необобщенной и в обобщенной форме, но нас будет интересовать наиболее употребительная обобщенная форма. Рассмотрим три интерфейса.

13.9.4.1 Сравнение объектов. Интерфейс IComparable<T>

Интерфейс IComparable<T> определяет способ сравнения двух объектов, служащий для их упорядочивания. В нем имеется единственный метод CompareTo(T), который сравнивает текущий объект (CO) с объектом-параметром T и возвращает значение типа **integer**, указывающее необходимую для упорядочивания по возрастанию позицию CO относительно T. Возможны три варианта возвращаемых значений:

- меньше нуля – CO должен предшествовать T;
- равно нулю – CO и T при упорядочивании находятся на одной позиции;
- больше нуля – CO должен следовать после T.

Если тип T отличается от типа CO, возбуждается исключение ArgumentException.

Интерфейс IComparable<T> позволяет сортировать элементы, хранящиеся в стандартных коллекциях. Также можно непосредственно обратиться к методу CompareTo для сравнения пары объектов.

Пусть имеется некоторый список выпускников, в котором указаны фамилии с инициалами, год выпуска и набранный средний балл. Требуется хранить этот список и обеспечить получение его элементов, упорядоченных по возрастанию года выпуска, а при одном годе выпуска – по убыванию среднего балла. При одном годе выпуска и равных баллах, фамилии с инициалами следует располагать в алфавитном порядке. Создадим класс *Выпускники* с интерфейсом IComparable<T> и виртуальными свойствами *Фамилия*, *Выпуск, СреднийБалл*., чтобы при необходимости унаследовать класс, позволяющий делать корректировку хранимых данных. Приучаясь к хорошему стилю программирования, поля объявляем закрытыми.

```
// p13032
type
  Выпускники = class(IComparable<Выпускники>)

  private
    ФамилияИО: string;
    ГодВыпуска: integer;
    СрБалл: real;
  public

    constructor(Фам: string; Год: integer; Балл: real) :=
        (ФамилияИО, ГодВыпуска, СрБалл) := (Фам, Год, Балл);

    function CompareTo(Выпускник: Выпускники): integer;
    begin
      Result := Sign(Self.ГодВыпуска - Выпускник.ГодВыпуска);
      if Result = 0 then
        Result := -Sign(Self.СрБалл - Выпускник.СрБалл);
      if Result = 0 then
        if Self.ФамилияИО < Выпускник.ФамилияИО then
          Result := -1
        else
          if Self.ФамилияИО > Выпускник.ФамилияИО then
            Result := 1;
    end;

    property Фамилия: string read ФамилияИО; virtual;
    property Выпуск: integer read ГодВыпуска; virtual;
    property СреднийБалл: real read СрБалл; virtual;
  end;
```

```
begin
  var НашиВыпускники := new List<Выпускники>;
  НашиВыпускники.Add(new Выпускники('Смирнов А.И.',2010, 4.27));
  НашиВыпускники.Add(new Выпускники('Петров К.В.',2012, 3.72));
  НашиВыпускники.Add(new Выпускники('Тарасова Г.А.',2010, 4.6));
  НашиВыпускники.Add(new Выпускники('Попова И.Ф.',2008, 4.47));
  НашиВыпускники.Add(new Выпускники('Смирнов А.И.',2008, 4.25));
  НашиВыпускники.Add(new Выпускники('Смирнов А.И.',2010, 4.73));
  НашиВыпускники.Sorted.PrintLines;
end.
```

Программа выведет упорядоченный список:

```
(Попова И.Ф.,2008,4.47)
(Смирнов А.И.,2008,4.25)
(Смирнов А.И.,2010,4.73)
(Тарасова Г.А.,2010,4.6)
(Смирнов А.И.,2010,4.27)
(Петров К.В.,2012,3.72)
```

Если вы хоть на мгновение подумали, что можно отсортировать полученный список и без интерфейса, сделайте простой эксперимент. Уберите из заголовка класса упоминание об интерфейсе. При попытке выполнить программу возникнет исключение с сообщением «Ошибка времени выполнения: По крайней мере в одном объекте должен быть реализован интерфейс IComparable». Ну что же, теперь вы знаете, как его реализовать!

Справедливости ради надо заметить, что отсортировать данный список без интерфейса все же можно. Для этого надо последовательно использовать сортировки по каждому свойству:

```
НашиВыпускники.
    OrderBy(t -> t.ГодВыпуска).
    ThenByDescending(t -> t.СреднийБалл).
    ThenBy(t -> t.Фамилия).
    PrintLines;
```

13.9.4.2 Сравнение объектов. Интерфейс IComparer<T>

Интерфейс IComparer<T>, подобно IComparable<T>, определяет способ сравнения двух объектов, служащий для их упорядочивания. В нем имеется единственный метод Compare(x: T, y: T), который сравнивает между собой объекты x и y. Метод возвращает значение типа **integer**, указывающее необходимую для упорядочивания по возрастанию позицию x относительно y. Возможны три варианта возвращаемых значений:

- меньше нуля – x должен предшествовать y;
- равно нулю – x и y при упорядочивании находятся на одной позиции;
- больше нуля – x должен следовать после y.

Интерфейс IComparer<T> используется в методах BinarySearch и Sort списка List<T>. На основе этого интерфейса классы SortedDictionary и SortrdList выстраивают упорядоченность элементов.

В отличие от IComparable<T>, интерфейс IComparer<T> не указывается для класса сравниваемых объектов. Вместо этого создается отдельный класс-компаратор, поддерживающий IComparer<T>, в методе Compare которого задается алгоритм сравнения. Методы сортировки умеют принимать ссылку на объект класса-компаратора в качестве параметра. Такой механизм позволяет задавать различные критерии упорядочивания объектов, определив несколько классов-компараторов.

Рассмотрим, как можно реализовать упорядоченность по различным критериям для задачи, приведенной выше. Построим три класса компаратора, по одному на каждое свойство. Характер упорядочивания свойств также возьмем из этой задачи.

```
// p13033
type
  Выпускники = class
  private
    ФамилияИО: string;
    ГодВыпуска: integer;
    СрБалл: real;
  public
    constructor(Фам: string; Год: integer; Балл: real) :=
        (ФамилияИО, ГодВыпуска, СрБалл) := (Фам, Год, Балл);
    property Фамилия: string read ФамилияИО; virtual;
    property Выпуск: integer read ГодВыпуска; virtual;
    property СреднийБалл: real read СрБалл; virtual;
  end;


  СортироватьПоФамилии = class(IComparer<Выпускники>)
  public
    function Compare(a, b: Выпускники): integer;
    begin
      if a.ФамилияИО < b.ФамилияИО then
        Result := -1
      else
        if a.ФамилияИО > b.ФамилияИО then
          Result := 1
        else
          Result := 0
    end;
  end;


  СортироватьПоБаллу = class(IComparer<Выпускники>)
  public
    function Compare(a, b: Выпускники) :=
        -Sign(a.СрБалл - b.СрБалл);
  end;
```

```
  СортироватьПоГоду = class(IComparer<Выпускники>)
public
  function Compare(a, b: Выпускники) :=
      Sign(a.ГодВыпуска - b.ГодВыпуска);
end;

begin
  var НашиВыпускники := new List<Выпускники>;
  НашиВыпускники.Add(new Выпускники('Смирнов А.И.',2010, 4.27));
  НашиВыпускники.Add(new Выпускники('Петров К.В.',2012, 3.72));
  НашиВыпускники.Add(new Выпускники('Тарасова Г.А.',2010, 4.6));
  НашиВыпускники.Add(new Выпускники('Попова И.Ф.',2008, 4.47));
  НашиВыпускники.Add(new Выпускники('Смирнов А.И.',2008, 4.25));
  НашиВыпускники.Add(new Выпускники('Смирнов А.И.',2010, 4.73));
  Println('По фамилиям:');
  НашиВыпускники.Sort(new СортироватьПоФамилии);
  НашиВыпускники.PrintLines;
  Println('По убыванию среднего балла:');
  НашиВыпускники.Sort(new СортироватьПоБаллу);
  НашиВыпускники.PrintLines;
  Println('По году выпуска:');
  НашиВыпускники.Sort(new СортироватьПоГоду);
  НашиВыпускники.PrintLines;
end.
```

Программа выведет следующие результаты:

```
По фамилиям:
(Петров К.В.,2012,3.72)
(Попова И.Ф.,2008,4.47)
(Смирнов А.И.,2008,4.25)
(Смирнов А.И.,2010,4.27)
(Смирнов А.И.,2010,4.73)
(Тарасова Г.А.,2010,4.6)
По убыванию среднего балла:
(Смирнов А.И.,2010,4.73)
(Тарасова Г.А.,2010,4.6)
(Попова И.Ф.,2008,4.47)
(Смирнов А.И.,2010,4.27)
(Смирнов А.И.,2008,4.25)
(Петров К.В.,2012,3.72)
По году выпуска:
(Смирнов А.И.,2008,4.25)
(Попова И.Ф.,2008,4.47)
(Смирнов А.И.,2010,4.27)
(Тарасова Г.А.,2010,4.6)
(Смирнов А.И.,2010,4.73)
(Петров К.В.,2012,3.72)
```

Можно было не определять три отдельных класса компаратора, а создать один с параметром, определяющим вид сортировки. Например, определить перечислимый тип данных *КритерииСортировки = (ПоФамилии, ПоБаллу, ПоГоду)*, объявить в классе *Сортировать* поле этого типа, инициализируемое параметром конструкто-

ра, а в теле метода Compare организовать ветвление по значению поля при помощи оператора **case**. В этом случае вызов сортировки будет иметь примерно следующий вид: *НашиВыпускники.Sort(new Сортировать(ПоФамилии))*. Конечно, также возможны и другие решения.

13.9.4.3 Клонирование объекта. Интерфейс ICloneable

Интерфейс ICloneable позволяет клонировать объект. В языках программирования, использующих ссылочную модель данных, необходимо различать копирование и клонирование. При копировании создается копия ссылки на объект, а при клонировании – копия самого объекта. Например, если X – некий объект, то операция Y := X поместит в Y не копию объекта, а всего лишь копию ссылки и после этого как X, так и Y будут ссылаться на один и тот же объект. Фактически, объект получит еще одно имя – и только.

Клонирование бывает поверхностным (shallow) и глубоким (deep). При поверхностном клонировании в копию переносятся данные размерного типа и ссылки на данные ссылочного типа. При глубоком клонировании должна создаваться точная копия объекта, что требует обхода всех членов внутри него, поскольку в классе могут встречаться объекты, полями которых являются другие объекты. Можно реализовать собственные методы копирования или воспользоваться стандартным интерфейсом ICloneable, имеющим единственный метод Clone, на который возлагаются функции клонирования.

Поддержка классом интерфейса ICloneable сообщает программному окружению, что для копирования можно (и рекомендовано) использовать метод Clone. Осуществляя реализацию этого интерфейса следует помнить, что он описан для объектов базового класса object, поэтому придется выполнять даункаст.

В общем случае организация глубокого клонирования объекта не является тривиальной. Известен способ глубокого клонирования с помощью «конструктора копирования». Его смысл в том, чтобы сделать параметром конструктора объект этого же класса и написать код, копирующий из объекта-параметра каждое поле.

```
type
  Obj = class
  public
    constructor(p: Obj);
    begin
      (x, y, z) := (p.x, p.y, p.z);
      a := Copy(p.a)
    end;
    auto property x: integer;
    auto property y: integer;
    auto property z: integer;
    auto property a: array of integer;
  end;
```

Для осуществления глубокого клонирования объекта *A* в объект *B* будет достаточно написать *A := **new** Obj(B)*.

Код с перечислением полей, особенно когда их много, может показаться не лучшим решением. В этом случае можно выполнить поверхностное копирование при помощи защищенного (**protected**) метода MemberwiseClone, принадлежащего базовому классу object. Объекты, стоящие за ссылочными полями, придется копировать самостоятельно. Создадим собственный метод *MyClone*, реализующий клонирование и учтем, что MemberwiseClone можно вызывать только изнутри класса.

```
type
  Obj = class
  public
    function MyClone: Obj;
    begin
      Result := Obj(Self.MemberwiseClone);
      Result.a := Copy(Self.a)
    end;
    auto property x: integer;
    auto property y: integer;
    auto property z: integer;
    auto property a: array of integer;
  end;
```

Для того, чтобы реализовать в классе интерфейс ICloneable, потребуется переименовать метод MyClone, описанный выше, в Clone и немного его изменить.

> При вызове метода IClone не забывайте, что он возвращает объект класса object, а не вашего класса и требуется явный даункаст.

```
type
  Obj = class(System.ICloneable)
  public
    function Clone: object;
    begin
      Result := Obj(Self.MemberwiseClone);
      Obj(Result).a := Copy(Self.a)
    end;
    auto property x: integer;
    auto property y: integer;
    auto property z: integer;
    auto property a: array of integer;
  end;
```

Интерфейс ICloneable подразумевает реализацию метода Clone для объекта базового класса object и потребовалось явное приведение типа object к *Obj* для выполнения копирования элементов массива *a*.

В качестве примера рассмотрим полную программу, выполняющую копирование, поверхностное клонирование и глубокое клонирование для объектов класса *Obj*. Создадим эталонный объект *oE*, а затем при помощи глубокого клонирования

будем копировать его в объект *oA* и различными методами переносить в объект *oB*. Оценим степень влияния изменения значения полей *oB* на *oA*. Для сокращения кода добавим в класс *Obj* два вспомогательных метода.

```
// p13034
type
  Obj = class(System.ICloneable)
  public
    constructor(px, py, pz: integer; pa: array of integer);
    begin
      (x, y, z) := (px, py, pz);
      a := Copy(pa);
    end;

    function MyClone: Obj; // поверхностное клонирование
    begin
      Result := Obj(Self.MemberwiseClone);
    end;

    function Clone: object;
    begin
      Result := Obj(Self.MemberwiseClone);
      Obj(Result).a := Copy(Self.a)
    end;

    auto property x: integer;
    auto property y: integer;
    auto property z: integer;
    auto property a: array of integer;
  end;

procedure PrinrObj(msg: string; pO: Obj);
begin
  Write($'{msg}: x={pO.x}, y={pO.y}, z={pO.z}, a=[');
  pO.a.Print;
  Writeln(']')
end;

procedure Test(E: Obj; var A, B: Obj; msg: string; P: Obj-> Obj);
begin
  A := Obj(E.Clone);
  Println(msg);
  PrinrObj('Эталон', E);
  B := P(A);
  (B.y, B.a[2]) := (8, 9);
  PrinrObj('OA    ', A);
  PrinrObj('OB    ', B);
end;
```

```
begin
  var oE := new Obj(1, 2, 3, Arr(4, 5, 6, 7)); // эталон
  var oA, oB: Obj;
  Test(oE, oA, oB, 'Копирование', t -> t);
  Test(oE, oA, oB, 'Поверхностное клонирование', t -> t.MyClone);
  Test(oE, oA, oB, 'Глубокое клонирование', t -> Obj(t.Clone));
end.
```

Разница между копированием и клонированием отражена в результатах работы программы. Присваивание после копирования «испортило» как размерные, так и ссылочные данные, а после поверхностного клонирования – только ссылочные.

```
Копирование
Эталон: x=1, y=2, z=3, a=[4 5 6 7]
OA    : x=1, y=8, z=3, a=[4 5 9 7]
OB    : x=1, y=8, z=3, a=[4 5 9 7]
Поверхностное клонирование
Эталон: x=1, y=2, z=3, a=[4 5 6 7]
OA    : x=1, y=2, z=3, a=[4 5 9 7]
OB    : x=1, y=8, z=3, a=[4 5 9 7]
Глубокое клонирование
Эталон: x=1, y=2, z=3, a=[4 5 6 7]
OA    : x=1, y=2, z=3, a=[4 5 6 7]
OB    : x=1, y=8, z=3, a=[4 5 9 7]
```

Надо понимать, что пример носит скорее иллюстративный, чем практический характер. Здесь можно было обойтись без интерфейса. Но, реализовав в классе *Obj* стандартный интерфейс, теперь можно на его основе строить записи или контейнерные классы, в которых клонирование таких объектов уже будет обеспечено.

13.10 Статический класс System.Console

Статический ненаследуемый класс System.Console весьма популярен в программах, написанных на языке C#, но и в PascalABC.NET многие авторы программ любят его использовать. Запускать программу, использующую этот класс, следует посредством комбинации клавиш {Shift}{F9}.

Консоль – это окно операционной системы, через которое осуществляется обмен текстовыми сообщениями с пользователем. В частности, в Windows под консолью понимается окно с интерфейсом командной строки. Класс Console обеспечивает широкий набор операций для работы с консолью.

Обращение к свойствам и методам этого класса производится в следующем виде:

```
System.Console.ИмяСвойства
System.Console.ИмяМетода
System.Console.ИмяМетода(СписокПараметров)
```

Если постоянное упоминание System вызывает раздражение, можно выполнить подключение этого пространства имен посредством оператора **uses** System и обращаться к членам класса Console непосредственно:

```
uses System;

Console.ИмяСвойства
Console.ИмяМетода
Console.ИмяМетода(СписокПараметров)
```

Подробная информация по классу Console приведена в документации Microsoft, а здесь будет рассмотрены только некоторые свойства и методы.

13.10.1 Некоторые свойства класса Console

Перечисленные свойства доступны для чтения и изменения. Под буфером понимается ассоциированная со строками ввода и вывода область памяти, с которой осуществляется обмен данными. Окно является визуализированной частью буфера. Размеры окна могут совпадать с размерами буфера или быть меньше него; в последнем случае окно получает полосы прокрутки. Значения, принимаемые по умолчанию, ниже указаны в квадратных скобках.

- BackgroundColor: ConsoleColor – цвет фона строки [Black];
- ForegroundColor: ConsoleColor – цвет текста строки [Gray];
- BufferHeight: **integer** – высота буфера в строках [300];
- BufferWidth: **integer** – ширина буфера в символах [80];
- WindowHeight: **integer** – высота окна в строках [25];
- WindowWidth: **integer** – ширина окна в символах [80];
- Title: string – текст, выводящийся в заголовке окна.

Экран имеет черный фон. На нем цветом ForegroundColor отображаются текстовые строки, имеющие фон BackgroundColor. Имена стандартных цветов находятся в ConsoleColor и доступны для выбора при вводе «точки».

```
uses System;

begin
  Console.Title := 'Тестовое окно';
  Console.BackgroundColor := ConsoleColor.DarkBlue;
  Console.ForegroundColor := ConsoleColor.Cyan;
  Console.WriteLine('Привет!');
end.
```

13.10.2 Некоторые методы класса Console

Класс Console при вводе и выводе использует три стандартных потока: стандартный входной поток для чтения данных, стандартный выходной поток для вывода данных и стандартный поток вывода ошибок. По умолчанию обмен данными происходит с консолью, но имеется возможность перенаправить обмен в файлы.

- Beep – воспроизводит звук «бип»;

- Clear – очищает буфер и стирает содержимое экрана;
- Read: **integer** – читает очередной символ и возвращает его код;
- ReadKey: ConsoleKeyInfo – читает и возвращает код очередной нажатой клавиши;
- ReadLine: **string** – читает и возвращает текстовую строку;
- ResetColor – устанавливает стандартные значения цветов в окне;
- Write(ВыводимаяСтрока) – выводит строку;
- WriteLine(ВыводимаяСтрока) – выводит строку, затем переходит к новой строке;
- Write(ФорматнаяСтрока, ВыводимаяСтрока) – форматирует и выводит строку;
- WriteLine(ФорматнаяСтрока, ВыводимаяСтрока) – форматирует и выводит строку, затем переходит к новой строке;
- WriteLine – выводит символ перевода к новой строке.

Форматные строки описывались в части 6 (см. 6.2.26.2).

```
uses System;

begin
  Console.ForegroundColor := ConsoleColor.Magenta;
  Console.WriteLine('Длина окружности диаметра {0} равна {1}',
      2.16, 2.16 * Pi)
end.
```

13.11 Класс – обработчик исключений

Как указано в Справке, все классы исключений являются потомками класса System.Exception, включающего следующие открытые члены:

```
type
  Exception = class
  public
    constructor Create;
    constructor Create(message: string);
    property Message: string; // только на чтение
    property StackTrace: string; // только на чтение
  end;
```

Свойство *Message* возвращает сообщение, связанное с объектом исключения. Свойство *StackTrace* возвращает стек вызовов подпрограмм на момент генерации исключения.

Классом Exception можно воспользоваться с целью замены стандартного сообщения собственным.

```
// p13035
begin
  try
    var i := 0;
    Println(4 div i)
  except
    on e: System.DivideByZeroException do
      raise new Exception('Как надоело делить на ноль!');
  end
end.
```

Для определения собственного исключения нужно унаследовать новый класс от System.Exception. В простейшем случае тело класса может быть пустым – так поступают, если нужно выделить какое-либо исключение именем:

```
// p13036
type
  ОпятьЭтоДелениеЦелогоНаНоль = class(Exception)
end;

begin
  try
    var i := 0;
    Println(4 div i)
  except
    on e: System.DivideByZeroException do
      raise new ОпятьЭтоДелениеЦелогоНаНоль;
  end
end.
```

Можно наполнить класс некоторым содержанием, определив нужные действия в его конструкторе. Например, определить функцию для выполнения целочисленного деления так, чтобы при нулевом значении делителя она вызывала исключение, которое будет обрабатываться написанным классом.

```
// p13037
type
  НаНольДелитьНельзя = class(Exception)
    constructor(p: Exception);
    begin
      var s := p.StackTrace.ToWords.Last;
      $'Div0: Строка {s}: нулевой делитель, присвоен максимум'.Println
    end;
  end;

function Div0(a, b: integer): integer;
begin
  try
    Result := a div b
  except
    on e: System.DivideByZeroException do
      try
        Result := integer.MaxValue;
        raise new НаНольДелитьНельзя(e);
      except
      end
  end
end;

begin
  var i := 0;
  Println(Div0(4, i))
end.
```

Здесь вложенный **try** с пустым **except** подавляет сообщение о возникшем после **raise** исключении.

Почему был выбран именно такой путь? Изначально нас не устраивала существующая реакция на операцию div при нулевом делителе. Можно рассуждать следующим образом: либо следует перегрузить операцию, либо изменить существующую обработку возникающего исключения.

Если даже удастся перегрузить операцию **div** для типа **integer** (обычно для базовых классов перегружать стандартные операции нельзя), придется определять целочисленное деление через какие-то другие операции, что очень неэффективно. Переопределить исключение DivideByZeroException также не получится. Следовательно, остается возможность либо создать свой класс на базе integer и определить для него весь набор необходимых операций, функций и процедур, либо написать отдельную функцию для выполнения операции целочисленного деления создания свой класс для обработки исключения DivideByZeroException, что и было сделано.

Рассмотрим еще один пример создания собственного класса для обработки исключения. Ранее приводилась программа p13006 (см. 13.1.4) с классом, позволяющим вычислять параметры треугольников, заданных длинами сторон. В ней во многих местах выполнялась проверка возможности существования треугольника. Предлагаемый вариант программы для обработки ситуации с «невозможным» треугольником использует простой собственный класс *ТреугольникНевозможен*, реализующий пользовательское исключение.

```
type // p13038
  ТреугольникНевозможен = class(Exception)
    constructor;
    begin
      Println('Невозможно построить треугольник');
      Println
    end;
  end;

  Треугольник = class
  private
    aa, bb, cc, r1, r2, P, S, p2: real;
    aa1, bb1, cc1: real;

    procedure Вычислить;
    begin
      try
        if (aa + bb > cc) and (aa + cc > bb) and (bb + cc > aa) then
        begin
          P := aa + bb + cc;
          p2 := P / 2;
          S := Sqrt(p2 * (p2 - aa) * (p2 - bb) * (p2 - cc));
          r1 := S / p2;
          r2 := aa * bb * cc / (4 * S);
          ВывестиРезультаты
        end
        else
        begin
          (aa, bb, cc) := (aa1, bb1, cc1);
          raise new ТреугольникНевозможен
        end
      except
        // сбрасываем возбужденное исключение
      end
    end;

    procedure SetAA(v: real);
    begin
      aa1 := aa;
      aa := v;
      Println('Длины сторон:', aa, bb, cc);
      Вычислить
    end;

    procedure SetBB(v: real);
    begin
      bb1 := bb;
      bb := v;
      Println('Длины сторон:', aa, bb, cc);
      Вычислить
    end;
```

```
    procedure SetCC(v: real);
    begin
      cc1 := cc;
      cc := v;
      Println('Длины сторон:', aa, bb, cc);
      Вычислить
    end;

    procedure ВывестиРезультаты;
    begin
      $'Периметр {P}, площадь {S:f3}'.Println;
      Writeln('Радиус вписанной окружности ', r1:0:3);
      Writeln('Радиус описанной окружности ', r2:0:3);
      Println
    end;

  public

    constructor(pa: real := 3.0; pb: real := 4.0; pc: real := 5.0);
    begin
      (aa, bb, cc) := (pa, pb, pc);
      (aa1, bb1, cc1) := (aa, bb, cc);
      Println('Длины сторон:', aa, bb, cc);
      Вычислить
    end;

    property a: real read aa write SetAA;
    property b: real read bb write SetBB;
    property c: real read cc write SetCC;

  end;

begin
  var oT := new Треугольник;
  oT.c := 5.1;
  oT.b := 6.7;
  oT.a := 18;
  oT.a := 4.13;
end.
```

Вместо 80 строк код программы теперь содержит 91 строку. Но на мой взгляд программа стала более «прозрачной». Методы здесь более специализированы, следовательно, если понадобится внести какие-то изменения, это можно будет сделать проще.

Результат работы программы выглядит следующим образом:

```
Длины сторон: 3 4 5
Периметр 12, площадь 6.000
Радиус вписанной окружности 1.000
Радиус описанной окружности 2.500

Длины сторон: 3 4 5.1
Периметр 12.1, площадь 5.995
Радиус вписанной окружности 0.991
Радиус описанной окружности 2.552

Длины сторон: 3 6.7 5.1
Периметр 14.8, площадь 7.240
Радиус вписанной окружности 0.978
Радиус описанной окружности 3.540

Длины сторон: 18 6.7 5.1
Невозможно построить треугольник

Длины сторон: 4.13 4 5
Периметр 13.13, площадь 8.011
Радиус вписанной окружности 1.220
Радиус описанной окружности 2.578
```

В случае, когда попытка изменения длины одной из сторон приводит к невозможности построить треугольник, теперь длина стороны принимает прежнее значение. Еще одно преимущество использования свойства вместо предоставления прямого доступа к полю.

13.12 Для самостоятельного решения

T13.1. В части 3 (см. 3.6.7) приводилась задача табуляции функции двух переменных. Реализуйте класс Tabula для табуляции произвольной функции одной переменной на интервале [a; b] с шагом h. Функция должна передаваться в класс вместе с параметрами табуляции. Класс должен содержать метод .ToTab, возвращающий последовательность кортежей (x, f(x)), где x – значение аргумента, f(x) – соответствующее значение функции. Реализуйте расширение .OutTab, выводящее результат табуляции в две колонки с задаваемой шириной вывода и количеством знаков в дробной части для каждой колонки. Расширение должно вернуть исходную последовательность для встраивания в цепочку.

T13.2. Разработайте абстрактный класс tRoot, имеющий виртуальные методы вычисления действительных корней уравнений и вывода их значений. Унаследуйте от tRoot классы для решения уравнений первой (tLinear) и второй (tSquare) степени.

T13.3. Дана строка, представляющая собой правильное выражение в польской инверсной записи (ПОЛИЗ). В строке допустимо использовать запись целых чисел со знаком, а также знаки операций +, -, * и / (деление нацело). Все элементы в

строке разделены пробелом. Реализуйте класс, содержащий функцию, возвращающую вычисленное значение выражения, записанного в ПОЛИЗ. Проверьте работу программы со строкой

```
'36 18 3 * - 5 4 7 5 - * - /'
```

соответствующей выражению

$$\frac{36-18\times 3}{5-4\times(7-5)}$$

Вычисление выражения, записанного в ПОЛИЗ, можно провести с помощью стека. Читаем очередной элемент выражения. Если это число – заносим его в стек. Если знак операции – выполняем ее над двумя последними числами, помещенными в стек. Процесс продолжаем до окончания просмотра строки.

T13.4. В библиотеке NumLibABC имеется класс Fraction для работы с простыми дробями. Дробь хранится в виде пары числовых значений числителя (поле numerator) и знаменателя (поле denominator) типа BigInteger. Для создания дроби можно, кроме конструктора, вызывать функцию Frc(числитель, знаменатель). В 13.8 приводится утверждение о том, что «Любой существующий тип данных, независимо от того, является ли он пользовательским или базовым… можно ***расширить***, добавив ***методы расширения***». Также, в 13.8.1 упомился принцип Открытости/Закрытости (OCP – Open/Close Principle), в соответствии с которым к классу можно добавлять новые методы без доступа к его исходному коду. С целью проверки этих утверждений разработайте методы расширения Sum, Average, Min и Max для стандартной коллекции List<T>, используя тип Fraction и для простоты считая, что список не может быть пустым. Заполните список List некоторым количеством простых дробей, сгенерированных случайным образом (учитывайте, что знаменатель дроби должен быть натуральным числом, а числитель – любым целочисленным). С помощью разработанных расширений найдите сумму элементов списка, их среднее арифметическое, а также дроби с минимальным и максимальным значением.

T13.5. В библиотеке NumLibABC имеется класс Fraction для работы с простыми дробями. Дробь хранится в виде пары числовых значений числителя (поле numerator) и знаменателя (поле denominator) типа BigInteger. Для создания дроби можно, кроме конструктора, вызывать функцию Frc(числитель, знаменатель). Заполните список List не менее чем десятью простыми дробями, сгенерированными случайным образом (учитывайте, что знаменатель дроби должен быть натуральным числом, а числитель – любым целочисленным). Затем отсортируйте этот список по убыванию и переформируйте, оставив в начале три самых больших дроби и две самых малых. Найдите сумму сформированного списка.

Вы сможете выполнить сортировку при помощи метода List.OrderDescending, если определите для класса дробей интерфейс IComparable. Поскольку интерфейс нельзя «доопределить», придется унаследовать от Fraction новый класс и определить интерфейс в нем. Перегруженные операции не наследуются, но их можно выпол-

нять посредством апкаста к fraction. Конечно, есть и другие пути решения задания, например, не определять подкласс с интерфейсом, а написать сортировку самостоятельно.

Часть 14

Ответы и решения

Программирование сегодня – это гонка между разработчиками, стремящимися писать все лучше защищенные от идиотов программы, и Природой, стремящейся создавать все более совершенных идиотов. Пока побеждает Природа.

Рич Кук, писатель-фантаст

Приведены ответы и решения заданий, размещенных в конце частей с 1 по 13.

14.1 К части 1

T1.1. Целочисленный тип **shortint** – это целое со знаком, занимающее один байт. Из таблицы 1.1 «Целые типы PascalABC.NET» определяем диапазон представления значений для этого типа (-128 .. 127). Значение shortint.MaxValue – это 127. Данный тип может представить 256 значений, из которых 128 отрицательные, одно нулевое и 127 положительные. Переполнения не возникает, вместо этого получаем, что 127 + 1 = -128. Значения чисел следуют по кольцу. Следовательно, через каждые 256 сложений с единицей получается исходное значение и нас интересует лишь остаток от деления заданного числа сложений на 256, т.е. 1154 mod 256.

Можно последовательно вычитать 256, получая числа 898, 642, 386 и 130. А можно вспомнить, что $256 = 2^8$, поэтому несколько раз взятое число 256 будет также кратно степени двойки. Ближайшая к 1154 степень двойки – десятая, $2^{10} = 1024$. Далее находим 1154 – 1024 = 130. Достаточно выполнить 130 сложений. Первое же сложение 127 + 1 даст -128. Останется 129 сложений. -128 + 129 = 1. Это и есть ответ.

T1.2. Как и в предыдущем случае, по таблице определяем, что тип **smallint** занимает два байта, его минимальное значение равно -32768. Однобайтный тип **shortint** имеет максимальное значение 127. Получаем -32768 + 127 = -32641. Оба операнда будут приведены к типу **integer**, результат также будет иметь тип **integer**.

T1.3. Площадь стен – это площадь боковой поверхности параллелепипеда $S_{бок} = 2 \times Высота(Длина + Ширина)$ за вычетом площадей проемов.

```
begin
  var (Длина, Ширина, Высота) := ReadInteger3('Размеры, см:');
  var ЦенаБанки := ReadInteger('Цена банки:');
  var (ШиринаДвери, ВысотаДвери) := (92, 210);
  var ПлощадьДвери := ШиринаДвери * ВысотаДвери;
  var (ШиринаОкна, ВысотаОкна) := (207, 150);
  var ПлощадьОкна := ШиринаОкна * ВысотаОкна;
  var Площадь := 2 * Высота * (Длина + Ширина) - ПлощадьДвери - ПлощадьОкна;
  var (РасходКраскиНа1КвМ, КраскиВБанке) := (120, 3800);
  var РасходКраски := РасходКраскиНа1КвМ * Площадь div 10000;
  var Банок := РасходКраски div КраскиВБанке +
      Min(1, РасходКраски mod КраскиВБанке);
  Println('Банок:', Банок, 'на сумму', Банок * ЦенаБанки)
end.
```

В приведенной программе есть одна маленькая «хитрость». Чтобы получить количество банок краски, нужно общий расход краски разделить на количество краски в одной банке. Пусть расход составляет 5102 г, а в банке 3800 г. Делим нацело 5102 на 3800 и получаем 1 банку. Ясно, что ее не хватит, нужно еще 5102 – 3800 = 1302 г, т.е. банок нужно две. Всегда увеличивать результат на 1 банку? А если расход составит ровно 3800г? С этой целью введен «довесок» с функцией Min. Нет остатка – функция вернет 0, есть – вернет единицу. Можно было написать и более экзотический вариант: Sign(РасходКраски **mod** КраскиВБанке).

14.2 К части 2

Т2.1. Для округления до ближайшего большего целого используем функцию Ceil, предварительно уменьшив полученное значение на 0.1.

```
begin
  var (Длина, Ширина, Высота) := ReadReal3('Размеры, м:');
  var ЦенаБанки := ReadInteger('Цена банки:');
  var (ШиринаДвери, ВысотаДвери) := (0.92, 2.1);
  var ПлощадьДвери := ШиринаДвери * ВысотаДвери;
  var (ШиринаОкна, ВысотаОкна) := (2.7, 1.5);
  var ПлощадьОкна := ШиринаОкна * ВысотаОкна;
  var Площадь := 2 * Высота * (Длина + Ширина) - ПлощадьДвери - ПлощадьОкна;
  var (РасходКраскиНа1КвМ, КраскиВБанке) := (0.120, 3.8);
  var РасходКраски := РасходКраскиНа1КвМ * Площадь;
  var Банок := Ceil(РасходКраски / КраскиВБанке - 0.1);
  Println('Банок:', Банок, 'на сумму', Банок * ЦенаБанки)
end.
```

Т2.2. Ответ выведем с точностью до двух знаков после запятой. Отрицательное значение свидетельствует о том, что бассейн не наполнится, а опустошится.

```
begin
  var (t1, t2) := ReadReal2('Введите время для обеих труб:');
  var t := 1 / (1 / t1 + 1 / t2);
  Writeln('Общее время составит ', t:0:2)
end.
```

14.3 К части 3

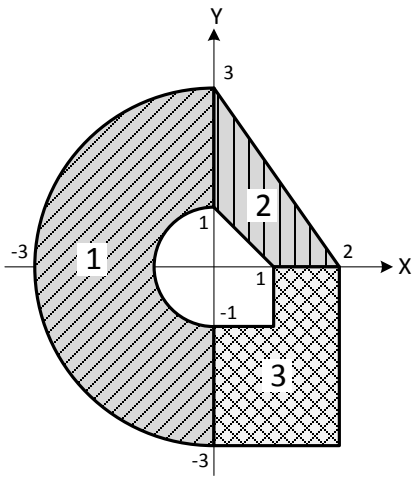
Т3.1. Приводится вариант текста программы.

```
begin
  var (d, m, y) := ReadInteger3('Укажите день, месяц и год:');
  if m > 2 then
    m -= 3
  else
  begin
    y -= 1;
    m += 9
  end;
  var (c, a) := (y div 100, y mod 100);
  var j := (146097 * c) div 4 + (1461 * a) div 4 +
```

```
      (153 * m + 2) div 5 + d + 1721119;
  Println('Юлианский день:',j)
end.
```

T3.2. Приводится вариант текста программы

```
begin
  var Сдача := ReadInteger('Сумма сдачи:');
  var КоличествоМонет := 0;
  var ВыдатьМонет := Сдача div 10;
  if ВыдатьМонет > 0 then
  begin
    Println('10 руб:', ВыдатьМонет);
    КоличествоМонет += ВыдатьМонет;
    Сдача := Сдача mod 10;
  end;
  if Сдача > 5 then
  begin
    Println('5 руб:', 1);
    КоличествоМонет += 1;
    Сдача -= 5;
  end;
  if Сдача > 2 then
  begin
    ВыдатьМонет := Сдача div 2;
    Println('2 руб:', ВыдатьМонет);
    КоличествоМонет += ВыдатьМонет;
    Сдача := Сдача mod 2;
  end;
  if Сдача = 1 then
  begin
    Println('1 руб:', 1);
    КоличествоМонет += 1;
  end;
  Println('Всего монет:', КоличествоМонет)
end.
```



T3.3. В этой задаче очень важно правильно написать уравнения линий, но это уже было сделано в условии. Далее, заштрихованную область нужно разбить на фрагменты, представляющие собой треугольники, прямоугольники, многоугольники и окружности, либо части окружностей. В данном решении выделены три фрагмента, принадлежащие различным четвертям координатной плоскости. Такое деление упрощает работу.

Фрагмент №1 – это область, ограниченная полукольцом, в которой $x \leqslant 0$. Заштрихованной области принадлежит граница и внутренняя часть круга большего радиуса за исключением

круга меньшего радиуса. Условие №1 можно записать в виде $x^2 + y^2 \le 9 \wedge x^2 + y^2 \ge 1 \wedge x \le 0$

Фрагмент №2 принадлежит I четверти. Заштрихованная область расположена на прямой с уравнением y = 1 – x, либо выше нее, но при этом она расположена на прямой с уравнением y = 3 - 1.5x, либо ниже нее. «На прямой или выше» означает условие y ⩾ f(x), «на прямой, либо ниже» - соответственно y ⩽ f(x). Опишем фрагмент №2 в виде $x \ge 0 \wedge y \ge 0 \wedge y \ge 1 - x \wedge y \le 3 - 1.5x$

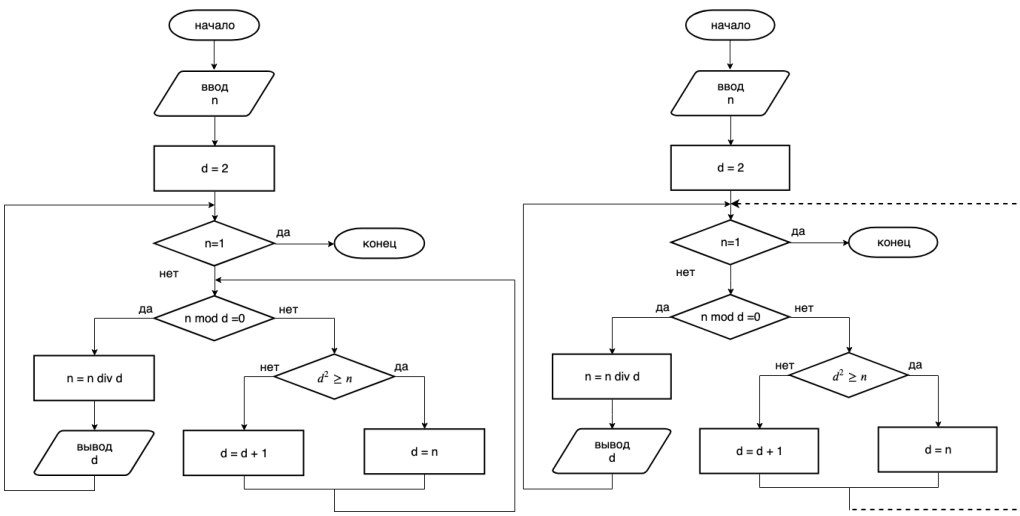
Фрагмент №3 принадлежит IV четверти. Он образован прямоугольником, из которого исключена квадратная область. К сожалению, мы пока не изучили, как сделать в PascalABC.NET операцию пересечения областей, поэтому воспользуемся условием «Принадлежит прямоугольнику и при этом не принадлежит квадрату». Не забываем, что и прямоугольник, и квадрат располагаются в IV четверти.
$x \ge 0 \wedge y \le 0 \wedge y \ge -3 \wedge x \le 2 \wedge \neg(x \ge 0 \wedge y \le 0 \wedge y > -1 \wedge x < 1)$

Знание булевой алгебры позволяет при желании немного преобразовать полученное выражение, но это совершенно необязательно:
$x \ge 0 \wedge y \le 0 \wedge y \ge -3 \wedge x \le 2 \wedge (y \le -1 \vee y > 0 \vee x \ge 1 \vee x < 0)$

Поскольку точка не может одновременно располагаться во всех четвертях, все три полученных условия связываются по «или». Сама программа несложна:

```
begin
  var (x, y) := ReadReal2('Введите значения x и y:');
  var Обл1 :=
      (x * x + y * y <= 9) and (x * x + y * y >= 1) and (x <= 0);
  var Обл2 :=
      (x >= 0) and (y >= 0) and (y >= 1 - x) and (y <= 3 - 1.5 * x);
  var Обл3 := (x >= 0) and (y <= 0) and (y >= -3) and (x <= 2) and
      not ((y <= 0) and (y > -1) and (x >= 0) and (x < 1));
  Println(Обл1 or Обл2 or Обл3)
end.
```

Т3.4. Тот самый редкий случай, когда рисовать блок-схему полезно.



Слева дана блок-схема, составленная строго по алгоритму. В ней нарушена структурность: ветвление, возникшее при анализе условия n mod d = 0 не сошлось в одну точку, что не позволит применить оператор **if – then – else**. Структурность можно восстановить, если внести небольшое изменение, показанное на правой блок-схеме штриховой линией. Ниже приведен код одной из возможных реализаций алгоритма.

```
begin
  var n := ReadInteger('n =');
  var d := 2;
  while n <> 1 do
  begin
    if n mod d = 0 then
    begin
      n := n div d;
      Print(d)
    end
    else
      if d * d > n then
        d := n
      else
        d += 1
  end
end.
```

Решение задачи показало, что алгоритм, заданный в виде последовательностей шагов с явными переходами, может создать проблему, связанную с его реализацией в современных языках. По этой причине лучше сразу составлять алгоритм, опираясь на условные конструкции и циклы – тогда вы не будете терять время на рисование блок-схем для структуризации.

Т3.5. От знака минуса можно избавиться, взяв абсолютную величину числа. Цифры числа (начиная с младших) можно получать в цикле путем нахождения остатка от деления числа на 10 с последующим уменьшением числа в 10 раз.

```
begin
  var n := Abs(ReadInteger('n ='));
  var s := 0;
  while n > 0 do
  begin
    s += n mod 10;
    n := n div 10
  end;
  Println('Сумма цифр в числе равна', s)
end.
```

T3.6. Примем за основу предыдущую программу. Учтем, что нужно хранить максимальную сумму цифр и число, которому она принадлежит.

```
begin
  var (k, Smax) := (0, 0);
  loop 15 do
  begin
    var m := Random(-999999, 999999);
    Print(m);
    var (n, S) := (Abs(m), 0);
    while n > 0 do
    begin
      s += n mod 10;
      n := n div 10
    end;
    if S > Smax then
      (k, Smax) := (m, S)
  end;
  Writeln(NewLine, 'Число ', k, ', сумма цифр ', Smax)
end.
```

T.3.7. Приводится один из возможных вариантов программы.

```
begin
  var n: int64;
  Write('Введите число: ');
  Read(n);
  n := Abs(n); // от злобных буратинок с их отрицательными числами
  while n > 9 do
  begin
    var s: int64 := 0;
    var k: int64 := n;
    while k > 0 do
    begin
      s := s + k mod 10;
      k := k div 10
    end;
    n := s;
  end;
  Println(n)
end.
```

14.4 К части 4

T4.1. Приводится один из возможных вариантов программы.

```
function Длина(ax, ay, bx, by: real) :=
    Sqrt((bx - ax) ** 2 + (by - ay) ** 2);

function Площадь(ax, ay, bx, by, cx, cy: real): real;
begin
  var la := Длина(bx, by, cx, cy);
  var lb := Длина(ax, ay, cx, cy);
  var lc := Длина(ax, ay, bx, by);
```

```
  if (la + lb > lc) and (la + lc > lb) and (lb + lc > la) then
  begin
    var p2 := (la + lb + lc) / 2;
    Result := Sqrt(p2 * (p2 - la) * (p2 - lb) * (p2 - lc))
  end
  else
    Result := 0;
end;

begin
  var ax, ay, bx, by, cx, cy: real;
  Print('Введите координаты x y для каждой точки:');
  Read(ax, ay, bx, by, cx, cy);
  var S := Площадь(ax, ay, bx, by, cx, cy);
  if S = 0 then
    Println('Треугольник невозможен')
  else
    Writeln('S = ', S:0:2)
end.
```

Т4.2. Приводится один из возможных вариантов программы.

```
procedure PP(a: real; var P, S: real) :=
    (P, S) :=(4 * a, a * a);

procedure PP(a, b: real; var P, S: real);
begin
  if b=0 then
    PP(a, P, S)
  else
    (P, S):=(2 * (a + b), a * b)
end;

procedure PP(a, b, c: real; var P, S: real);
begin
  if c = 0 then
    PP(a, b, P, S)
  else
  begin
    if (a + b > c) and (a + c > b) and (b + c > a) then
    begin
      P := a + b + c;
      var p2 := P / 2;
      S := Sqrt(p2 * (p2 - a) * (p2 - b) * (p2 - c))
    end
    else
      (P, S) := (0, 0)
  end
end;
```

```
begin
  var (a, b, c) := ReadReal3('Введите значения длин:');
  var P, S: real;
  PP(a, b, c, P, S);
  if P = 0 then Println('Треугольнгик невозможен')
  else
    if b = 0 then
      Writeln('Квадрат: периметр ', P:0:2, ', площадь ', S:0:2)
    else
      if c = 0 then
        Writeln('Прямоугольник: периметр ', P:0:2, ', площадь ', S:0:2)
      else
        Writeln('Треугольник: периметр ', P:0:2, ', площадь ', S:0:2)
end.
```

Т4.3. Случайное вещественное число, имеющее два знака после запятой можно получить при помощи датчика случайных вещественных чисел, округляя значения посредством функции round, либо при помощи датчика случайных целых чисел с последующим их делением на 100.

```
function Число := Round(Random(-15.0, 15.0), 2);

function Четверть(x, y: real): integer;
begin
  case Sign(x) of
    -1:
      case Sign(y) of
        -1: Result := 3;
        1: Result := 2;
        0: Result := 0
      end;
    1:
      case Sign(y) of
        -1: Result := 4;
        1: Result := 1;
      else Result := 0
      end;
    0: Result := 0
  end
end;
```

```
begin
  loop 10 do
  begin
    var (x, y) := (Число, Число);
    var k := Четверть(x, y);
    Write('(', x, ';', y, ') - ');
    if k = 0 then
      Writeln('не определено')
    else
      Writeln(k)
  end
end.
```

Т4.4. Расстояние от начала координат до некоторой точки вычисляется по теореме Пифагора как длина гипотенузы треугольника, у которого длины катетов совпадают со значениями координат этой точки. Точка не принадлежит III четверти, если условие принадлежности x<0 ∧ y<0 ложно. Для поиска максимума не обязательно сравнивать расстояния-гипотенузы, достаточно сравнивать их квадраты.

```
function Число := Round(Random(-15.0, 15.0), 2);

begin
  var Lmax := 0.0;
  var xmax, ymax: real;
  loop 10 do
  begin
    var (x, y) := (Число, Число);
    if not ((x < 0) and (y < 0)) then
    begin
      var L := x * x + y * y;
      if L > Lmax then
        (xmax, ymax, Lmax) := (x, y, Sqrt(L))
    end;
    Writeln('(', x, ';', y, ')')
  end;
  Writeln('Наиболее удалена точка (', xmax, ';', ymax, ')')
end.
```

Т4.5. Рекурсия. И большие значения аргументов. Тот самый случай, когда полезно вспомнить афоризм «Компьютер должен считать, а человек – думать». Когда рекурсивная функция вычисляет какие-то значения, всегда нужно сначала думать, а затем заставлять (или не заставлять) компьютер считать. При программировании рекурсивной функции следует начинать «с конца», т.е. определять условие завершения вычисления. F(a,b,c)=1, если хотя бы один из аргументов будет меньше единицы. А до этих пор нужно считать, уменьшая некоторые или все аргументы на 1. Но это означает, что при любых a, b и c значение F рано или поздно станет равно 1. Тогда выражение «с тремя F» сведется к 1+1-1=1. Вот и ответ. При любых a, b и c значение функции равно 1. Задача написания функции, всегда возвращающей единицу, тривиальна. А если все же написать функцию «в лоб», не используя мозг? Ну что тогда … успехов Вам в ожидании результатов!

T4.6. Приводится один из возможных вариантов программы.

```
function ЦифровойКорень(n: int64): int64;
begin
  if n < 10 then
    Result := n
  else
  begin
    var s: int64 := 0;
    while n > 0 do
    begin
      s := s + n mod 10;
      n := n div 10
    end;
    Result := ЦифровойКорень(s)
  end
end;

begin
  var n: int64;
  Write('Введите число: ');
  Read(n);
  n := Abs(n); // от злобных буратинок с отрицательными числами
  Println(ЦифровойКорень(n))
end.
```

14.5 К части 5

T5.1. Приводится один из возможных вариантов функции

```
function СлучайноеВещественное
    (n: integer; a, b: real; m: integer): sequence of real :=
    ArrRandomReal(n, a, b).Select(t -> Round(t, m));
```

T5.2. Приводится один из вариантов программы

```
begin
  var a := ArrRandom(ReadInteger('n ='), 10, 20);
  a.Println;
  var m := ReadInteger('Какое число ищем?');
  var k := a.Where(t-> t = m).Count;
  $'Число {m} встречается {k} раз'.Println
end.
```

T5.3. Приводится один из вариантов программы

```
begin
  var a := ArrRandom(ReadInteger('n ='), -99, 99);
  a.Println;
  a := a.Where(t -> t.InRange(15, 60)).ToArray;
  var m := a.Length;
  var r := (a.Aggregate(1.0, (p, q) -> p * q)) ** (1 / m);
  Println('Среднее геометрическое равно', r)
end.
```

T5.4. Простые множители можно получать по алгоритму, предложенному в задании Т3.4. Нужны не все множители числа, а лишь максимальный из них (он получается по алгоритму последним).

```
function НаибольшийПростой(n: integer): integer;
begin
  var d := 2;
  while n <> 1 do
  begin
    if n mod d = 0 then
      n := n div d
    else
    if d * d > n then
      d := n
    else
      d += 1
  end;
  Result := d;
end;

begin
  var a := ArrRandom(ReadInteger('n ='), 1, 1000000000);
  a.Println;
  a.Select(t -> НаибольшийПростой(t)).Max.Println
end.
```

T5.5. Координаты точек по оси x будем хранить в массиве *x*, координаты точек по оси y – соответственно в массиве *y*. Номера точек в программе совпадают с индексами массивов [0 .. n-1].

```
function L2(i, j: integer; x, y: array of integer) :=
    Sqr(x[j] - x[i]) + Sqr(y[j] - y[i]);

begin
  var n := ReadInteger('n =');
  var x := ArrRandom(n, -99, 99);
  var y := ArrRandom(n, -99, 99);
  for var i := 0 to n - 1 do
    Write('(', x[i], ',', y[i], ') ');
  Writeln;
  var im, jm: integer;
  var L2max := -1;
  for var i := 0 to n - 2 do
    for var j := i + 1 to n - 1 do
    begin
      var L2t := L2(i, j, x, y);
      if L2t > L2max then
        (L2max, im, jm) := (L2t, i, j)
    end;
  Writeln('Максимальное расстояние между точками ', im + 1, ' и ',
      jm + 1, ' равно ', Sqrt(L2max):0:2)
end.
```

T5.6. Воспользуемся генератором бесконечной последовательности, а количество членов отберем по условию TakeWhile. Это позволит написать «однострочное» решение.

```
begin
  0.Step.Select(t -> 1 / 2 ** t).TakeWhile(t -> t >= 1e-15).Sum.Println
end.
```

T5.7. Приводится один из возможных вариантов программы.

```
begin
  var n := ReadInteger('Число проб:');
  var k := SeqGen(n, t -> Random2(-1.0, 1.0))
      .Where(t -> t[0] ** 2 + t[1] ** 2 <= 1.0).Count;
  Writeln(4 * k / n:0:2)
end.
```

14.6 К части 6

T6.1. Приводится один из возможных вариантов программы.

```
begin
  var s := ReadlnString('->').Trim.ToWords.JoinIntoString(' ');
  s.Println
end.
```

T6.2. Приводится один из возможных вариантов программы.

```
begin
  var s := '  Да    охранюся я     от    мушек,  ' + NewLine +
    '  От    дев, не     знающих   любви,   ' + NewLine +
    '    От дружбы слишком    нежной и    —   ' + NewLine +
    'От   романтических    старушек.  ' + NewLine +
    ' ' * 20 + 'М.  Ю.  Лермонтов   ';
  var РазделителиСтрок := NewLine.ToCharArray;
  s := s.Trim.ToWords(РазделителиСтрок)
      .Select(t -> t.Trim.ToWords.JoinIntoString(' '))
      .JoinIntoString(NewLine);
  s.Println
end.
```

Т6.3. Приводится один из возможных вариантов программы.

```
begin
  var s := '  Да    охранюся я     от    мушек,  ' + NewLine +
    '  От    дев, не     знающих   любви,   ' + NewLine +
    '    От дружбы слишком    нежной и    –   ' + NewLine +
    'От   романтических    старушек.  ' + NewLine +
    ' ' * 20 + 'М.  Ю.  Лермонтов   ';
  var РазделителиСтрок := NewLine.ToCharArray;
  var n := s.ToWords(РазделителиСтрок).Last.MatchValue('\s+').Length;
  var a := s.Trim.ToWords(РазделителиСтрок)
      .Select(t -> t.Trim.ToWords.JoinIntoString(' ')).ToArray;
  a[a.High] := ' ' * n + a.Last;
  s := a.JoinIntoString(NewLine);
  s.Println
end.
```

Т6.4. Приводится один из возможных вариантов программы.

```
begin
  var s := '    Однажды  Лебедь,    Рак, да    Щука' + NewLine +
      '  Везти с  ' + Chr(9) + '  поклажей    воз взялись   ';
  var a := s.MatchValues('\w+').OrderByDescending(t -> t.Length)
      .Take(3).ToArray;
  foreach var w in a do
  begin
    s := s.Replace(w,'');
  end;
  s.Println
end.
```

Т6.5. Приводится один из возможных вариантов программы.

```
begin
  10000.Range.Aggregate(BigInteger(1), (p, q) -> p * q).ToString
      .Where(c -> c = '7').Count.Println
end.
```

14.7 К части 7

Т7.1. Для инициализации элементов массива можно воспользоваться конструктором записи. Массив описывается как поле типа **array of integer**, остальное выполняется в конструкторе, получающем посредством параметров количество элементов массива и диапазон для генерации случайных значений.

Конечно, это не лучший пример использования записей, поскольку в дополнение к собственно массиву температур мы получаем имя записи, которое постоянно приходится добавлять. Удобнее напрямую определить функцию, создающую и возвращающую инициализированный массив. Но можно представить себе усложненный вариант задачи, когда в записи будут поля с уже вычисленной средней температурой и т.п. В таких случаях использование записи окажется вполне оправданным.

```
type
  tRec = record
    mt: array of real;
    constructor(days: integer; tmin, tmax: real);
    begin
      mt := new real[days];
      for var i := 0 to mt.High do
        mt[i] := Round(Random(tmin, tmax), 1)
    end;
  end;

begin
  var (суток, минимум, максимум) := (30, 13.7, 24);
  var t := new tRec(суток, минимум, максимум);
  t.mt.Println;
  $'Средняя температура равна {t.mt.Average:f1}'.Println
end.
```

Т7.2. Для представления дробных чисел с произвольной разрядностью можно использовать запись с полями для целой части, дробной части и количеством нулей, находящимся между запятой и первой цифрой в дробной части, отличной от нуля. Целую и дробную части удобно разместить в данных типа **BigInteger**, а для количества нулей будет достаточно типа **integer**. Алгоритм получения суммы дробных частей слагаемых можно построить на основе обычного сложения «в столбик». Уделите внимание добавлению единицы в целую часть при переносе, а также выравниванию слагаемых в дробной части «по запятой» за счет приписывания нужного количества нулей справа.

```
type
  ДлинноеДробное = record
    ЦЧ: BigInteger;
    ДЧ: BigInteger;
    ВедущихНулей: integer;

    constructor(s: string);
    begin
      var ПозицияЗапятой := Pos(',',s);
      ВедущихНулей := 0;
      if ПозицияЗапятой = 0 then
      begin
        ЦЧ := BigInteger.Parse(s);
        ДЧ := BigInteger(0)
      end
      else
      begin
        ЦЧ := BigInteger.Parse(s[:ПозицияЗапятой]);
        s := s[ПозицияЗапятой + 1:];
        ДЧ := BigInteger.Parse(s);
        ВедущихНулей := ДЧ.IsZero ? 0 : s.Length - ДЧ.ToString.Length;
      end
    end;
```

```
    function ToString: string; override;
    begin
      Result := ЦЧ.IsZero ? '0' : ЦЧ.ToString;
      if ДЧ <> 0 then
        Result += ',' + ВедущихНулей * '0' + ДЧ.ToString
    end;

  end;

function СуммаЦЧ(A, B: ДлинноеДробное) := A.ЦЧ + B.ЦЧ;

function СуммаДЧ(A, B: ДлинноеДробное): ДлинноеДробное;
begin
  var Sa := A.ВедущихНулей * '0' + A.ДЧ.ToString;
  var Sb := B.ВедущихНулей * '0' + B.ДЧ.ToString;
  var (La, Lb) := (Sa.Length, Sb.Length);
  if La <> Lb then
  begin
    var Нулей := Abs(La - Lb);
    if La > Lb then
      Sb += Нулей * '0'
    else
      Sa += Нулей * '0';
    (La, Lb) := (Sa.Length, Sb.Length);
  end;
  var Сумма := BigInteger.Parse(Sa) + BigInteger.Parse(Sb);
  var t := Сумма.ToString;
  Result.ВедущихНулей := La - t.Length; // La = Lb
  if Result.ВедущихНулей = -1 then

  begin
    Delete(t,1,1);
    Result.ЦЧ := BigInteger(1);
    if t.Any(c->c <> '0') then
    begin
      Result.ДЧ := BigInteger.Parse(t);
      Result.ВедущихНулей := 0;
      while t[Result.ВедущихНулей+1] = '0' do
        Inc(Result.ВедущихНулей)
    end
    else
    begin
      Result.ДЧ := BigInteger(0);
      Result.ВедущихНулей := 0
    end
  end

  else
  begin
    Result.ЦЧ := BigInteger(0);
    Result.ДЧ := Сумма
  end
end;
```

```
function Add(A,B: ДлинноеДробное): ДлинноеДробное;
begin
  var t:= СуммаДЧ(A,B);
  Result.ЦЧ := СуммаЦЧ(A,B)+ t.ЦЧ;
  Result.ДЧ := t.ДЧ;
  Result.ВедущихНулей := t.ВедущихНулей
end;


begin
  var p := new ДлинноеДробное('0,999999999');
  var q := new ДлинноеДробное('0,000000001');
  Println(Add(p,q))
end.
```

Для самопроверки приводятся результаты, выданные программой на предложенных тестовых данных:

```
1) 648281630499284914830228879684528 4,92176998421314961132603106774 32
2) 243539156774719384715150521402433838481 81,048808800869094991377115867262
3) 627730409266514495483083635192093750208348093,795713648375157400281354681
4) 1,24
5) 2,0009
6) 1,20015
7) 875766109119982744146465
8) 0,0000301
9) 1
```

Т7.3. Некоторые трудности могут возникнуть, если не использовать «культуру» System.Globalization.CultureInfo.GetCultureInfo('ru-RU'), позволяющую работать с привычными форматами представления даты. Необходимую для этого информацию обычно получают из Интернет.

```
function DOW(date: string): string; // формат дд.мм.гггг
begin
  var cu := System.Globalization.CultureInfo.GetCultureInfo('ru-RU');
  var dt := DateTime.ParseExact(date, 'd', cu);
  Result := Format(dt.ToString('ddd', cu))
end;

begin
  DOW('01.01.2020').Println
end.
```

T7.4. Приводится один из возможных вариантов программы. Для вывода номера дня и наименования месяца используется формат даты ‘dd.MMM’.

```
begin
  var cu := System.Globalization.CultureInfo.GetCultureInfo('ru-RU');
  var Год := ReadInteger('Укажите четыре цифры года:');
  var dt := '01.01.' + Год;
  var dat := DateTime.ParseExact(dt, 'd', cu);
  while Format(dat.ToString('ddd', cu)) <> 'Пт' do
    dat := dat.AddDays(1);
  var dtEnd := DateTime.ParseExact('31.12.' + Год, 'd', cu);
  while dat <= dtEnd do
  begin
    Print(Format(dat.ToString('dd.MMM', cu)));
    dat := dat.AddDays(7)
  end
end.
```

Пример вывода для 2019 года:

```
Укажите четыре цифры года: 2019
04.янв 11.янв 18.янв 25.янв 01.фев 08.фев 15.фев 22.фев 01.мар 08.мар 15.мар
22.мар 29.мар 05.апр 12.апр 19.апр 26.апр 03.май 10.май 17.май 24.май 31.май
07.июн 14.июн 21.июн 28.июн 05.июл 12.июл 19.июл 26.июл 02.авг 09.авг 16.авг
23.авг 30.авг 06.сен 13.сен 20.сен 27.сен 04.окт 11.окт 18.окт 25.окт 01.ноя
08.ноя 15.ноя 22.ноя 29.ноя 06.дек 13.дек 20.дек 27.дек
```

14.8 К части 8

T8.1. Здесь можно построить функцию, которая для a_{ij} возвращает необходимый символ.

$$a_{i,j} = \begin{cases} *, \text{ при } (j=i) \vee (j=n-i-1),\ i=0..n-1 \\ 1,\ \text{при } i+1 \le j \le n-2-i \\ 2,\ \text{при } (n-i \le j \le n-1) \wedge (i > j) \\ 3,\ \text{при } (n-i \le j \le n-1) \wedge (i < j) \\ 4, \text{ в прочих случаях} \end{cases}$$

К сожалению, многочисленные ветвления по сложным условиям – не самая сильная сторона языка Паскаль, поэтому в программной реализации код выглядит несколько запутанным. Сформированные отступы позволяют сделать логику программы более наглядной.

```
begin
  var n := ReadInteger('n =');
  var a := new char[n, n];
  for var i := 0 to n - 1 do
    for var j := 0 to n - 1 do
      if (j = i) or (j = n - 1 - i) then
        a[i, j] := '*'
      else
        if (j >= i + 1) and (j <= n - 2 - i) then
          a[i, j] := '1'
        else
          if (j >= n - i) and (j <= n - 1) then
            if j > i then
              a[i, j] := '2'
            else
            begin
              if j < i then
                a[i, j] := '3'
            end
          else a[i, j] := '4';
  a.Println(2);
end.
```

T8.2. Приводится один из возможных вариантов.

```
begin
  var (Год, Месяц) := ReadInteger2('Укажите год и месяц:');
  var ПервоеЧисло := new DateTime(Год, Месяц, 1);
  var ВсегоДней := DateTime.DaysInMonth(Год, Месяц);
  var НомерДняНедели := integer(ПервоеЧисло.DayOfWeek) - 1;
  if НомерДняНедели = -1 then
    НомерДняНедели := 6;
  var Колонок := Ceil((НомерДняНедели + ВсегоДней) / 7) + 1;
  var a := new string[7, Колонок];
  a.SetCol(0, 'Пн Вт Ср Чт Пт Сб Вс'.ToWords);
  var d := 1;
  for var i := НомерДняНедели to 6 do
  begin
    a[i, 1] := d.ToString;
    Inc(d)
  end;
  for var j := 2 to Колонок - 1 do
  begin
    for var i := 0 to 6 do
    begin
      a[i, j] := d.ToString;
      Inc(d);
      if d > ВсегоДней then break
    end;
    if d > ВсегоДней then break
  end;
  a.Println(3)
end.
```

T8.3. Приводится один из возможных вариантов.

```
procedure Delete(r, c: integer; var a: array[,] of integer);
begin
  var (m, n) := (a.RowCount, a.ColCount);
  var t := new integer[m - 1, n - 1];
  var (it, jt) := (0, 0);
  for var i := 0 to m - 1 do
    if i <> r then
    begin
      for var j := 0 to n - 1 do
        if j <> c then
        begin
          t[it, jt] := a[i, j];
          Inc(jt)
        end;
      Inc(it);
      jt := 0;
    end;
  a := t
end;

begin
  var (m, n) := ReadInteger2('Число строк и столбцов массива:');
  var a := MatrRandom(m, n, -99, 99);
  Writeln('Исходный массив');
  a.Println(4);
  Writeln('-' * 4 * n);
  var v := a.Rows.Select(row -> row.Average).ToArray;
  for var row := 0 to m - 1 do
    for var col := 0 to n - 1 do
      if Abs(a[row, col] - v[row]) > 75 then a[row, col] := 0;
  while True do
  begin
    var w := a.ElementsWithIndices.Where(t -> t[0] = 0).ToArray;
    if w.Count = 0 then break;
    Delete(w[0][1], w[0][2], a)
  end;
  Writeln('Результирующий массив');
  a.Println(4)
end.
```

14.9 К части 9

T9.1. Приводится один из возможных вариантов. При использовании текстовых файлов для хранения вещественных значений может возникать дополнительная погрешность вычислений из-за преобразований между двоичным и текстовым представлением числовых значений. Данное вещественного типа занимает 8 байт памяти, а его символьное представление – до 16-17 байт. Кроме этого, в текстовом файле значения разделяются, а символы-разделители также занимают место.

```
begin
  var fb := CreateBinary('MyFile.bin'); // бестиповый
  var ft := OpenWrite('MyFile.txt'); // текстовый
  var s := 0.0;
  loop 1000000 do
  begin
    var x := Random(-9.0, 9.0);
    s += x;
    fb.Write(x);
    ft.Writeln(x);
  end;
  Println(s);
  fb.Seek(0);
  var sf := 0.0;
  while not Eof(fb) do
    sf += fb.ReadReal;
  Println(sf, s-sf);
  fb.Close;
  ft.Close;
  ft := OpenRead('MyFile.txt');
  sf := 0.0;
  while not Eof(ft) do
    sf += ft.ReadlnReal;
  Println(sf, s-sf);
  ft.Close;
end.
```

T9.2. Приводится один из возможных вариантов. Площадь треугольника вычисляется по известной формуле Герона. Если задание выполнено правильно, получится результат в интервале [1160; 1166].

```
function ТреугольникВозможен(p: string): boolean;
begin
  var m := p.ToWords;
  var a := m[0].ToInteger;
  var b := m[1].ToInteger;
  var c := m[2].ToInteger;
  Result := (a + b > c) and (a + c > b) and (b + c > a)
end;

function Площадь(p: string): real;
begin
  var m := p.ToWords;
  var a := m[0].ToInteger;
  var b := m[1].ToInteger;
  var c := m[2].ToInteger;
  var pp := (a + b + c) / 2;
  Result := Sqrt(pp * (pp - a) * (pp - b) * (pp - c))
end;
```

```
begin
  var n := 500000;
  var fn := 'MyFile.txt';
  var f := OpenWrite(fn);
  loop n do
  begin
    var r := Random3(1, 99);
    Writeln(f, r[0], ' ', r[1], ' ', r[2])
  end;
  f.Close;
  f := OpenWrite('Temp.tmp');
  foreach var s in ReadAllLines(fn) do
    if ТреугольникВозможен(s) then
      f.Writeln(s);
  f.Close;
  DeleteFile(fn);
  Rename(f, fn);
  var s := ReadAllLines(fn).Select(t -> Площадь(t)).Average;
  $'Средняя площадь равна {s:f3}'.Println
end.
```

Т9.3. Сравнение текстов программ Т9.3 и Т9.2 показывает, что в данном случае использование типизированных файлов не дает каких-то особых преимуществ перед текстовыми.

```
type
  R = record
  a, b, c: integer;
  end;

function ТреугольникВозможен(p: R): boolean;
begin
  var (a, b, c) := (p.a, p.b, p.c);
  Result := (a + b > c) and (a + c > b) and (b + c > a)
end;

function Площадь(p: R): real;
begin
  var (a, b, c) := (p.a, p.b, p.c);
  var pp := (a + b + c) / 2;
  Result := Sqrt(pp * (pp - a) * (pp - b) * (pp - c))
end;
```

```
begin
  var n := 500000;
  var fn := 'MyFile.bin';
  var d: R; // буфер обмена данными
  var f: file of R;
  Assign(f, fn);
  f.Rewrite;
  loop n do
  begin
    (d.a, d.b, d.c) := Random3(1, 99);
    Write(f, d);
  end;
  var ft: file of R;
  Assign(ft, 'Temp.tmp');
  ft.Rewrite;
  f.Seek(0);
  while not Eof(f) do
  begin
    Read(f, d);
    if ТреугольникВозможен(d) then
      Write(ft, d)
  end;
  ft.Close;
  f.Close;
  DeleteFile(fn);
  Rename(ft, fn);
  var s := f.Elements&<R>.Select(t -> Площадь(t)).Average;
  $'Средняя площадь равна {s:f3}'.Println
end.
```

T9.4. Проще всего отфильтровать слова при помощи регулярного выражения. Сформируем на основе каждого файла массив уникальных слов (.Distinct), а затем найдем пересечение этих массивов, (.Intersect), отсортируем результат в алфавитном порядке, поместим его в текстовый файл, и выведем посредством .Println.

Исходные файлы взяты из свободного доступа. Они содержат вторую главу поэмы А. С. Пушкина «Евгений Онегин» и две первых части поэмы М. Ю. Лермонтова «Корсар». Результирующий файл должен содержать 186 слов.

```
begin
  var Шаблон := '[^\WA-z]{2,}';
  var s1 := ReadAllText('T9.4a.txt').MatchValues(Шаблон)
      .Select(t -> t.ToLower).Distinct;
  var s2 := ReadAllText('T9.4b.txt').MatchValues(Шаблон)
      .Select(t -> t.ToLower).Distinct;
  s1.Intersect(s2).Order.Println.WriteLines('T9.4c.txt')
end.
```

14.10 К части 10

T10.1. Приводится один из возможных вариантов.

```
type
  tDeque = record
    Голова, Хвост: Stack<integer>;
    constructor;
    begin
      Голова := new Stack<integer>;
      Хвост := new Stack<integer>
    end;
  end;

procedure ВХвост(Дек: tDeque; x: integer);
begin
  Дек.Хвост.Push(x)
end;

procedure ВГолову(Дек: tDeque; x: integer);
begin
  Дек.Голова.Push(x)
end;

function ИзХвоста(Дек: tDeque): integer;
begin
  if Дек.Хвост.Count > 0 then
    Result := Дек.Хвост.Pop
  else
  begin
    while Дек.Голова.Count>1 do
      Дек.Хвост.Push(Дек.Голова.Pop);
    Result := Дек.Голова.Pop
  end
end;

function ИзГоловы(Дек: tDeque): integer;
begin
  if Дек.Голова.Count > 0 then
    Result := Дек.Голова.Pop
  else
  begin
    while Дек.Хвост.Count>1 do
      Дек.Голова.Push(Дек.Хвост.Pop);
    Result := Дек.Хвост.Pop
  end
end;
```

```
begin
  var МойДек := new tDeque;
  ВХвост(МойДек,1);
  ВХвост(МойДек,2);
  ВХвост(МойДек,3);
  ВГолову(МойДек,4);
  ВХвост(МойДек,5);
  ИзГоловы(МойДек).Print;
  ИзХвоста(МойДек).Print;
  ИзГоловы(МойДек).Print;
  ИзХвоста(МойДек).Print;
  ИзХвоста(МойДек).Println
end.
```

T10.2. Выбран алгоритм, использующий дополнительную очередь.

```
begin
  var Q := new Queue<integer>;
  Loop 25 do
    Q.Enqueue(Random(10,99));
    Q.Println;
  var max := Q.Max;
  var Qt := new Queue<integer>;
  while Q.Peek <> max do
    Qt.Enqueue(Q.Dequeue);
  while Qt.Count > 0 do
    Q.Enqueue(Qt.Dequeue);
  Q.Println;
end.
```

T10.3. Приводится один из возможных вариантов.

```
begin
  var L := SeqRandom(20, -99, 99).ToList;
  L.Println;
  var (pb, pe) := (0, L.Count - 1);
  while pb < pe do
  begin
    while (L[pb] >= 0) and (pb < pe) do
      Inc(pb);
    while (L[pe] < 0) and (pb < pe) do
      Dec(pe);
    if pb < pe then
    begin
      (L[pb], L[pe]) := (L[pe], L[pb]);
      Inc(pb);
      Dec(pe)
    end
  end;
  L.Println;
end.
```

T10.4. Приводится один из возможных вариантов.

```
function ЧислоПростое(Число: integer): boolean;
// является ли натуральное число простым?
begin
  if Число < 2 then
    Result := False
  else
    if Число < 4 then
      Result := True
    else
    begin
      var НеДелится := Число.IsOdd;
      var Делитель := 3;
      while НеДелится and (Sqr(Делитель) <= Число) do
      begin
        НеДелится := Число mod Делитель <> 0;
        Делитель += 2
      end;
      Result := НеДелится
    end
end;

begin
  var Множество := SeqRandom(15, 1, 100000).ToHashSet;
  Множество.Println;
  var ПодмножествоПростых := new HashSet<integer>;
  var ПодмножествоСоставных := new HashSet<integer>;
  foreach var Элемент in Множество do
    if ЧислоПростое(Элемент) then
      ПодмножествоПростых += Элемент
    else
      ПодмножествоСоставных += Элемент;
  Print('Простые:');
  ПодмножествоПростых.Println;
  Print('Составные:');
  ПодмножествоСоставных.Println
end.
```

14.11 К части 11

T11.1. Приводится один из возможных вариантов. Правило Крамера порождает нудную, длинную и однообразную писанину, если вычислять определители, рпаскрывая их по строке или по столбцу. Очень легко допустить ошибки в формулах, поэтому написанную программу нужно прогонять через набор тестов значительного объема. А как их вычислять по-другому, обычно изучают только в вузах (метод Ньютона и т.п.).

```
uses Kramer;

begin
  var a1: array[,] of real := ((6, -5), (1, 3));
  var b1 := Arr(23.0, 8.0);
  var r1 := Solve(a1, b1);
  r1.Println;
  var a2: array[,] of real := ((3, -2, 5), (7, 4, -8), (5, -3, -4));
  var b2 := Arr(7.0, 3.0, -12.0);
  var r2 := Solve(a2, b2);
  r2.Println
end.
```

Модуль Kramer должен находиться в файле Kramer.pas.

```
unit Kramer;

function Solve(a: array[,] of real; b: array of real): array of real;
begin
  case b.Length of
    3:
      begin
        var d := a[0, 0] * (a[1, 1] * a[2, 2] - a[2, 1] * a[1, 2]) -
            a[0, 1] * (a[1, 0] * a[2, 2] - a[2, 0] * a[1, 2]) +
            a[0, 2] * (a[1, 0] * a[2, 1] - a[2, 0] * a[1, 1]);
        if d = 0 then
          Result := new real[0]
        else
        begin
          var d1 := b[0] * (a[1, 1] * a[2, 2] - a[2, 1] * a[1, 2]) -
              a[0, 1] * (b[1] * a[2, 2] - b[2] * a[1, 2]) +
              a[0, 2] * (b[1] * a[2, 1] - b[2] * a[1, 1]);
          var d2 := a[0, 0] * (b[1] * a[2, 2] - b[2] * a[1, 2]) -
              b[0] * (a[1, 0] * a[2, 2] - a[2, 0] * a[1, 2]) +
              a[0, 2] * (a[1, 0] * b[2] - a[2, 0] * b[1]);
          var d3 := a[0, 0] * (a[1, 1] * b[2] - a[2, 1] * b[1]) -
              a[0, 1] * (a[1, 0] * b[2] - a[2, 0] * b[1]) +
              b[0] * (a[1, 0] * a[2, 1] - a[2, 0] * a[1, 1]);
          Result := Arr(d1 / d, d2 / d, d3 / d);
        end
      end;
```

```
      2:
        begin
          var d := a[0, 0] * a[1, 1] - a[1, 0] * a[0, 1];
          if d = 0 then
            Result := new real[0]
          else
          begin
            var d1 := b[0] * a[1, 1] - b[1] * a[0, 1];
            var d2 := b[1] * a[0, 0] - b[0] * a[1, 0];
            Result := Arr(d1 / d, d2 / d);
          end
        end;
    else
      Result := new real[0];
    end
  end;

  end.
```

Приведенная программа выведет следующие строки:

```
4.73913043478261 1.08695652173913
1 3 2
```

Для проверки полученных результатов следует решить системы уравнений каким-либо способом – вручную, с помощью Интернет-сервисов и т.п. Поскольку в состав PascalABC.NET входит пакет численных методов, можно воспользоваться им, внеся в программу Т11.1 небольшие изменения:

```
uses NumLibABC;

begin
  var a1: array[,] of real := ((6, -5), (1, 3));
  var oD := new Decomp(a1);
  var b1 := Arr(23.0, 8.0);
  oD.Solve(b1);
  b1.Println;
  var a2: array[,] of real := ((3, -2, 5), (7, 4, -8), (5, -3, -4));
  oD := new Decomp(a2);
  var b2 := Arr(7.0, 3.0, -12.0);
  oD.Solve(b2);
  b2.Println
end.
```

Результаты полностью совпадают с ранее полученными.

14.12 К части 13

T13.1. Приводится один из возможных вариантов.

```
type
  Tabula = auto class
  public
    a, b, h: real;
    f: real-> real;

    function ToTab: sequence of (real, real);
    begin
      if h <> 0 then
      begin
        if b < a then Swap(a, b);
        var n := Trunc((b - a) / h) + 1;
        for var i := 1 to n do
        begin
          var x := a + h * (i - 1);
          yield (x, f(x))
        end
      end;
    end;

  end;


function OutTab(Self: sequence of (real, real); mx, nx, my, ny: integer):
    sequence of(real, real); extensionmethod;
begin
  foreach var s in Self do
    Writeln(s[0]:mx:nx, s[1]:my:ny);
  Result := Self
end;

begin
  var t := new Tabula(-4 * Pi, 5 * Pi, pi / 3,
      x -> 3 * x * Cos(x) ** 2 / (1.1 + Sin(x)));
  t.ToTab.OutTab(13, 9, 20, 9).Select(t -> t[1]).Max.Println;
end.
```

В качестве примера выполнена табуляция функции

$$y = \frac{3x\cos^2 x}{1.1+\sin x},\ \text{для}\ x = -4\pi\left(\frac{\pi}{3}\right)5\pi$$

Также, по результатам табуляции найдено максимальное значение функции на указанном интервале.

```
-12.566370614        -34.271919857
-11.519173063         -4.394337825
-10.471975512         -3.994852568
 -9.424777961        -25.703939893
 -8.377580410        -26.854134632
 -7.330382858        -23.497367803
 -6.283185307        -17.135959929
 -5.235987756         -1.997426284
 -4.188790205         -1.597941027
 -3.141592654         -8.567979964
 -2.094395102         -6.713533658
 -1.047197551         -3.356766829
  0.000000000          0.000000000
  1.047197551          0.399485257
  2.094395102          0.798970514
  3.141592654          8.567979964
  4.188790205         13.427067316
  5.235987756         16.783834145
  6.283185307         17.135959929
  7.330382858          2.796396798
  8.377580410          3.195882055
  9.424777961         25.703939893
 10.471975512         33.567668290
 11.519173063         36.924435119
 12.566370614         34.271919857
 13.613568166          5.193308339
 14.660765717          5.592793596
 15.707963268         42.839899822
42.8398998216788
```

T13.2. Приводится один из возможных вариантов.

```
type
  tRoot = abstract class
    function GetRoots: array of real; abstract;
    procedure OutRoots(t: array of real); abstract;
  end;

  tLinear = class(tRoot)
    private
      coef: array of real;
    public
      constructor(coef: array of real);
      begin
        if coef.Length = 2 then
          Self.coef := coef;
      end;

      function GetRoots: array of real; override;
      begin
        if coef <> nil then
          Result := Arr(-coef[1]/coef[0])
        else
          Result := nil
      end;

      procedure OutRoots(a: array of real); override;
      begin
        if a <> nil then
        begin
          var s := $'Корень уравнения {coef[0]}x ';
          if coef[1]> 0 then
            s += $'+ {coef[1]} = 0'
          else
            if coef[1] = 0 then
              s+= '= 0'
            else
              s += $'- {-coef[1]} = 0';
          s += $' равен {a[0]}';
          Println(s)
        end
        else
          Println('Неверно заданы коэффицианты уравнения')
      end;
  end;
```

```
  tSquare = class(tRoot)
    private
      coef: array of real;
    public
      constructor(coef: array of real);
      begin
        if coef.Length = 3 then
          Self.coef := coef;
      end;

      function GetRoots: array of real; override;
      begin
        if coef <> nil then
        begin
          var D := Sqrt(Sqr(coef[1]) - 4*coef[0]*coef[2]);
          Result := Arr((-coef[1]-D)/2/coef[0], (-coef[1]+D)/2/coef[0])
        end
        else
          Result := nil
      end;

      procedure OutRoots(a: array of real); override;
      begin
        if a <> nil then
        begin
          var s := $'Корни уравнения {coef[0]}x^2 ';
          if coef[1]> 0 then
            s += $'+ {coef[1]}x '
          else
            if coef[1] < 0 then
              s += $'- {-coef[1]}x ';
          if coef[2]> 0 then
            s += $'+ {coef[2]} = 0'
          else
            if coef[2] < 0 then
              s += $'- {-coef[2]} = 0';
          s += $' равны {a[0]}, {a[1]}';
          Println(s)
        end
        else
          Println('Неверно заданы коэффициенты уравнения')
      end;
  end;

begin
  var obj := new tLinear(Arr(1.2, 3.6));
  var a := obj.GetRoots;
  obj.OutRoots(a);
  var obj1 := new tSquare(Arr(2.0, -7.0, -15.0));
  a := obj1.GetRoots;
  obj1.OutRoots(a)
end.
```

Приведенная программа дала следующие решения:

```
Корень уравнения 1.2x + 3.6 = 0 равен -3
Корни уравнения 2x^2 - 7x - 15 = 0 равны -1.5, 5
```

T13.3. Приводится один из возможных вариантов.

```
type
  ВычислитьПОЛИЗ = class
    private
      Слова: array of string;
      Стек: Stack<integer>;
    public
      constructor(ПОЛИЗ: string);
      begin
        Слова := ПОЛИЗ.ToWords;
        Стек := new Stack<integer>;
      end;

      function Вычислить: integer;
      begin
        foreach var Слово in Слова do
        begin
          var Целое: integer;
          if Слово.TryToInteger(Целое) then
            Стек.Push(Целое)
          else
            case Слово[1] of
            '+': Стек.Push(Стек.Pop + Стек.Pop);
            '-':
              begin
                var Вычитаемое := Стек.Pop;
                Стек.Push(Стек.Pop - Вычитаемое)
              end;
            '*': Стек.Push(Стек.Pop * Стек.Pop);
            '/':
              begin
                var Делитель := Стек.Pop;
                Стек.Push(Стек.Pop div Делитель)
              end
            end
        end;
        Result := Стек.Pop
      end;
end;

begin
  var Объект := new ВычислитьПОЛИЗ('36 18 3 * - 5 4 7 5 - * - /');
  Объект.Вычислить.Println
end.
```

T13.4. В этом задании всего лишь нужно реализовать необходимые расширения для List<T> по примитивным алгоритмам.

```
uses NumLibABC;

function Sum(Self: List<Fraction>): Fraction;
    extensionmethod;
begin
  Result := new Fraction;
  foreach var d in Self do
    Result += d
end;

function Average(Self: List<Fraction>): Fraction;
    extensionmethod :=
Self.Sum / Self.Count;

function Min(Self: List<Fraction>): Fraction;
    extensionmethod;
begin
  Result := Self[0];
  for var i := 1 to Self.Count - 1 do
    if Self[i] < Result then
      Result := Self[i]
end;

function Max(Self: List<Fraction>): Fraction;
    extensionmethod;
begin
  Result := Self[0];
  for var i := 1 to Self.Count - 1 do
    if Self[i] > Result then
      Result := Self[i]
end;

begin
  var L := new List<Fraction>;
  var n := ReadInteger('n =');
  loop n do
    L.Add(Frc(Random(-99, 99), Random(1, 99)));
  Writeln(L);
  var s := L.Sum;
  Println('Сумма равна', s, '=', s.ToReal);
  s := L.Average;
  Println('Среднее равно', s, '=', s.ToReal);
  s := L.Min;
  Println('Минимум равен', s, '=', s.ToReal);
  s := L.Max;
  Println('Максимум равен', s, '=', s.ToReal)
end.
```

T13.5. Приводится один из возможных вариантов.

```
uses NumLibABC;

type
  Frac = class(Fraction, IComparable<Frac>)
  public
    function CompareTo(elem: Frac): integer;
    begin
      if Fraction(Self) = Fraction(elem) then
        Result := 0
      else
      if Fraction(Self) > Fraction(elem) then
        Result := 1
      else
        Result := -1
    end;

  end;

function Sum(Self: List<Frac>): Fraction;
    extensionmethod;
begin
  Result := new Fraction;
  foreach var d in Self do
    Result += Fraction(d)
end;

begin
  var L := new List<Frac>;
  var n := ReadInteger('n =');
  loop n do
    L.Add(new Frac(Random(-99, 99), Random(1, 99)));
  L.Println;
  var s := L.OrderDescending.ToArray;
  L := (s.Take(3)+s.TakeLast(2)).ToList;
  L.Println;
  var a := L.Sum;
  Println('Сумма равна', a, '=', a.ToReal);
end.
```

Ниже показан пример работы программы.

```
n = 12
-4/27 -16/99 82/39 -49/43 7/5 23/19 -31/91 9/11 -11/8 2/3 9 47/42
9 82/39 7/5 -49/43 -11/8
Сумма равна 669997/67080 = 9.98802921884317
```

Приложения

Приложение 1

Ключевые (зарезервированные) слова языка PascalABC.NET

Следующие ключевые слова зарезервированы в языке и не могут быть непосредственно использованы в качестве имен:

```
and array as auto begin case class const constructor destructor div do downto
else end event except extensionmethod file finalization finally for foreach
function goto if implementation in inherited initialization interface is label
lock loop mod nil not of operator or procedure program property raise record
repeat sealed set sequence shl shr sizeof template then to try type typeof
until uses using var where while with xor
```

В случае необходимости следует экранировать ключевые слова, размещая перед ними символ &, например, &begin, &array и т.п.

Имеются также ***контекстно-ключевые слова***, которые считаются ключевыми только в определенном контексте. Их можно использовать в качестве имен без экранирования.

```
abstract default external forward internal on overload override params private
protected public read reintroduce unit virtual write
```

Экранирование также может использоваться при совпадении ключевых слов с именами Microsoft .NET.

Приложение 2

Информация по обновлению версий PascalABC.NET

Версии обновляются с официального сайта http://pascalabc.net. Ниже кратко описано обновление версии из графической оболочки.

- В меню *Помощь* выберите пункт *Проверить обновления*;
- При наличии обновления появится всплывающее окно с указанием версии доступного обновления;
- В случае согласия на обновление в оболочке будет открыта часть страницы официального сайта для выбора одной из семи опций:
  - PascalABC.NET + Microsoft .NET Framework (для первичной установки в операционной системе версии, старшей чем Windows XP);
  - PascalABC.NET StandardPack (для последующих установок в операционной системе версии, старшей чем Windows XP, если нужно устанавливать электронный задачник PT4);
  - Файлы только для инсталляции в дисплейных классах мехмата ЮФУ;
  - PascalABC.NET MiniPack (для последующих установок в операционной системе версии, старшей чем Windows XP, если не нужна установка электронного задачника PT4);

  - PABCNETC.ZIP – архив для установки консольного компилятора и компонент поддержки, предназначенный для операционных систем на базе Linux;
  - PascalABC.NET + Microsoft .NET Framework 4.0 (для первичной установки и обновлений в операционной системе Windows XP);
  - PascalABCNET.chm (для загрузки файла справки).
- После выбора нужной опции можно загрузить файл с обновлением или сразу же провести обновление. В случае загрузки файл только сохраняется на компьютере в отведенной для загрузок папке. При проведении обновления файл загружается в рабочую папку, распаковывается и запускается. Чтобы провести обновление, потребуется закрыть оболочку после получения соответствующего сообщения.

Можно провести обновление принудительно, зайдя на официальный сайт PascalABC.NET и выбрав на главной странице кнопку «Скачать». В этом случае, в зависимости от настроек операционной системы, будет выдано приглашение скачать файл или сразу же запустить обновление. Если файл был скачан, откройте его и запустите на выполнение.

В процессе установки (обновления) отключайте резидентные антивирусные средства, выполняющие онлайн-проверку, если не уверены в их надежной работе. На форуме по PascalABC.NET не раз задавались вопросы, связанные с некорректной работой компилятора или откомпилированных программ и часто в этом был виноват установленный в системе антивирус невысокого качества.

Приложение 3

Некоторые возможности графической оболочки PascalABC.NET

Работа с оболочкой PascalABC.NET описана в большом количестве книг и методических разработок, например, в размещенной на официальном сайте PascalABC.NET книге Валерия Рубанцева «Занимательные уроки с Паскалем». В связи с этим было принято решение упомянуть лишь наиболее часто употребляемые приемы и возможности.

Исходный текст новой программы может быть набран непосредственно во встроенном редакторе и это наиболее рекомендуемый способ, поскольку во время работы система помощи Intellisence будет давать подсказки. Кроме этого, во время набора автоматически производится создание отступов, выделяющих логическую структуру программного кода.

Существующий текст можно загрузить в редактор, для чего он (текст) должен иметь расширение .pas и .cs. Последнее расширение используется для программ, написанных на языке C#, которые оболочка умеет понимать, форматировать и даже выполнять с определенными ограничениями.

Подготовленный текст программы можно сохранить, но для выполнения программы сохранение не требуется. Программу можно откомпилировать, получив исполняемый файл, модуль или бибилиотеку dll – это определяется написанным кодом. Настройки (меню Сервис – Настройки – Опции компиляции») позволяют определить, нужно ли удалять после выполнения созданный выполняемый файл, а также управляют режимом отладки.

Выполнять программу можно под оболочкой, запустив ее при помощи клавиши {F9}, либо без связи с оболочкой, используя комбинацию клавиш {Shift}{F9}. В последнем случае вывод производится в отдельном консольном окне.

Возможности отладчика достаточно обширны. Если нужно выполнить по шагам программу от самого начала, запустите ее клавишей {F8}. За каждое нажатие этой клавиши выполняется ровно один оператор основной программы. Приостановок в вызываемых программных единицах нет. Если требуется пройти и по вызываемой программе, в строке с ее вызовом нажмите {F7}, а после перехода в нужный код можно продолжать нажимать {F8}, в противном случае по {F7} вы зайдете в системные библиотеки и рискуете там потеряться. Можно также установить курсор на любой строке кода и нажать {F4} – программа начнет выполняться и приостановится на отмеченной строке. Клавиша {F10} позволит выйти из подпрограммы и остановиться.

А теперь представьте, что в коде программы встретился цикл на сотню тысяч проходов. Сто тысяч раз нажимать клавишу {F8}? На этот случай отладчик предусматривает расстановку ***точек прерывания*** – строк кода, на которых выполнение программы будет приостановлено. Точки прерывания – они и есть точки: крупные и красные, расположенные на левой рамке окна редактора. Расставляются и удаляются щелчком мышки по рамке. После остановки на точке дальнейшее выполнение программы можно продолжить по упомянутым выше клавишам. Как уже упоминалось, можно организовать точку остановки по курсору и нажатию {F4}.

В точке прерывания можно открыть окно «Локальные переменные» и просмотреть типы и значения всех переменных, которые видны в этой точке. Окно «Просмотр выражений» позволяет оперативно ввести интересующее выражение и отобразить его результат. В выражении нельзя обращаться к пользовательским функциям (но можно использовать стандартные, например Sin, Sqrt и т.д.) или операциям, требующим компиляции кода, например, строить лямбда-выражения. Если программа работает с большим массивом A, можно отобразить его элемент, введя выражение по типу A[183]. Срезы в качестве выражения для просмотра использовать нельзя. После просмотра не забудьте снова переключиться на окно вывода. Полный перечень окон доступен в меню «Вид».

Приложение 4

Математические подпрограммы

А. Функции в алфавитном порядке

Abs(x: число) – возвращает модуль (абсолютную величину) числа x;
Abs(x: Complex): Complex – возвращает модуль комплексного числа x;
ArcCos(x: real) – возвращает угол в радианах, косинус которого равен x. Значение аргумента x должно находиться в пределах от -1 до 1;
ArcSin(x: real) – возвращает угол в радианах, синус которого равен x. Значение аргумента x должно находиться в пределах от -1 до 1;
ArcTan(x: real) – возвращает угол в радианах, тангенс которого равен x;
Ceil(x: real): integer – возвращает наименьшее целое, не меньшее, чем x («потолок»);
Conjugate(x: Complex) – возвращает число, комплексно-сопряженное с x;
Cos(x: real) – возвращает косинус угла x, заданного в радианах;
Cos(x: Complex) – возвращает косинус комплексного угла x, заданного в радианах;
Cosh(x: real) – возвращает гиперболический косинус угла x, заданного в радианах;
Cplx(re, im: real): Complex – конструирует комплексное число с вещественной частью re и мнимой частью im;
CplxFromPolar(magnitude, phase: real): Complex – конструирует комплексное число по его полярным координатам;
DegToRad(x: real) – переводит значение угла из градусной меры в радианную;
Exp(x: real) – возвращает экспоненту числа x;
Exp(x: Complex) – возвращает экспоненту комплексного числа x;
Floor(x: real): integer - возвращает наибольшее целое, не превышающее x («пол»);
Frac(x: real) – возвращает дробную часть числа x;
Int(x: real): integer – возвращает целую часть числа x;
Ln(x: real) – возвращает натуральный логарифм числа x;
Ln(x: Complex) – возвращает натуральный логарифм комплексного числа x;
Log(x: real) – возвращает десятичный логарифм числа x;
Log(x: Complex) – возвращает десятичный логарифм комплексного числа x;
Log10(x: real) – возвращает десятичный логарифм числа x;
Log10(x: Complex) – возвращает десятичный логарифм комплексного числа x;
Log2(x: real) – возвращает логарифм числа x по основанию 2;
LogN(основание, x : real) – возвращает логарифм числа x по указанному основанию;
Max(a, b: число) – возвращает максимальное из чисел a и b;
Min(a, b: число) – возвращает минимальное из чисел a и b;
Odd(i: целое) – возвращает True, если i нечетно и False в противном случае;
Power(x, y: real) – возвращает x в степени y;
Power(x, y: Complex) – возвращает x в степени y для комплексных чисел;
Power(x: real; n: integer) – возвращает x в степени n;
Power(x: BigInteger; n: integer) – возвращает x в степени n;
RadToDeg(x: real) – переводит значение угла из радианной меры в градусную;
Random(МаксимальноеЗначение: integer) – возвращает случайное целое в диапазоне [0, МаксимальноеЗначение – 1];

Random(МаксимальноеЗначение: real) – возвращает случайное вещественное число в диапазоне [0, МаксимальноеЗначение];
Random(a, b: integer) – возвращает случайное целое в диапазоне [a, b];
Random(a, b: real) – возвращает случайное вещественное число в диапазоне [a, b];
Random( ) – возвращает случайное вещественное число в диапазоне [0, 1];
Random2(МаксимальноеЗначение: integer) – возвращает кортеж из двух случайных целых чисел в диапазоне [0, МаксимальноеЗначение – 1];
Random2(МаксимальноеЗначение: real) – возвращает кортеж из двух случайных вещественных чисел в диапазоне [0, МаксимальноеЗначение];
Random2(a, b: integer) – возвращает кортеж из двух случайных целых чисел в диапазоне [a, b];
Random2(a, b: real) – возвращает кортеж из двух случайных вещественных чисел в диапазоне [a, b];
Random2( ) – возвращает кортеж из двух случайных вещественных чисел в диапазоне [0, 1];
Random3(МаксимальноеЗначение: integer) – возвращает кортеж из трех случайных целых чисел в диапазоне [0, МаксимальноеЗначение – 1];
Random3(МаксимальноеЗначение: real) – возвращает кортеж из трех случайных вещественных чисел в диапазоне [0, МаксимальноеЗначение];
Random3(a, b: integer) – возвращает кортеж из трех случайных целых чисел в диапазоне [a, b];
Random3(a, b: real) – возвращает кортеж из трех случайных вещественных чисел в диапазоне [a, b];
Random3( ) – возвращает кортеж из трех случайных вещественных чисел в диапазоне [0, 1];
Round(x: real): integer – возвращает значение x, округленное до ближайшего целого. Если значение x находится посередине между двумя целыми числами, округление осуществляется к ближайшему четному числу (так называемое ***банковское округление***), например, Round(2.5)=2, Round(3.5)=4;
Round(x: real; digits: integer): real – возвращает значение x, округленное до ближайшего вещественного числа с digits знаками после запятой;
RoundBigInteger(x: real): BigInteger – возвращает значение x, округленное до ближайшего целого типа BigInteger;
Sign(x: число): integer – возвращает значение -1, 0 или 1 в зависимости от знака числа x;
Sin(x: real) – возвращает синус угла x, заданного в радианах;
Sin(x: Complex) – возвращает синус комплексного угла x, заданного в радианах;
Sinh(x: real) – возвращает гиперболический синус угла x, заданного в радианах;
Sqr(x: число) – возвращает квадрат числа x;
Sqrt(x: real) – возвращает квадратный корень из числа x;
Sqrt(x: Complex) – возвращает квадратный корень из комплексного числа x;
Tan(x: real) – возвращает тангенс угла x, заданного в радианах;
Tanh(x: real) – возвращает гиперболический тангенс угла x, заданного в радианах;
Trunc(x: real): integer – возвращает целую часть вещественного числа x, полученную отбрасыванием дробной части;

TruncBigInteger(x: real): Biginteger – возвращает целую часть вещественного числа x, полученную отбрасыванием дробной части.

Примечание: для комплексных чисел определены и другие математические функции, например тангенс, гиперболические и обратные тригонометрические функции, но они доступны как статические (классовые). Их перечень подскажет Intellisence, если в редакторе, ввести Complex с последующей точкой. Также перечень и описание этих функций можно найти в материалах по Microsoft .NET.

В. Процедуры

Randomize(seed: integer) – инициализирует датчик псевдослучайных чисел, используя значение seed. При одном и том же seed генерируются одинаковые псевдослучайные последовательности;

Randomize – инициализирует датчик псевдослучайных чисел, используя системное время, что позволяет избегать повторения одинаковых псевдослучайных последовательностей.

Предметный указатель

$

$reference, 402

A

abstract, 476
and, 72
array of, 167

B

begin … **end**, 33
BigInteger, 39, *278*
boolean, 70
break, 89
byte, 39

C

cardinal, 39
case, 79
char, 201
class, *446*
const, 39, 103, 263
constructor, 451
continue, 90

D

DateTime, 281
decimal, 60, *280*
double, 60

E

exception, 428
exit, 91
extensionmethod, 122, 471, *494*

F

False, 70
file, 309
finalization, 402
for, 83
foreach, 132, *137*
forward, 112
function, 108

I

if … **then** … **else**, 74
implementation, 399, *400*
inherited, 466
initialization, 401
int64, 39
integer, 39
interface, 399, 499
internal, 449

L

longint, 39
longword, 39
loop, 82

N

NET-сборка, 449
new, 452
nil, 456
not, 71

O

on, 435
operator, 468
operator explicit, 470
operator implicit, 470
or, 71
overload, 112
override, 454, 469, 473

P

params, 107, *192*
pointer, 275
private, 449
procedure, 103
program, 405
property, 458
protected, 449
public, 449

R

raise, 437, *441*
read, 459
real, 60
record, 255
reintroduce, 473
repeat ... **until**, 88

S

sealed, 450
Self, 494, *495*
sequence of, 126
set of, 161
shortint, 39
single, 60
smallint, 39
static, 456
string, 213

T

True, 70
try ... **except**, 435
try ... **finally**, 440
type, 254

U

uint64, 39
unit, 399
uses, 399, *402*

V

value, 459
var, 42, 103
virtual, 472

W

while ... **do**, 86
word, 39
write, 459

X

xor, 72

Y

yield, 133

А

атрибуты, 447
- геттер (getter), 447
- сеттер (setter), 447

Б

базовый Паскаль, 28
библиотека dll, 408
- подключение, 409
- структура, 409

блок, 34

В

выражение, 44
- арифметическое, 44
- логическое, 70

Д

данные
- ввод, 54, 66, 130, 171, 208, 214, 291
- вывод, 34, 65, 217, 261, 291
- область видимости, 116

декремент, 56
динамический массив, 166
- агрегирование (свертка) Aggregate, 178
- ввод элементов, 171
- генераторы, 169
- индексы, 166
- матрица, 166
- проецирование Select, 175
- срезы, 180

создание, 167
документирующий комментарий, 410

З

запись, 255
- вывод, 261
- инициализаторы полей, 259
- инициализация, 259
- конструктор, 257
- операция присваивания, 262
- поле записи, 255

захват переменной, 119

И

идентификатор. *См.* имя
имя, 42
- глобальное, 403
- локальное, 403
- область видимости, 404
- позднее связывание, 472
- раннее связывание, 472

инициализация, 42
инкапсуляция, 452
инкремент, 56
интерфейс, 498
- IClonable, 508
- IComparable, 503
- IComparer, 505
- наследование, 502
- описание, 499
- реализация в классе, 500

исключение, 428
- пользовательское, 435
- стандартное, 435

К

класс, 120, *446*
- System.Console, 511
- абстрактный метод, 476
- автокласс, 465
- анонимный, 492
- виртуальный метод, 472
- деконструктор, 490
- конструктор, 255, *450*
- метод класса, 120, 448, *457*
- метод расширения, 122, *494*
- модификатор доступа, 449
- обработик исключений, 513
- операции **as** и **is**, 482
- описание класса, 449
- перегрузка операций, 468
- поле класса, 120, *448*
  - статическое (классовое), *456*
  - экземплярное, *456*
- производный. *См.* унаследованный
- расширение. *См.* метод расширения
- свойство класса, 448, *458*
  - автосвойство, 466
  - индексное, 463
  - расширенное, 459
- сопоставление с образцом, 488
- унаследованный, 449
- член класса, 448

ключевые слова, 34
кодовая таблица, 201
коллекция, 350
- Dictionary (словарь), 385
- HashSet (множество), 376
- SortedDictionary, 395
- SortedList, 394
- SortedSet, 390
- контейнер, 350
- обобщенная, 350

комментарии, 34
компаратор, 391
константа, 39
кортеж, 143
кортежное присваивание, 43

Л

ленивые вычисления. *См.* отложенные вычисления
литерал, 41
лямбда-выражение, 116
лямбда-процедура, 118
лямбда-функция, 118

М

массив, 166
- двухмерный, 286
- динамический. *См.* динамический массив
- многомерный, 299
- статический. *См.* статический массив

матрица, 286
- ввод элементов, 291
- вывод элементов, 291
- конструктор, 288
- описание и создание, 287

множество, 160
- конструктор, 161, *162*
- операции, 162
- создание, 161

модуль, 398
- инициализация, 401
- интерфейс, 400
- пространство имен, 403
- реализация, 400
- стандартный, 421
- структура, 399
- упрощенный синтаксис, 407
- учебный, 423
- финализация, 402

Н

наследование, 452, *466*
- принцип подстановки Лисков, 468

О

объект
- метод, 447
- поле, 447
- свойство, 447

объект первого класса, 117
операнды, 44
оператор, 34
- выбора, 79
- присваивания, 50
- составной, 34
- условный, 74
- цикла, 81

операторные скобки, 33
операция, 44
- арифметическая, 44
- логическая, 71
- отношения, 71
- присваивания, 42
- условная, 76

отложенные вычисления, 132
очередь, 358

П

парадигма программирования, 32
параметр цикла, 83
параметры подпрограмм, 101
- аргументы. *См.* фактические
- значение по умолчанию, 106
- передача по значению, 101
- передача по ссылке, 101
- фактические, 104
- формальные, 104

переменная, 42
- внутриблочная, 403
- полиморфная, 473

подпрограмма, 100
- обратный вызов, 114
- опережающее объявление, 111
- перегрузка имени, 112
- побочный эффект, 109
- процедура, 100
- упрощенный синтаксис, 103
- функция, 100, 108

полиморфизм, 452, 472
последовательность, 124
- агрегирование (свертка) Aggregate, 141
- бесконечная, 130
- ввод элементов, 130
- генераторы, 127
- детерминированная, 126
- недетерминированная, 126
- операции с последовательностями, 150
- поиск, 156
- проецирование Select, 137
- разбиение, 148
- создание, 126
- сортировка, 154
- срез Slice, 149
- технология LINQ to Objects, 137
- фильтрация Where, 146

приведение типа, 62
- автоматическое (неявное), 42, 483
- апкаст, 482
- даункаст, 482
- явное, 52, 483

приоритет операций, 46, 72
процедура. *См.* подпрограмма

Р

разыменование указателя, 276
регулярное выражение, 233
- квантификатор, 236
- метасимвол, 234
- опции, 239
- экранирование, 234

рекурсия, 111

С

символ, 200
- ввод, 208
- операции преобразования, 207

список
- LinkedList, 363
- List, 370
- линейный список, 351
- связный список, 362

ссылка, 103
стандартные модули
- для работы с графикой, 421

статический массив, 193
стек, 351
строка, 200
- арифметические операции, 219
- ввод, 214
- вывод, 217
- интерполированная, 249
- короткая (размерная), 213
- основанная на строках .NET, 213
- разбиение на слова, 227
- регулярное выражение. *См.* регулярное выражение
- составное форматирование, 247
- срез, 222
- форматирование данных, 246

Т

тип данных, 35
- вещественный, 60
- диапазонный, 265
- динамический, 102, *473*
- запись. *См.* запись
- инстанцирование, 255, *273*
- логический, 70
- обобщенный, 255, *273*
- перечислимый, 263
- порядковый, 202
- процедурный, 114
- размерный, 102
- символьный, 201
- синоним, 255
- ссылочный, 102
- статический, 102
- строковый, 212
- указатели, 275
- файловый, 306
- целый, 38
- эквивалентность и совместимость типов, 265

точечная нотация, 121

Ф

файл, 306
- бестиповый, 307
- двоичный, 307
- запись, 306
- операции с файловым указателем, 343
- операция записи в файл, 321
- операция чтения файла, 331
- открытие и закрытие, 312
- текстовый, 307
- типизированный, 307
- файловая переменная, 306
- файловый указатель, 310

функция. *См.* подпрограмма

Ц

цикл, 81
- **foreach**, 137
- loop, 82
- заголовок цикла, 81
- с постусловием, 88
- с предусловием, 86
- со счетчиком, 83
- тело цикла, 81

Э

экранирующий символ, 43