Лицензия

Авторские права на публикуемые материалы принадлежат автору книги Осипову Александру Викторовичу. Публикация данных материалов не предполагает извлечения какой-либо коммерческой выгоды.

Публикуемые материалы защищены действующим законодательством об авторском праве. Все предусмотренные этим законодательством права на опубликованные материалы принадлежат их автору.

Официальным источником для распространения материалов является Интернет-сайт //pascalabc.net, ссылка на который при цитировании обязательна. Разрешается свободно копировать и распространять исключительно на безвозмездной основе опубликованные материалы при условии сохранения их в неизменном виде и с указанием авторства. Передача материалов третьим лицам разрешается при условии сохранения в них страницы с настоящей лицензией. Исключение делается для учебных заведений: при подготовке раздаточного материала допускается страницу с лицензией не включать. Любые другие способы распространения опубликованных материалов при отсутствии письменного разрешения автора запрещены.

Запрещается любым организациям осуществлять любого рода лицензирование опубликованного материала и осуществлять какую бы то ни было иную связанную с авторскими правами деятельность без письменного разрешения автора.

ЧАСТЬ 7

7 Типы данных

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**). Сразу отметим, что 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*** (глава 7.6), для которого конкретный тип определяется позднее. 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 (глава 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 (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**.

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, т.е. динамический массив элементов типа Ученик. С помощью SetLength массиву выделяется память для размещения 26 элементов – именно столько учеников будет в классе. Далее нужно занести исходные данные, для чего организуется цикл с перебором по всем ученикам. Построение обращения к конкретному полю должно быть понятно из приведенного фрагмента программы.

Для вывода значений полей записи приходится строить обращение к каждому полю. Необходимость записывать для каждого поля длинный перечень 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. Дело в том, что тип **record** на самом деле является классом, поэтому соответствующая терминология распространяется и на него.

Один из методов мы выделим особо. Это функция с фиксированным именем Create, называемая ***конструктором***, причем 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 – 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 свой конструктор (и даже несколько конструкторов с разными параметрами), будет вызван тот конструктор, параметры которого будут заданы 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.

В начале главы 7.1 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.

```
// p070102
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.

```
// p070103
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

Об инициализации записи уже частично писалось в разделе 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;
```

§1. 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.

§2. 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.

§3. 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);
```

§4. 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.

```
// p070105
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.

```
// p070106
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. Используя пример из предыдущей главы 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.

```
// p070108
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, а каждый его элемент занимает в памяти 4 байта. Это дает возможность использовать значения перечислимого типа 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);
```

Теперь 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.

Перечислимый тип можно не описывать в разделе типов, а 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

Представляет 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 нормально отработает 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». Это как понимать - 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 (глава 7.7) с совпадающими базовыми типами;
- 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;

- Т1 – бестиповый указатель **pointer** , Т2 – типизированный указатель;
- Т1 – указатель или процедурная переменная, Т2 = **nil**;
- Т1 – процедурная переменная, Т2 – имя процедуры или функции со структурно эквивалентными параметрами
- Т1 и Т2 имеют классовый тип (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;
- Т1 имеет тип интерфейса (часть 11), Т2 – тип класса, реализующего этот интерфейс.

Если тип Т2 совместим по присваиванию с типом Т1, говорят, что тип Т2 ***неявно приводится*** к типу Т1.

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].

```
// p0705a
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, современных 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.

```
// p0705b
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 … a_k, a_{k-1} …

```
function Perm<T>(a: array of T): array of T; // p0706
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, …>(…)

7.6.1 Дополнительные операции с обобщенным типом

Как уже отмечалось, 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.

```
// p070601
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 элементов 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 PascalABC.NET этот тип зарезервирован для другого понятия, поэтому указатели здесь 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 тип данных заранее не определен.

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. К бестиповому указателю операция разыменования неприменима. 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.

7.7.1 Пример работы с указателями

Реализуем с помощью указателей так называемый односвязный список из трех элементов. Односвязным он называется потому, что между элементами списка устанавливается одна связь, а именно, со следующим элементом.

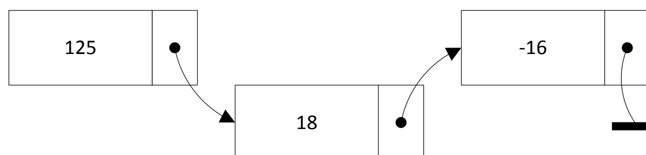

Рис. хх. Односвязный список.

```
type // p070701
  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 она вынесена в отдельную главу.

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('123456789012345678901234567890');
```

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 // p070802
  var a := BigInteger.MinusOne; // это -1
  var b := integer(a);
  a := BigInteger.Parse('1234567890123456789012345');
  Print(b, real(a)) // -1 1.23456789012346E+24
  var s := a.ToString; // к строке приводим обычным методом
  Println(s) // 1234567890123456789012345
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 // p0709
  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 // p0710a
  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 я сейчас себе представил 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 // p0710b
  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**.