ГЛАВА 6

Конструирование новых типов данных

В конце 60-х годов с появлением транзисторов, а затем интегральных схем, стоимость компьютеров резко снизилась, а их производительность росла почти экспоненциально. Появилась возможность решать все более сложные задачи, но это требовало умения обрабатывать самые разнообразные типы данных. Такие языки, как ALGOL-68 и затем Pascal, стали поддерживать абстракцию данных. Программисты смогли описывать свои собственные типы данных. Это стало еще одним шагом к предметной области и от привязки к конкретной машине.

Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++.

Две предыдущие главы познакомили нас с базовой техникой создания программ с использованием языка программирования высокого уровня Object Pascal. Того, что мы успели изучить к настоящему моменту, было бы вполне достаточно, чтобы чувствовать себя в программировании более-менее уверенно, если бы не одно «но».

Рассмотрим простую задачу. Пусть нам требуется найти средний балл студента за сессию, количество экзаменов в которой равно трем. Математически решение очевидно. Сумма трех чисел делится на три – получается результат. Составить программу тоже никаких проблем. Объявляем три переменных `Mark1`, `Mark2`, `Mark3`, переменную `AvMark`, дальше все понятно.

Теперь усложним ситуацию. Нам нужен средний балл студента по всем сессиям (понятно, что это актуально для студентов, проучившихся хотя бы год, но таких как раз больше). Как теперь представить в программе все оценки? Не создавать же по отдельной переменной под каждую!

Следующее усложнение: нужен средний балл для каждого студента в группе (на курсе, по всему факультету). Помимо того, что оценок становится просто огромное количество, нужно еще знать, какой средний

балл к кому относится, то есть неплохо бы внести в программу информацию о студенте (номер группы, ФИО). И было бы совсем хорошо связать все, что относится к каждому студенту, в единую конструкцию, то есть создать в программе объект «студент».

Суммируем сказанное. В подавляющем большинстве практических задач существует потребность в обработке большого числа однотипных данных, при этом сами данные часто имеют сложное и неоднородное «внутреннее устройство». Следовательно, языки программирования, претендующие на широкое использование сообществом программистов, обязаны иметь средства для представления таких данных и удобной работы с ними. Знакомству с этими средствами и посвящена настоящая глава.

1. Абстрактные типы данных

Начиная решение любой прикладной задачи, мы прежде всего выясняем, с какими объектами имеем в ней дело, то есть определяем *предметную область* задачи. Каждый объект предметной области обладает некоторой характеризующей его совокупностью параметров. При этом в зависимости от конечной цели, которую надо достигнуть в процессе решения, из всей совокупности параметров мы выбираем интересующие и отбрасываем остальные. В рассмотренной выше задаче нахождения среднего балла студента за период обучения нас не интересуют ни рост каждого из них, ни цвет глаз, ни вес, хотя понятно, что в других задачах именно эти данные могут иметь значение. Таким образом, как мы уже отмечали в первой главе, в процессе решения мы работаем не с самими объектами, а с их моделями или, говоря другими словами, *абстракциями*. Абстрагирование является одним из фундаментальных методов научного исследования. Что касается программирования, то в нем абстракциями пронизано все.
В конечном счете, любая программа, как отображение кусочка реального мира на последовательность нулей и единиц, есть одна из высших форм абстракции, придуманных человечеством.

Вместе с тем, как мы не раз уже отмечали, несмотря на всю мощь человеческого разума, мыслить исключительно в двоичной логике нам довольно тяжело. Даже на аппаратном уровне в компьютерах поддерживаются такие общепринятые математические понятия, как целое и вещественное числа в десятичной системе счисления. А языки программирования высокого уровня предоставляют программисту соответствующие *встроенные* типы данных. Таким образом,

обеспечивается уровень абстракции, достаточный для создания программ, принадлежащих предметной области «математика», то есть расчетных.

Однако при всем нашем уважении к математике подавляющее большинство программ в современном компьютерном мире принадлежат другим предметным областям, задания на их разработку создаются в терминах более сложных, чем числа, а значит, средства программирования, и в первую очередь языки, должны предоставлять возможность создания и использования при разработке программ соответствующих абстракций.

Языки программирования высокого уровня паре «объект – класс объектов» как элементов предметной области ставят в соответствие пару «переменная – абстрактный тип данных». При этом в состав любого языка входят некоторые простые типы данных, реализация которых выполняется разработчиками компилятора. Например, многие языки программирования высокого уровня предлагают тип данных «строка», который, конечно же, не поддерживается процессором. В силу важности этого понятия еще раз напомним: термином *абстрактный* в данном случае подчеркивается тот факт, что тип данных не реализован аппаратно, а сконструирован средствами языка программирования. В идеале средства создания новых типов данных должны быть реализованы в языке так, чтобы использование встроенных в язык типов ничем не отличалось от использования типов, созданных программистом.

Как мы обсуждали в главе 4, определяя тип данных, мы обязаны указать:

1. Множество значений.
2. Набор операций, применимых к значениям данного типа.
3. Способ представления значений в оперативной памяти и их интерпретации.
4. Размер оперативной памяти, необходимый для хранения значений.

В языке Pascal при создании нового типа данных способ представления и интерпретации, а также размер определяются компилятором автоматически в соответствии с конструкцией типа данных. Множество значений либо задается программистом явно, либо определяется на основе множеств значений типов данных, из которых построен новый тип. Что касается операций, некоторые элементарные придаются новому типу автоматически («наследуются»), остальные программист реализует в виде подпрограмм[1].

[1] Подробно о подпрограммах мы поговорим в главе 7.

2. Определение типов прямым заданием множества значений

Построение любого типа данных начинается с формирования множества значений. Поскольку размер оперативной памяти для хранения значений типа всегда конечен, то и задавать мы можем лишь конечные множества. Сказанное справедливо и для всеми любимых вещественных чисел. Например, на хранение мантиссы в типе `Real` отведено 52 бита, что с очевидностью означает: максимальное количество разных чисел одного и того же порядка, скажем в диапазоне от 0 до 1, равно 2^{52}, что, конечно, весьма не мало, но все же не бесконечно много. Из сказанного напрямую вытекает следующий вывод – наиболее естественным механизмом создания абстрактного типа представляется простое перечисление элементов множества.

2.1. Перечислимый тип данных

Предположим, мы решаем задачу автоматизации управления уличным движением, для чего разрабатываем соответствующую программную систему. Очевидно, что одним из важных объектов нашей предметной области будет *светофор*. Попытаемся построить его модель.

После некоторых раздумий приходим к выводу, что единственным имеющим значение в контексте данной задачи параметром объекта «светофор» является параметр *сигнал*. Возможные состояния: «красный», «желтый», «зеленый». Для представления данного параметра в программе необходим тип данных. Простейшее решение – использовать любой из встроенных целочисленных типов, представив каждый из цветов некоторым *кодом*. Например, «0» – красный, «1» – желтый и «2» – зеленый.

Надеемся, читателю очевидно, что это решение имеет существенные недостатки.

Первый – при написании текста программы разработчик должен постоянно держать в голове «таблицу перекодировки», то есть помнить, как он обозначил цвета. Это практически гарантированный источник ошибок, не говоря уже о существенных трудностях, связанных, скажем, с внезапно возникшей потребностью поменять значения с 0, 1, 2 на -1, 0, 1. Избавиться от этой проблемы достаточно просто, введя константы вида `clRed = 0`.

Со вторым недостатком сложнее. Переменной, обозначающей состояние светофора (в соответствии с множеством значений выбранного целочисленного типа), можно будет присвоить значение, отличное от 0, 1, 2. К чему это приведет во время работы программы – неизвестно, скорее всего, ни к чему хорошему. Найти такую ошибку в тексте программы будет весьма не просто, а компилятор нам в этом помочь не сможет, поскольку с его точки зрения все будет вполне корректно. Что же делать?

Кардинальное решение рассмотренных проблем состоит в объявлении нового типа данных *светофор* как набора из трех упорядоченных значений *красный*, *желтый*, *зеленый*.

В языке Pascal объявление типа производится в специальном разделе, начинающемся с зарезервированного слова **type**. В общем виде объявление выглядит следующим образом:

```
type
  <имя типа> = <определение типа>;
```

Имя типа – это произвольный идентификатор. Определение типа – синтаксическая конструкция, индивидуальная для различных способов объявления типов данных. В нашем случае речь идет о создании *перечислимого типа* данных, который в соответствии с названием определяется перечислением *имен констант* типа в круглых скобках через запятую, например:

```
type
  TrafficLight = (clRed, clYellow, clGreen);
```

Имя константы – произвольный идентификатор, в силу чего на имена констант перечислимых типов распространяется общее правило неповторяемости – один и тот же идентификатор нельзя использовать более чем в одном перечислимом типе. Таким образом, следующий код вызовет ошибку компиляции:

```
type
  TrafficLight = (clRed, clYellow, clGreen);
  Light = (clRed, clYellow, clGreen, clBlack);
```

Подчеркнем, что в круглых скобках при определении типа указываются лишь имена констант, а не способы представления значений. Поясним это важное различие. Если мы записываем вещественную константу –3.14, то мы понимаем, что за этим общепринятым символическим обозначением (фактически, именем константы) стоит аппаратно реализованная форма представления вещественных чисел с набором операций над ними. Разумеется, в аппаратуре, так же как и в стандартном программном обеспечении, нельзя заранее предусмотреть все возможные типы

конструируемых пользователем значений и придумать заранее, например, способ представления значения «зеленый сигнал светофора». Поэтому «внутри машины» значения перечислимого типа кодируются одинаково вне зависимости от того содержательного смысла, который придает им программист. Код значения перечислимого типа – это его порядковый номер в списке определения типа. Нумерация значений начинается с нуля. Заметим, что вывести имя константы на печать с помощью оператора `Write` не удастся, более того, согласно спецификации этого оператора аргументы перечислимого типа им не поддерживаются, то есть не только имя константы, но и само ее значение вывести на экран, используя оператор печати, невозможно. К вопросу о выводе информации о значении переменной перечислимого типа мы вернемся позднее.

Вспоминая еще раз составляющие понятия «тип данных», мы видим, что пока нами задано лишь множество значений. Вторая важная часть – набор операций – существенно зависит от контекста задачи. Как мы уже отмечали, эти операции обычно реализуются в виде подпрограмм. Однако некоторые простейшие операции система программирования Delphi автоматически связывает с вновь создаваемым типом данных.

Для перечислимого типа данных имеется возможность выполнять операцию присваивания значений. Можно сравнивать значения, учитывая, что они упорядочены в соответствии с их записью при объявлении типа. К значениям перечислимого типа можно применять функции `Ord`, `Succ` и `Pred`. Первая из них возвращает код значения (напоминаем, что первое значение имеет код ноль), вторая выдает следующее после аргумента значение в списке, а последняя – предыдущее. Следует помнить, что функция `Succ` не определена на последнем значении, а `Pred` – на первом.

Язык Pascal предоставляет некоторые средства контроля за применением операций над значениями сконструированного типа. Прежде всего, на этапе компиляции будут отмечены все ошибки, связанные с присвоением переменным значений констант, не входящих в описание типа. Кроме того, запрещается присваивать переменным одного перечислимого типа данных значения переменных другого перечислимого типа (что является очевидным следствием предыдущего ограничения).

Ввод значений переменных перечислимого типа наиболее разумно выполнять, предоставляя пользователю возможность выбора из имеющегося в типе списка вариантов. Впрочем, программирование такого способа сопряжено с некоторыми усилиями, предоставляем читателю возможность попытаться сделать это самостоятельно. Второй достаточно неплохой с пользовательской точки зрения способ состоит в использовании строковых констант. В нашем случае это будут константы, обозначающие цвета. Рассмотрим пример.

```
{ ================================================= }
{ Пример 6.1 }
{ Ввод значений перечислимого типа }
Program Input;

{$APPTYPE CONSOLE}
type
   TrafficLight = (clRed, clYellow, clGreen);
var
   T: TrafficLight;
   S: string[6];
begin
   { Ввод в переменную T }
   Readln(S);
   if S = 'Red' then
      T := clRed
   else
      if S = 'Yellow' then
         T := clYellow
      else
         if S = 'Green' then
            T := clGreen
         else begin
            Writeln('Неопознанное значение');
            Halt;
         end;
   { Конец ввода }
   ...
end.
{ ================================================= }
```

Заметим, что совместно с переменными перечислимого типа наиболее выгодно использование оператора множественного выбора **case**. Однако в данном случае выбор осуществляется по переменной S, имеющей строковый тип, поэтому приходится обходиться вложенным условным оператором.

Еще одно замечание связано с досрочным выходом из программы при вводе неверного значения цвета. Как мы уже отмечали в предыдущей главе, на практике такой вариант является крайне неудачным и должен быть заменен циклом **repeat**, в котором ввод значения повторяется до получения корректного результата.

2.2. Тип «диапазон»

Вторая возможность конструирования нового скалярного типа данных заключается в выделении подмножества значений ранее определенного типа.

Допустим, некоторая программа включает обработку дат, в частности дней месяца. День месяца – это число в диапазоне от 1 до 31. Использование в качестве информационной модели для этого понятия одного из целых типов (даже самого «маломощного» из них – `Byte`) приводит к проблемам, аналогичным тем, что были рассмотрены в предыдущем разделе, то есть к отсутствию средств защиты от использования некорректных констант. Следовательно, желательно иметь средства, позволяющие явно ограничить круг используемых значений, выделив его из некоторого ранее определенного типа. Такие средства обеспечиваются возможностью конструирования типа *диапазон*.

Формальная запись типа *диапазон* выглядит так:

```
type
  <имя типа> = <константа 1>..<константа 2>;
```

Диапазоны могут определяться только на основе порядковых типов данных: целого, символьного и перечислимого. Константы определяют нижнюю и верхнюю границы диапазона. Соответственно `константа 1` должна быть меньше, чем `константа 2`.

Например,

```
type
  Day = 1..31;
  CapLet = 'A'..'Z';
  Color = (clBlack, clLightGray, clWhite, clRed, clGreen,
    clBlue);
  BWColor = clBlack..clWhite;
```

При компиляции ведется контроль за правильностью использования констант. Если, допустим, имеется объявление переменной

```
var
   D: Day;
```

и в программе встретится оператор присваивания

```
D := 35;
```

то компилятор выдаст ошибку, говорящую о том, что константа вне диапазона допустимых значений.

Более того, в процессе исполнения программы также можно обеспечить проверку присваиваемых значений. Это делается с помощью уже известной нам директивы компилятора `{$R}`.

Допустим, в программе имеются объявления

```
var
```

```
C: Color;
BW: BWColor;
```

и операторы присваивания

```
{$R+}
C := clRed;
BW := C;
```

На этапе компиляции ошибочное присвоение константы «красное» (clRed) «черно-белой» переменной `BW` не может быть обнаружено (типы `Color` и `BWColor` совместимы по присваиванию, о чем мы будем говорить ниже). Во время же исполнения при наличии директивы `{$R+}` будет выдано сообщение об ошибке и программа будет аварийно завершена.

При определении типа «диапазон» с ним связывается весь набор операций типа-родителя. Так, над диапазонами целых чисел определены все арифметические и другие операции. Проблема может возникнуть с результатом операции, не входящим в диапазон. Но это уже разговор, связанный с совместимостью типов, который, как только что отмечалось, мы поведем ниже.

3. Определение типов путем комбинирования ранее определенных типов данных

Другой подход к построению новых типов связан с указанием способа конструирования входящих в них значений и основан на группировке по некоторым правилам значений ранее определенных типов. Язык Pascal предлагает три способа формирования сложных значений: объединение однотипных значений (регулярный тип или массив), объединение разнотипных значений (записи) и определение множеств.

3.1. Регулярный тип. Массивы

Вернемся к примеру, с которого мы начинали эту главу. Нам нужна программа учета успеваемости студентов факультета. Минимально необходимой информацией о каждом студенте является ФИО и оценки по экзаменам за каждую сданную сессию. Очевидно, оценки студента являются целыми значениями, то есть принадлежат одному и тому же типу. Также очевидно, что мы не можем завести по отдельной переменной под каждую из них. Кроме того, информация, описывающая каждого студента,

также является типовой в том смысле, что структура этой информации одинакова для любого студента.

Итак, мы имеем необходимость использования в программе набора однотипных данных. Познакомимся с тем, как решается подобная задача.

3.1.1. Объявление типа «массив». Большие объемы однотипных данных представляются средствами языка программирования в виде так называемых регулярных типов или *массивов*. Для задания массива требуется указать его имя (идентификатор), размерность (вектор, матрица, трехмерный массив и т.д.), число элементов по каждому измерению и способ индексации элементов в каждом измерении (тип индекса).

Конструкция объявления типа *массив* имеет следующий вид:

```
<имя типа> = array[<тип(ы) индекса(ов)>] of <тип элемента>;
```

Здесь используются два зарезервированных слова: **array** (массив) и **of**. Конструкция задает размерность массива и способ индексации.

Обычно индексы задаются списком типов диапазонов. Вообще говоря, в качестве типа индексов может выступать любой порядковый тип. При этом совершенно не обязательно использовать анонимное объявление. В квадратных скобках можно записывать имя типа. Приведем примеры объявлений массивов:

```
type
  Sequence = array[1..10] of Real;
  MatrixOfReal = array[1..50, 1..100] of Real;
  Index = -20..20;
  MatrixOfInt = array[Index, -10..100] of Integer;
  CubeOfChar = array[0..50, 'a'..'z', 50..100] of Char;
  BooleanVector = array[Byte] of Boolean;
  Month = (Jan, Feb, Mac, Apr, May, Jun, Jul, Aug, Sep,
    Oct, Nov, Dec);
  NDaysInMonth = array[Month] of 1..31;
```

В примерах использована, на первый взгляд, довольно экзотическая индексация отрицательными числами, символами, значениями перечислимого типа. Однако в конкретных приложениях использование индексов подобного рода может быть вполне уместно, например, если речь идет о массиве строк-пунктов некоторой инструкции или договора, которые часто индексируются латинскими буквами, или, как в последнем примере, о количестве дней в каждом месяце.

Рассмотрим важный вопрос, связанный со способом выделения оперативной памяти под массив. Как было отмечено ранее, число элементов массива задается как число элементов порядкового типа, выбранного в качестве типа индекса. Это означает, что размер памяти,

занимаемой массивом, фиксируется во время компиляции и не может быть изменен при исполнении программы – так называемое *статическое распределение памяти*.

Статическое распределение памяти под массивы приводит к некоторым сложностям. В частности, невозможно увеличить число элементов, если во время работы программы возникнет такая необходимость. Эту проблему решают массивы динамические и вообще *динамическое распределение памяти*, то есть выделение во время работы программы, на которое зато требуется дополнительное время. Работу с динамическими массивами мы рассмотрим позднее в главе 9.

Как же справиться с изменением размерностей массивов при статическом распределении? Здесь существует простой подход, используемый во всех языках, где также реализовано статическое распределение памяти (например, язык Fortran). Из анализа задачи делается вывод о максимально возможном размере массива, и этот размер указывается в объявлении. Например, если разрабатывается программа бухгалтерского учета на предприятии, которая должна суммировать заработную плату по цехам, то следует в качестве размера массива, хранящего зарплаты работников цеха, взять максимально возможное количество работников в одном цехе. Но что же делать, если со временем в эксплуатацию введут новый цех, где число работников будет больше прежнего максимального? В этой ситуации, конечно, без перекомпиляции программы обойтись не удастся. Другое дело, что можно порекомендовать прием, сводящий переделку программы к минимуму (и мы настоятельно советуем читателю использовать его повсеместно). Идея состоит в том, что размеру массива присваивают символическое имя в разделе объявлений констант, которое впоследствии и используют в тексте программы. Переделка такой программы – всего лишь смена соответствующей строки. Пример подобного объявления массива:

```
const
  NumberOfWorkers = 150;
type
  Salary = array[1..NumberOfWorkers] of Real;
```

Еще раз напомним, что `NumberOfWorkers` – это константа, которая в принципе не может быть изменена в процессе исполнения программы.

Второй аспект этой проблемы – ограничение на максимальный размер массива. Например, формально правильное объявление типа

```
type
  LargeVector = array[Integer] of Real;
```

вызовет во время компиляции сообщение об ошибке, гласящее, что структура слишком велика. В чем здесь дело? Ведь `Integer` – порядковый

тип, разрешенный для употребления в качестве типа индексов! Чтобы понять причину некорректности подобного объявления, давайте посчитаем, сколько памяти понадобится под такой массив. Тип `Integer` в языке Object Pascal, как мы помним, имеет размер 4 байта, то есть способен хранить 2^{32} различных значений. Умножаем это число на 8 байт – размер типа `Real` – получаем 32 Гб. Подобный объем адресного пространства способны предоставить лишь 64-разрядные операционные системы, работающие на 64-разрядных же процессорах, в то время как подавляющее большинство потребительских операционных систем являются в настоящий момент 32-разрядными и объем памяти, с которым может работать в них программа, не превышает 4 Гб.

3.1.2. Объявление переменных и констант типа «массив». Переменные типа «массив» объявляются обычным способом в секции **var**, например:

```
var
   Shop1, Shop2: Salary;
   X: Sequence;
   CubeOf: CubeOfChar;
   M: MatrixOfInt;
```

Синтаксис языка Pascal допускает так называемые анонимные объявления типов, когда вновь определяемому типу данных имя не присваивается, а его описание производится прямо при объявлении переменной, например:

```
var
   Page: array[1..30] of String[60];
```

Здесь переменная Page сразу определяется как страница из тридцати строк по 60 символов в каждой. При этом общее имя для нового типа не вводится. С точки зрения хорошего стиля программирования, а также целого ряда практических соображений, связанных с проблемой совместимости типов, передачей таких переменных в качестве параметров в подпрограммы, и ряда других, такой способ объявления переменных следует признать неудачным.

Имеется возможность задавать и константы типа «массив».

Общий вид описания константы-массива представлен ниже:

```
const
   <имя> : <описание типа массив | имя типа массив> =
      (<список значений>);
```

Как видим, такие константы автоматически получаются типизированными. Более того, для массивов это единственный способ определения констант.

Например,

```
const
  StandardSeq: Sequence = (1, 2, 3, 4, 5, 3 + 3, 7, 16
    div 2, 9, 10);
  abcd: array[1..2, 1..2] of Char = (('a', 'b'),
    ('c', 'd'));
```

Как видно из примера, список значений задается через запятую. Сами значения могут задаваться выражениями из констант. При задании массива-константы должны быть определены все его элементы в соответствии с определением типа, то есть в первом примере должно быть задано ровно 10 значений, а во втором 4. Второй пример показывает задание матрицы символов:

```
a b
c d
```

Заметим, что матрица задается по строкам, заключаемым в круглые скобки.

В точности так же по синтаксису выполняется и инициализация переменных типа «массив»:

```
var
  efgh: array[1..2, 1..2] of Char = (('e', 'f'), ('g', 'h'));
```

Отличие константы `abcd` от переменной `efgh` в том, что ни один из элементов `abcd` не может быть изменен, компилятор «строго» за этим следит, в то время как элементы `efgh` можно с легкостью менять в теле программы. Чуть ниже мы продемонстрируем этот факт на примере работы с датами.

3.1.3. Стандартные операции. В языке Pascal стандартные операции над массивами составляют скромный набор из двух возможностей. Во-первых, мы можем присвоить значение переменной типа «массив» другой переменной того же типа. Например, можно записать операторы присваивания

```
X := StandardSeq;
Shop1 := Shop2;
```

осуществляющие групповую пересылку значений элементов.

Другая возможность – доступ к элементу массива. Индекс элемента записывается в квадратных скобках после имени массива. Индекс

представляет собой в общем случае выражение указанного при объявлении массива типа.

```
X[2]
M[20, 30]
CubeOf[2, 'y', -50]
StandardSeq[Trunc(Exp(z - 8.7) - 6)]
```

В выражениях элемент массива используется в точности так же, как и обычная переменная.

Важная проблема при работе с массивами – обеспечение корректности значения индекса. Что произойдет, если заданный индекс выйдет за указанные при объявлении массива границы? Другими словами, эту ситуацию можно охарактеризовать как присвоение индексу-переменной значения другого типа данных. С содержательной точки зрения это, конечно, ошибка, которая может привести к неправильной работе программы. Язык Pascal имеет развитые средства контроля, позволяющие «выловить» выход индекса за границу как на этапе компиляции, так и на этапе исполнения.

Если индекс задан в виде константы, то уже при трансляции программы компилятор в состоянии проверить его корректность. Допустим, в программе встретился оператор присваивания

```
Z := X[100];
```

где `Z` некоторая вещественная переменная. Поскольку `X` принадлежит типу `Sequence` с границами 1..10, компилятор выдаст сообщение об ошибке. Более того, ошибка будет обнаружена, даже если индекс задан выражением из констант, так как оптимизирующий компилятор вычисляет все такие выражения на этапе трансляции.

Ситуация становится более сложной, если индекс вычисляется только во время исполнения. Если мы хотим обеспечить контроль индекса, то перед каждым его употреблением следует вставить команды сравнения его с границами. Разумеется, это увеличивает, во-первых, размер программы и, во-вторых, время исполнения. Чтобы избавить программиста от рутинной работы по вставке операторов проверки индексов, в язык Pascal включена директива `{$R}`, которую мы уже использовали ранее. Напомним, что по умолчанию она отключена. При ее включении, то есть записи в виде `{$R+}`, производится контроль в том числе и индексов элементов массивов. Допустим, имеется фрагмент программы:

```
i := 100;
Z := X[i];
```

При компиляции эта ошибка обнаружена не будет. Более того, при выключенной вышеупомянутой директиве ошибка не будет отмечена и во

время исполнения, а в переменную Z будет записано некоторое неизвестное заранее значение, содержавшееся в памяти вне массива X. Если же мы воспользуемся директивой, то во время исполнения будет выдано сообщение об ошибке и программа будет аварийно завершена.

Еще раз подчеркнем, что в силу больших накладных расходов этой директивой следует пользоваться только на этапе отладки программы.

3.1.4. Создание массивов сложной структуры. В предыдущих примерах в качестве типа элемента массива мы использовали встроенные типы. Вместе с тем в общем случае мы можем пользоваться любым типом, определенным в программе, в том числе и массивом. Такая возможность позволяет структурировать значения конструируемого типа и обеспечивать доступ к структурным частям значений.

Допустим, требуется определить некоторый матричный тип данных. Исходя из контекста использования матриц в конкретной задаче можно выделить несколько подходов к интерпретации ее значения. Например, ее можно рассматривать как таблицу чисел, а можно как последовательность (вектор) строк или столбцов. Каждый из этих подходов может быть отражен в определении типа:

```
const
  NRows = 10;      { число строк    }
  NColumns = 20;   { число столбцов }

type
  RowInd = 1..NRows;        { номера (индексы) строк    }
  ColInd = 1..NColumns;     { номера (индексы) столбцов }

  Matrix = array[RowInd, ColInd] of Real;    { таблица }

  Rows = array[ColInd] of Real; {строка}
  VecOfRows = array[RowInd] of Rows;    { вектор строк }

  Columns = array[RowInd] of Real; { столбец }
  VecOfColumns = array[ColInd] of Columns;
    { вектор столбцов }
```

При обращении к структурному элементу массива могут быть использованы соответствующие «структурные» способы записи индексов. Допустим, сделаны объявления переменных:

```
var
  M: Matrix;
  R: Rows;
  C: Columns;
  VR: VecOfRows;
  VC: VecOfColumns;
```

```
   E: Real;
```

Ниже приведены различные способы записи индексированных переменных:

```
E := M[2,4];
R[3] := 3.14;
C[5] := 2.7;
VR[7][2] := R[3];
VR[9][1] := M[2,4];
VR[2,5] := E;
VC[8,5] := 4.5;
VC[4] [6] := 5.6;
```

В качестве примера рассмотренных возможностей по работе с массивами приведем обещанный ранее фрагмент программы, обеспечивающий вывод значений перечислимого типа:

```
{ ================================================= }
{ Пример 6.2 }
Program Input1;

{$APPTYPE CONSOLE}

{ Вывод значений перечислимого типа }
type
   TrafficLight = (clRed, clYellow, clGreen);
const
   Colors: array[TrafficLight] of String[8] =
     ('clRed', 'clYellow', 'clGreen');
var
   T: TrafficLight;
begin
   ...
   T := clYellow;
   Writeln(Colors[T]);
   ...
end.
{ ================================================= }
```

В языке Pascal нет встроенных возможностей типа матричных операций, которые имеются в некоторых версиях языка Basic, нет даже возможности указания имени массива при вводе и выводе. Все содержательные операции пользователь должен программировать самостоятельно.

Познакомимся с элементарными приемами ввода и вывода массивов, которые основаны на применении операторов цикла. Стандартный способ ввода и вывода одномерного массива с известным количеством элементов

представлен в нижеследующем примере подсчета итоговой суммы заработной платы некоторого подразделения.

```
{ ================================================= }
{ Пример 6.3 }
{ Ввод и вывод массива }
Program SumOfSalary;

{$APPTYPE CONSOLE}

const
   NumOfMen = 30;
type
   ArrayOfWords = array[1..NumOfMen] of Word;
var
   N, S, i: Word;
   Salary: ArrayOfWords;

begin
   Writeln('Количество сотрудников?');
   Readln(N);
   if (N = 0) or (N > NumOfMen) then
   begin
      Writeln('Ошибка 1: Недопустимое количество сотрудников');
      Halt;
   end;
   Writeln('Вводите значения зарплат');
   S := 0;
   { Цикл ввода и суммирования }
   for i := 1 to N do
   begin
      Read(Salary[i]);
      S := S + Salary[i];
   end;
   Readln; { Читаем символ возврата каретки (Enter) после ввода }
   Writeln('Зарплаты');
   { Цикл вывода }

   for i := 1 to N do
      Write(Salary[i],'');
      { При выводе разделяем числа пробелами }
   Writeln;
   Writeln('Сумма ', S);
   Readln; { Задержка завершения программы }
end.
{ ================================================= }
```

Сделаем несколько замечаний по примеру.

Прежде всего, еще раз обращаем внимание на способ объявления массива `ArrayOfWords` с использованием именованной константы `NumOfMen`, указывающей максимально возможное количество работников, и наличие переменной `N`, используемой для задания текущего их числа. Подобная техника работы с массивами является, на наш взгляд, наиболее правильной.

Собственно ввод и вывод массива осуществляется с помощью оператора цикла с известным числом повторений **`for`**. Подавляющее большинство операций обработки массивов выполняется именно с помощью этого цикла, поскольку почти всегда можно заранее вычислить необходимое количество итераций.

Далее отметим, что элементы вводимого массива набираются на клавиатуре (и отражаются на экране), разделенные пробелом. После набора последнего элемента нажимается клавиша Enter, код которой должен быть прочитан оператором `Readln`.

В завершение разбора примера предлагаем читателю ответить на вопрос: сколько действий по контролю ввода мы опустили в приведенном выше коде?

Подумали? Давайте считать вместе. Количество сотрудников есть целое число, при вводе которого человек может ошибиться и нажать алфавитный символ вместо цифры. Это первая пропущенная проверка. Зарплата каждого сотрудника также является числом, с той же возможной проблемой при вводе, – это вторая проверка. Кроме того, зарплата является, безусловно, числом положительным (попробуйте ради интереса ввести отрицательное число в элемент массива `Salary` и посмотреть, что на самом деле получится) – это третья проверка. Наконец, при неудачном вводе, конечно же, не следует прерывать работу программы, нужно модифицировать код, добавив «обертки» в виде циклов **`repeat`**. Все эти необходимые в реальной программе действия мы сознательно опустили, поскольку их наличие серьезно увеличит объем кода. Однако еще раз повторяем: то, что является оправданием в учебнике, не может служить таковым в практической деятельности! Как это ни странно звучит, если вы не будете контролировать действия пользователя, этот самый пользователь перестанет работать с вашей программой и выберет другую.

3.1.5. Поиск и сортировка. В завершение раздела обсудим две самые распространенные задачи обработки массивов.

Любая программа, работающая с большими объемами информации, прежде всего должна решать задачу построения информационной модели

предметной области, то есть представления данных. Массивы в этом смысле являются классическим и достаточно удобным средством *хранения* большого количества однотипных данных. После того как с представлением разобрались, возникает следующая по важности проблема – получить доступ к сохраненной информации, то есть уметь находить нужные данные.

Представим себе, что мы разрабатываем программную систему для магазина, торгующего видеодисками. С каким вопросом наиболее часто будут сталкиваться продавцы такого магазина? Конечно же, он будет звучать примерно так: а есть у вас «пи-пи-пи (здесь было название фильма)»? Что должен делать продавец в ответ? Мы, как программисты, рассчитываем, что он произнесет: «Секундочку», запустит нашу программу (а может быть, она у него будет запущена постоянно), введет название фильма и узнает, есть такой или нет, и если есть, то где именно он располагается. Попытаемся решить эту задачу.

Итак, формально мы имеем массив названий фильмов, и фильм, который хочет найти покупатель. Пишем поиск.

```
{ ================================================= }
{ Пример 6.4 }
{ Поиск фильма в массиве }
Program FindFilm;

{$APPTYPE CONSOLE}

const
   MaxNumOfFilms = 10000;
type
   ArrayOfFilms = array[1..MaxNumOfFilms] of String[100];
var
   N, i, FPos: Word;
   Films: ArrayOfFilms;
   Film: String[100];
   IsFilmExist: Boolean;
begin
   ...
   Writeln('Название фильма?');
   Readln(Film);
   { Цикл поиска }
   IsFilmExist := False;
   for i := 1 to N do      { N – общее число фильмов }
      if Films[i] = Film then
      begin
         IsFilmExist := True;
         FPos := i;         { Запоминаем номер           }
         break;             { Если нашли, закачиваем цикл }
```

```
      end;
   if IsFilmExist then
      Writeln('Фильм есть. Номер: ', FPos)
   else
      Writeln('Фильма нет.');
   ...
end.
{ ======================================== }
```

Нетрудно видеть, что поиск фильма производится сравнением его названия `Film` с каждым элементом массива `Films`. При числе фильмов порядка нескольких тысяч этот поиск будет выполняться достаточно быстро. А если количество данных увеличить в десять, сто, тысячу раз? Поиск, который мы реализовали выше, так называемый *полный перебор* или *линейный поиск*, станет слишком долгим. Что же делать?

К счастью, во многих случаях данные, которые мы храним в массиве, устроены так, что их можно расположить в некотором порядке. Совсем просто это сделать с числами, достаточно понятно со строками, общепринятый порядок для них хотя и носит сложное название «лексикографический», тем не менее знаком каждому из нас – именно так расположены слова в любом словаре. Кстати, чтобы найти некоторое слово в словаре, мы, конечно же, не просматриваем его «от корки до корки», а ищем, отталкиваясь от алфавита, сначала по первой букве, примерно догадываясь, где расположены слова, которые с нее начинаются, потом по второй и так далее. То есть знание о том, что данные упорядочены, существенно помогает найти нужную информацию.

В общем случае в упорядоченном массиве используется так называемый *бинарный поиск*. Идея его заключается в следующем. На первом шаге находим середину массива. Сравниваем «средний» элемент с тем данным, которое мы хотим найти. Если средний элемент больше, значит, искать нужно в левой половине, иначе – в правой. В любом случае сдвигаем одну из границ области поиска, в результате чего она сужается вдвое. Повторяем до тех пор, пока не найдем или пока область поиска не сузится до одного элемента. Либо этот элемент – искомый, либо искомого элемента в массиве нет.

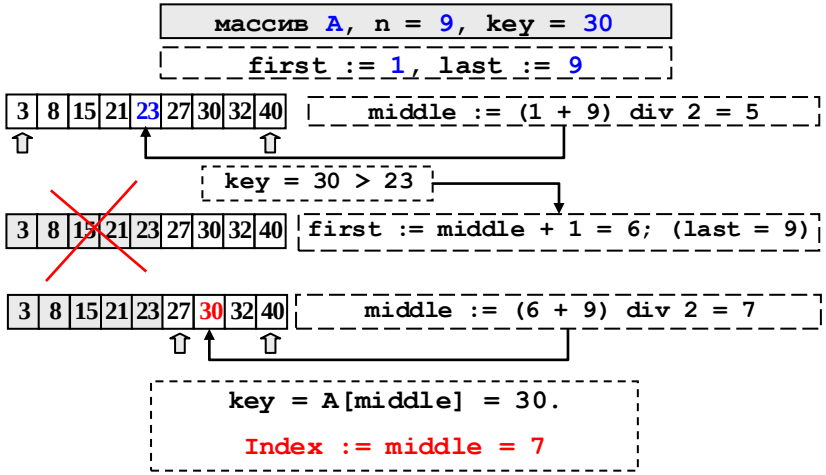

Рис. 6.1. Бинарный поиск в упорядоченном массиве

Реализовать этот алгоритм предоставляем читателю самостоятельно, а мы рассмотрим, как же собственно добиться упорядоченности данных. Процедура эта носит название *сортировка* и выполняется огромным количеством различных способов [10], простейшие из которых занимают всего десяток строк кода. Одну из таких простых сортировок мы сейчас и изучим – сортировка «пузырьком».

```
{ ================================================== }
{ Пример 6.5 }
{ Пузырьковая сортировка }
Program BubbleSort;

{$APPTYPE CONSOLE}

const
   MaxNum = 100000;
type
   ArrayOfNumbers = array[1..MaxNum] of Integer;
var
   N, i, j: Word;
   Numbers: ArrayOfNumbers;
   temp: Integer;
begin
   ...
   { Сортировка }
   for i := 2 to N do
      for j := N downto i do
```

```
        if Numbers[j] < Numbers[j - 1] then
        begin
          temp := Numbers[j];
          Numbers[j] := Numbers[j - 1];
          Numbers[j - 1] := temp;
        end;
   ...
end.
{ ================================================= }
```

Замечания по приведенному коду. Базовая операция большинства сортировок носит условное наименование «сравнить и переставить» – два элемента, расположенных не в том порядке, в каком требуется, меняются местами. В данном алгоритме эта операция применяется следующим образом. Массив условно делится на две части. Левая считается упорядоченной, из правой элементы перемещаются в левую, как бы «всплывают», что обеспечивает внутренний цикл. За одну итерацию внешнего цикла самый меньший (при сортировке по возрастанию) элемент правой части оказывается в точности рядом с левой частью, размер которой таким образом каждый раз увеличивается на единицу.

Рассмотренная нами сортировка не относится к числу самых быстрых, зато уж точно одна из самых простых. Желающие подробнее познакомиться с богатым миром алгоритмов сортировки могут обратиться к прекрасной книге Кнута «Искусство программирования» [10].

3.2. Записи

Рассмотрим пример. Объектом обработки многих программ являются даты, то есть данные вида «16 сентября 1992 г.». Очевидно, что значение этого данного состоит из разнотипных компонент: число месяца – это целое из диапазона 1..31, месяц, скорее всего, следует моделировать с помощью перечислимого типа, а год – это либо просто целое (если отрицательные числа интерпретируются как годы до нашей эры), либо некоторый диапазон, отвечающий условиям применения программы.

Моделирование значения типа «дата» не может быть произведено с помощью регулярного типа, так как компоненты принадлежат разным типам. Для такого рода случаев в языке Pascal имеется специальный способ конструирования типа в виде так называемых *записей*.

Значение типа «запись» состоит из ряда *полей*, каждое из которых принадлежит заданному при объявлении записи типу данных. Поля записи именуются, что обеспечивает возможность прямого доступа к значению поля по имени. Общий вид объявления следующий:

```
<имя типа> = record
   <имя поля 1>: <тип поля 1>;
   <имя поля 2>: <тип поля 2>;
   ...
   <имя поля N>: <тип поля N>;
end;
```

Здесь **record** (запись) и **end** – зарезервированные слова, ограничивающие описания полей, `имя поля` – идентификатор, `тип поля` – произвольный тип. Тип данных «дата» может быть задан с помощью следующей последовательности объявлений:

```
type
   Day = 1..31;
   Month = (Jan, Feb, Mac, Apr, May, Jun, Jul, Aug, Sep,
     Oct, Nov, Dec);
   Date = record
     D: Day;
     M: Month;
     Y: Integer;
   end;
```

С данным типом автоматически связываются операции присваивания и *выделения поля записи* – селектор полей. *Селектор* обозначается точкой после имени переменной типа «запись». За точкой должно следовать имя выделяемого поля.

Пусть, например, сделано объявление переменной:

```
var
   Today: Date;
```

Мы можем присвоить компонентам этой переменной значения с помощью следующих операторов присваивания:

```
Today.D := 16;
Today.M := Sep;
Today.Y := 1992;
```

В языке Pascal нет возможности одним оператором присваивания выполнить общую пересылку всех компонент значения записи. Это можно сделать только при задании типизированной константы-записи:

```
const
   Birthday: Date = (D: 5; M: Jan; Y: 1967);
```

Здесь значение каждого поля указывается после имени самого поля через двоеточие, поля отделяются друг от друга точкой с запятой.

Если же необходимо присвоить значения большому числу компонент одной переменной (или выполнить какие-то другие операции над ними), то, чтобы избежать записывания имени переменной, можно воспользоваться

специальным оператором **with**. Синтаксис этого оператора покажем на примере:

```
with Today do
begin
   D := 16;
   M := Sep;
   Y := 1992;
end;
```

В заголовке оператора указывается имя переменной, а в его теле используются лишь имена полей. По синтаксису после зарезервированного слова **do** может следовать только один оператор, поэтому приходится пользоваться операторными скобками.

Рассмотрим пример работы с записями. Ниже приведена программа вычисления «завтрашней» даты. Отметим, что в языке Pascal отсутствуют средства ввода и вывода значений записей как единого целого. Это понятно, так как при выводе или вводе сложно структурированной записи сразу возникает проблема форматирования, которая не может быть решена неким универсальным способом (при этом в записях могут встречаться значения перечислимого типа, что тоже является проблемой при вводе и выводе). Чтобы не загружать текст программы процедурами покомпонентного ввода и вывода записи «дата», которая содержит поле «месяц» перечислимого типа, мы зададим исходную дату операторами присваивания, а результат напечатаем с цифровым обозначением месяца.

```
{ ================================================= }
{ Пример 6.6 }
{ Вычисление даты «завтра» }
Program DateOfTomorrow;

{$APPTYPE CONSOLE}

type
   Day = 1..31;
   Month = (Jan, Feb, Mac, Apr, May, Jun, Jul, Aug, Sep,
     Oct, Nov, Dec);
   Date = record
     D: Day;
     M: Month;
     Y: Word;
   end;
var
   { число дней в каждом месяце }
   DM: array[Month] of Day = { объявление с инициализацией }
      (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
   Today, Tomorrow: Date; { сегодня, завтра }
```

```
begin
   with Today do
   begin
      D := 8;
      M := Jun;
      Y := 2005;
   end;
   if (Today.Y mod 4) = 0 then { если год високосный, }
      DM[Feb] := 29; { то в феврале 29 дней }
   if Today.D > DM[Today.M] then
   begin
      Writeln('Ошибка: Неправильная дата');
      Halt;
   end;
   if (Today.M = Dec) and (Today.D = 31) then { если новый год }
   begin
      Tomorrow.D := 1;
      Tomorrow.M := Jan;
      Tomorrow.Y := Succ(Today.Y);
   end
   else begin
      Tomorrow.Y := Today.Y;
      if Today.D = DM[Today.M] then { если новый месяц }
      begin
         Tomorrow.M := Succ(Today.M);
         Tomorrow.D := 1;
      end
      else begin
         Tomorrow.M := Today.M;
         Tomorrow.D := Succ(Today.D)
      end;
   end;
   Writeln(Tomorrow.D, '-', Ord(Tomorrow.M) + 1, '-',
     Tomorrow.Y);
   Readln;
end.
{ ================================================= }
```

Отметим, что в данной программе используется тот факт, что високосный год – это год, номер которого делится нацело на 4. На самом деле это не вполне верно. Исключением из этого правила является каждый год, которым заканчивается век, если номер самого века не делится на 4. Так, не были високосными годы 1700, 1800, 1900. И соответственно не будет год 2100. Таким образом, приведенная программа будет корректно работать лишь до 2099 года.

Поля записи, так же как элементы массива, могут иметь любой, в том числе сложный, тип данных, то есть быть, в свою очередь, массивами или записями. Эта возможность позволяет создавать хорошо структурированные информационные модели. Допустим, что объектом обработки некоторой программной системы учета кадров является информация о сотрудниках предприятия, включающая фамилию сотрудника, дату его рождения и общую оплату его труда по месяцам текущего года. Информационная модель объекта обработки может быть представлена в виде следующей записи (с использованием типов, определенных нами выше):

```
type
   Income = array[Month] of Word;
   Person = record
      Name: String[20];
      Birth: Date;
      Salary: Income;
   end;
var
   P: Person;
```

Доступ к полям переменной P можно получить следующим образом:

```
P.Name := 'Иванов И.И.';
P.Birth.D := 20;
P.Salary[Oct] := 5600;
```

3.3. Записи с вариантами

Запись представляет собой весьма общий способ конструирования новых типов данных. Однако существуют задачи, в которых простого объединения полей недостаточно для адекватного представления объектов предметной области. Пусть нам требуется составить программу, обслуживающую некоторую издательскую систему. Одна из функций этой программы – редактирование списков литературы, входящих в издания. Попытаемся описать объект «список литературы» – для краткости далее будем называть его «библиография».

Договоримся, что в библиографию входят два типа ссылок: ссылки на журнальные статьи и ссылки на книги. Ссылка на статью обычно содержит фамилии авторов, название статьи, название журнала, том, номер, страницы (номер начальной и номер последней страниц) и год. Ссылка на книгу также включает авторов и название, плюс к этому наименование

издательства, год и общее количество страниц. Как видно, в целом описания ссылок на статью и книгу различны, а значит, вообще говоря, они принадлежат разным типам. Это не позволяет сформировать тип данных «библиография» в виде естественно напрашивающейся структуры – одномерного массива ссылок.

Конечно, можно организовать два массива – один для ссылок на статьи, другой для ссылок на книги. Однако такой подход не отражает реальной ситуации и затрудняет работу – список-то ведь на самом деле один. Автор библиографии, составляя общий список литературы, руководствуется не видом ссылок, а некоторыми содержательными соображениями и перемешивает ссылки на книги и статьи в списке. А значит, при искусственном разбиении списка на две части возникнет необходимость организовывать в программе слияния двух списков. Кроме того, потребуются индивидуальные процедуры для обработки ссылок разных типов.

Таким образом, возникает вопрос: нельзя ли «сделать вид», что два по существу разных объекта принадлежат к одному типу данных? В рассматриваемой ситуации ответ на этот вопрос положителен, а метод решения заключается в применении *записей с вариантами*.

Запись с вариантами имеет следующий синтаксис:

```
record
   <список общих полей>;
   case <порядковый тип | переменная: порядковый тип> of
   <список констант 1>: (<список полей 1>);
   ...
   <список констант N>: (<список полей N>);
end;
```

Описание полей такой записи состоит из двух частей: после зарезервированного слова **record** следует описание полей, *общих* для всех разновидностей объектов, «покрываемых» данным типом. В нашем случае это описание полей «авторы», «название», «год». (Мы введем для них обозначения `Author`, `Title` и `Year` соответственно и будем относить первые два поля к типу `String`, а последний – к диапазону 1800..2100.)

Далее следует конструкция, внешне напоминающая известный нам оператор множественного выбора, но играющая несколько другую роль. В ее заголовке указывается тип данных, характеризующий различные *варианты* объектов, объединенных под одной «крышей». Обычно это перечислимый тип, именующий варианты (в общем случае, как указано выше, любой порядковый). Для нашего примера мы выберем перечислимый тип:

```
type
```

```
RefType = (Paper, Book);
```

Вместе с типом можно указать переменную, с ее помощью можно будет определять, к какому из вариантов принадлежит конкретный экземпляр записи. Необходимо, впрочем, отметить, что эта возможность не реализуется автоматически. Программист должен сам присвоить полю значение, характеризующее вариант. Более того, ни компилятор, ни системные средства времени исполнения не контролируют правильности заполнения этого поля – эта забота также целиком лежит на плечах программиста. Так что в нашем примере описание ссылки на статью может быть ошибочно помечено как книга. Тем не менее мы введем поле интерпретации записи:

```
case Ref: RefType of
```

Далее следуют описания вариантов записи, идентифицируемые константами выбора. Это списки индивидуальных для вариантов полей, элементы в них разделяются точкой с запятой, сами списки заключены в скобки.

Представим теперь полное описание типа данных «библиография»:

```
type
  PaperPages = record
    { начальная и конечная страницы статьи }
    Low, Hi: Word;
  end;
  RefType = (Paper, Book);   { имена вариантов ссылок }
  Reference = record         { ссылка                 }
    Author,                  { автор                  }
    Title: Str;              { заголовок              }
    Year: 1800..2100;        { год                    }



    case Ref: RefType of
    { статья }
    Paper: (Journal: String;     { название журнала }
      Volume,                    { том              }
      Number: Byte;              { номер            }
      PPages: PaperPages);       { страницы         }

    { книга }
    Book: (Publisher: String;       { издатель       }
      BPages: Word);                { кол-во страниц }
  end;
const
  NumRef = 100; { количество ссылок в библиографии }
type { библиография }
```

```
    Bibliography = array[1..NumRef] of Reference;
```

Если сделаны следующие объявления переменных:

```
var
   MyBookRefList: Bibliography;
   Article, Monograph: Reference;
```

то допустимы, например, такие операторы присваивания:

```
with Article do
begin
   Author := 'K.V.Ramani, M.R.Patel, and S.K.Patel';
   Title := 'An Expert System for Drug
   Preformulation in' + ' a Pharmaceutical Company';
   Year := 1992;
   Ref := Paper;
   Journal := 'Interfaces';
   Volume := 22;
   Number := 2;
   PPages.Low := 101;
   PPages.Hi := 108;
end;

with Monograph do
begin
   Author := 'В.Г.Абрамов, Н.П.Трифонов, Г.Н.Трифонова';
   Title := 'Введение в язык Паскаль';
   Year := 1988;
   Ref := Book;
   Publisher := 'Москва, «Наука»';
   BPages := 320;
end;

MyBookRefList[2] := Article;
MyBookRefList[11] := Monograph;
```

В заключение раздела отметим, что рассматриваемый вид записи фактически представляет собой набор шаблонов для разнотипной интерпретации одного и того же участка оперативной памяти.

3.4. Множества

Одна из самых распространенных задач, с которыми имеет дело человек в своей повседневной деятельности, – работа с текстами. Задачу эту решает специальный вид программ под названием «текстовые редакторы», один из самых главных объектов в которых – шрифт, определяющий начертание символов. Обычно в понятие «начертание» включают три признака,

характеризующих вид символов: полужирный, курсивный, подчеркнутый. Каждый из признаков может быть установлен или снят, и все они могут комбинироваться в любом сочетании. Обсудим, какими средствами можно адекватно представить в программе подобный объект.

Идея первая. Создаем запись вида:

```
FontStyle = record
   Bold: Boolean;
   Italic: Boolean;
   Underline: Boolean;
end;
```

Каждый из признаков можно установить (значение **True**) или снять (значение **False**). И все бы хорошо. Но!

Соображение первое. Очевидно, что для представления каждого признака требуется всего один бит – значит, мы потратили в восемь раз больше памяти, чем нужно в минимальном варианте. Казалось бы, что такое три байта на структуру! На самом деле довольно немало. Ведь все это – побочные затраты на оформление текста. Вообще говоря, они должны быть как можно меньше. Хорошо еще, что признаков всего лишь три. А если учесть больше?

Соображение второе. Нам вполне может потребоваться следующая операция. Имеются два фрагмента текста, каждый со своим оформлением. Нужно переоформить их так, чтобы сохранить те признаки начертания, которые имеются в каждом фрагменте, и снять разные. Для выполнения операции придется попарно сравнивать поля соответствующих каждому фрагменту записей.

Итак, первое решение оказалось не слишком удачным.

Для ситуаций, подобных описанной, в языке Pascal существует специальный тип, который наиболее адекватно их отражает, – *множество*.

Множество конструируется на основе базового типа, в качестве которого может выступать любой порядковый тип, имеющий не более чем 256 допустимых значений. Чаще всего в качестве основы для типа множества выступает перечислимый тип.

Значениями типа «множество» являются всевозможные подмножества, формируемые из элементов базового типа.

Общий вид объявления типа «множество» следующий:

```
<имя типа> = set of <базовый тип>;
```

Вот какие объявления можно сделать для поддержки рассмотренной выше задачи:

```
type
   FontStyle = (fsBold, fsItalic, fsUnderline);
```

```
FontStyles = set of FontStyle;
```

Создаем переменные:

```
var
  F1, F2, F3: FontStyles;
```

Теперь установим признаки.

```
F1 := F1 + [fsBold, fsItalic];
F2 := F1 - [fsBold];
```

Здесь использованы операции «объединение множеств» (+) и «разность множеств» (–), а также конструкция, создающая множество на основе указанных элементов:

```
[<список значений базового типа>]
```

Квадратные скобки в данном случае являются элементами синтаксиса языка, а не метасимволами формы Бэкуса–Наура.

Укажем все операции, применимые к переменным типа «множество».

Наименование	Обозначение	Тип результата
Объединение	+	Множество
Разность	–	Множество
Пересечение	*	Множество
Содержится в	<=	Булевский
Содержит	>=	Булевский
Равенство	=	Булевский
Неравенство	<>	Булевский
Является элементом	**in**	Булевский

Операция **in** фактически является более удобной формой операции <=, позволяя вместо проверки

```
[fsBold] <= F2
```

записать более понятное

```
fsBold in F2
```

С помощью множеств задача о выделении общего оформления решается одной строкой вида:

```
F3 := F1 * F2;
```

4. Приведение типов

Проблемы преобразования значений из одного типа данных в другой (приведения типов) мы уже касались в главе 4. Там мы рассмотрели

некоторые частные случаи, связанные с приведением числовых типов данных, а также преобразования строк в числа и наоборот.

В условиях когда программисту позволено создавать собственные типы данных, решение проблемы преобразования типов требует выработки некоторого общего подхода, и прежде всего введения строгого определения, какие типы считать одинаковыми.

4.1. Идентичные типы

С содержательной точки зрения два типа являются одинаковыми или *эквивалентными*, если они имеют совпадающие множества значений и наборы операций над значениями. Проблема состоит в возможностях компилятора на уровне синтаксического анализа текста программы распознать эквивалентность типов.

Рассмотрим следующие объявления типов данных:

```
type
  T1 = array[1..20] of array[1..30] of Real;
const
  M = 20;
  N = 30;
type
  T2 = array[1..M, 1..N] of Real;
```

Ясно, что в конечном итоге типы `T1` и `T2` задают один и тот же класс прямоугольных вещественных матриц с 20 строками и 30 столбцами, а различные способы объявления отражают лишь разные взгляды на один и тот же предмет. Вместе с тем установление эквивалентности этих типов требует анализа структуры объявлений, что представляет собой довольно трудную задачу, если учесть многовариантность способов определения типов, принятую в языке Pascal. Приведенный пример дает представление о понятии *структурной эквивалентности* типов данных.

В языке Pascal не выполняется структурный синтаксический анализ эквивалентных типов. В нем приняты более «сильные» требования, существенно снижающие круг типов, распознаваемых как эквивалентные. Этот подход носит название *именной эквивалентности*. Полагается, что два типа `T1` и `T2` являются *эквивалентными*, если `T1` и `T2` есть один и тот же идентификатор типа или `T1` объявлен как «равный» идентификатору `T2`. Эквивалентные в этом смысле типы в терминологии, принятой при описании языка Pascal, называются *идентичными*. Например, следующие объявления задают идентичные типы:

```
type
  T1 = Real; { T1, T2 и T3 являются идентичными }
  T2 = Real;
  T3 = T1;
  T5 = array[1..50] of Word;
  T6 = T5; { T5 и T6 являются идентичными }
```

Заметим, что такой подход делает все анонимно объявленные типы неэквивалентными. Исключение делается лишь для переменных, объявленных в одном операторе через запятую. Например, переменные A и B, объявленные как

```
var
  A: record
       x, y: Integer;
     end;
  B: record
       x, y: Integer;
     end;
```

принадлежат разным типам и при наличии в программе оператора присваивания A := B; компилятор сообщит об ошибке вида «Несовпадение типов».

В то же время переменные C и D, объявленные как

```
var
  C, D: record
          x, y: Integer;
        end;
```

принадлежат одному типу.

Разумеется, выяснение эквивалентности двух типов интересно не само по себе, а в контексте вопроса о возможности выполнения некоторых совместных действий над значениями разных типов. В зависимости от вида действий к обрабатываемым значениям предъявляются те или иные требования по *согласованию типов*. Например, при передаче фактического параметра в подпрограмму всегда требуется, чтобы он имел тип, идентичный соответствующему формальному параметру[1]. Однако при выполнении присваивания в зависимости от типов левой и правой частей в некоторых ситуациях требуется идентичность типов (как в вышеприведенном примере), а в других – более слабое согласование типов, называемое *совместимостью по присваиванию*. Мы, например, уже знаем, что вещественной переменной можно присвоить целое значение, которое просто автоматически будет преобразовано к вещественной форме представления.

[1] Более подробно речь об этом пойдет в главе 7.

4.2. Совместимые типы и совместимость по присваиванию

Два типа являются *совместимыми*, если выполняется хотя бы одно из следующих условий (мы приводим условия, касающиеся только рассмотренных типов данных):

- оба типа вещественные (имеются в виду различные виды вещественного типа);
- оба типа целые;
- один тип есть диапазон другого;
- оба типа являются диапазонами одного и того же типа;
- оба типа являются упакованными строковыми типами с одинаковым числом компонент (упакованным строковым типом называется одномерный массив символов, то есть **array**[M..N] **of** Char);
- один тип является строкой, а другой строкой, упакованной строкой или символьным типом;
- оба типа являются множествами над совместимыми типами;
- типы идентичны.

Наиболее часто проблема преобразования типов возникает при выполнении оператора присваивания. Возможность совмещения разнотипных правых и левых частей оператора присваивания регулируется набором правил, определяющих *совместимость по присваиванию*. Значение типа T2 является совместимым по присваиванию со значением типа T1, то есть возможно присваивание T1 := T2, если выполняется хотя бы одно из следующих условий (мы снова приводим правила, касающиеся только рассмотренных типов):

- оба типа идентичны;
- оба типа – совместимые порядковые типы;
- оба типа являются вещественными;
- T1 есть вещественный тип, а T2 – целый;
- оба типа есть строковые типы;
- T1 – строковый, а T2 – символьный;
- T1 – строковый, а T2 – упакованный строковый;
- оба типа есть совместимые упакованные строковые типы.

Еще раз напомним, что совместимость по присваиванию в общем случае означает возможность автоматического приведения типа значения из правой части к типу переменной в левой части оператора присваивания. Реализуется эта возможность компилятором автоматически путем вставки в исполняемый модуль дополнительных команд преобразования.

5. Выводы

Данная глава посвящена изучению способа абстрагирования от возможностей конкретного компьютера и его архитектуры, состоящего в конструировании новых типов данных. Конструирование новых типов данных приближает программу к решаемой задаче и делает ее понятнее для разработчика, что позволяет избегать массы глупых ошибок, допущенных из-за обычной невнимательности, усталости, плохого настроения. Говоря о языках программирования высокого уровня, мы имели в виду их приближенность к предметной области. Не правда ли, возможность создания типов данных в полном соответствии с решаемой задачей существенно повышает уровень языка и упрощает процесс разработки? В главе рассмотрены соответствующие средства, встроенные в Object Pascal: перечислимый тип, диапазон, массив, запись, множество.

Литература

1. Богатырев Р. Летопись языков Паскаль // Мир ПК. –№ 4.–2001.

2. Богатырев Р. От Паскаля к языку Zonnon: реализация новых идей на платформе .NET // Мир ПК.–2003.–№ 9.

3. Большая советская энциклопедия. Издание третье. – М.: Советская энциклопедия, 1970.

4. Брэдли Д. Программирование на языке ассемблера для персональной ЭВМ фирмы IBM.–М.: Радио и связь, 1988.

5. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание. – Бином, 1998.

6. Касперски К. Техника оптимизации программ. Эффективное использование памяти.– СПб.: БХВ-Петербург, 2003.

7. Кетков А., Кетков Ю. Практика программирования: Бейсик, Си, Паскаль. Самоучитель.–СПб.: БХВ-Петербург, 2001.

8. Кетков А., Кетков Ю. Практика программирования: Visual Basic, C++ Builder, Delphi. Самоучитель.–СПб.: БХВ-Петербург, 2002.

9. Кнут Д.Э. Искусство программирования. Т. 1. Основные алгоритмы.–М.:Вильямс, 2000.

10. Кнут Д.Э. Искусство программирования. Т. 3. Сортировка и поиск.–М.:Вильямс, 2004.

11. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ.–М.:МЦНМО, 1999.

12. Кэнту М. Delphi 7 для профессионалов.–Спб.:Питер, 2004.

13. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования.–М.: Мир, 1982.

14. Модернизация и ремонт персонального компьютера.– [http://www.allcompinfo.com]

15. Новичков А. Система генерации проектной документации Rational SoDA.– [http://www.citforum.ru/programming/digest/soda.shtml]

16. Олифер В.Г., Олифер Н.А. Сетевые операционные системы.–СПб.: Питер, 2002.

17. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ.– 4-е изд.–СПб.: Питер; М.: Издательско-торговый дом «Русская Редакция», 2001.

18. Служба тематических толковых словарей. – [http://www.glossary.ru/]

19. Стивенс Р. Delphi. Готовые алгоритмы.– Спб.:Питер, 2004.

20. Тейксейра С., Пачеко К. Borland Delphi 6. Руководство разработчика. –Вильямс, 2002.

21. Шень А. Программирование: теоремы и задачи.–М.:МЦНМО, 2004.

22. Юров В. Assembler.–СПб.: Питер, 2002.

23. Cantu M. Essential Delphi.– [www.marcocantu.com/edelphi]

24. Cantu M. Essential Pascal.– [www.marcocantu.com/epascal]

25. Cantu M. Mastering Borland Delphi 2005.–Sybex, 2005.

26. Cantu M. Mastering Delphi 7.–Sybex, 2003.

27. Clark R., Koehler S. The UCSD Pascal Handbook.–Prentice-Hall, 1982.

28. Dahl O., Dijkstra E., Hoare C.A.R. Structured Programming.–London, England: Academic Press, 1972.

29. Gutknecht J., Zueff E. Zonnon Language Report. Draft.–ETH Zurich, June 2003.

30. Gutknecht J., Zueff E. Zonnon for .NET, A Language and Compiler Experiment. // Computer Systems Institute, ETH Zürich, Switzerland. JMLC Conference.–August 2003.

31. Gutknecht J. Zonnon: A .NET Language Beyond C# // Moscow: Microsoft Conference, June 15-17, 2003.

32. Gutknecht J., Wirth N. The Oberon System // Software – Practice and Experience.–Vol.19, № 9.–1989.–p.857-893.

33. Jacobson I., Christerson M., Jonsson P., Overgaard G. Object-oriented Software Engineering.–Workingham, England: Addison-Wesley Publishing Company, 1992.

34. Jensen K., Wirth N. PASCAL – User Manual and Report, ISO Pascal Standard.–1974.

35. Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture.– Intel Corporation: 1997.

36. Meyer B. Object-oriented Software Construction.–Prentice Hall, 1988.

37. Rumbaugh H., Blaha M., Premarlani W., Eddy F., Lorensen W. Object-Oriented Modeling and Design.–Prentice-Hall, 1995.

38. Sedgewick R. Algorithms.–Reading, MA: Addison-Wesley, 1983.

39. Tesler L. Object Pascal Report. // Structured Language World, Vol.9, No.3.– 1985.–p. 10-14.

40. Thiriez H. Modelling of an interactive scheduling system in a complex environment // European Jornal of Operational Research. –1991.– №50.–p.37-47.

41. Wirth N. Program Development by Stepwise Refinement // Communications of the ACM vol.26(1).– January 1983.

42. Wirth N. The Programming Language Pascal // Acta Informatica, 1.–1971.–p. 35-63.

43. Wirth N. Programming in Modula-2.–Springer-Verlag, 1982.

44. Wirth N. The programming language Oberon // Software – Practice and Experience, Vol.18, № 7.–1988.–p.671-690.

45. Wirth N. Programming in Modula-2.–Springer, 1974.

46. Wirth N. Pascal and its Successors // Conference on Computer Pioneers.–Bonn, 2001.