А.О. Грудзинский, И.Б. Мееров, А.В. Сысоев

Методы программирования

Курс на основе языка Object Pascal

Нижний Новгород, 2005

Краткое содержание

Содержание

Введение

Пусть мысли, заключенные в книгах, будут твоим основным капиталом, а мысли, которые возникнут у тебя самого, – процентами на него.

Фома Аквинский

Здравствуйте, уважаемый Читатель! Хотите отправиться в путешествие? Куда? По бескрайним просторам моря информационных технологий. Точнее по той его части, которую принято называть “Разработка программного обеспечения”. Обещаем немало интересного.

Не любите покупать кота в мешке? Справедливо. Что ж, чтобы Вам было легче принять решение, опишем условия круиза.

Итак, наше путешествие пройдет на комфортабельном лайнере Delphi[1], построенном на верфях Borland Software Corporation. Лайнер оборудован по последнему слову техники: в Вашем распоряжении редактор исходных текстов с поддержкой языка высокого уровня Object Pascal, компилятор, сборщик, отладчик. Каждому пассажиру предоставляется подробное справочное руководство.

Ограничения по возрасту и начальному уровню знаний отсутствуют. На борт принимаются как не знакомые с миром программирования вообще, но желающие приобрести этот ценный опыт, так и те, кто уже связал свою профессиональную судьбу с информационными технологиями. Уверены, что каждый во время нашего путешествия найдет что-то для себя, либо новые знания и навыки, либо систематизацию и упорядочивание имеющихся представлений.

По ходу плавания предполагаются стоянки в следующих пунктах:

- общие вопросы создания программ, включая основные этапы процесса разработки и используемые средства;
- краткие сведения о среде исполнения программ;
- основные элементы и положения языка программирования Object Pascal;
- различные способы описания моделей объектов предметной области с помощью конструирования типов данных;
- вопросы динамического управления памятью и работы с файлами;
- технологии разработки: структурная, модульная, объектно-ориентированная.

В каждом пункте маршрута Вас ожидает экскурсионная программа с просмотром и обсуждением множества задач, вариантов их решения, необходимых языковых средств, примеров программ.

---

[1] Желающие вкусить романтики старины могут следовать в кильватере на все еще весьма крепком Turbo Pascal 7.0.

Наш круиз может стать Вашим первым шагом на пути к тому времени, когда сообщество программистов признает Вас морским волком в области разработки программного обеспечения.

Ну как? Нам удалось Вас заинтересовать? Тогда в путь!

1. Решение задач с использованием вычислительной техники

Компьютерная программа делает то, что вы приказали ей сделать, а не то, что вы хотели, чтобы она сделала.
“Третий закон Грира”

Представьте себе такую ситуацию: Вы руководитель отдела в программистской фирме. К Вам пришел заказчик со следующим предложением: “Мне нужна программа для нахождения равновесной цены на рынке”.

Вы: “Отлично. Вы пришли в нужное место. У нас лучшие специалисты по нахождению цен на рынке, и именно равновесных”.

Что? Что-то не так? Вы думаете, так явно себя рекламировать не стоит? Может быть. Впрочем, суть не в этом. Допустим, Вы с заказчиком обо всем договорились, убедили его, что лучших исполнителей ему не найти, и он окрыленный ушел, насвистывая: “Мы рождены, чтоб сказку сделать былью”. А Вы остались и призадумались: “А что теперь делать?”

Описанная ситуация отнюдь не является надуманной. И вопрос этот можно более четко переформулировать так: “Пусть у нас есть задача, для решения которой мы хотим (нам необходимо, мы не можем обойтись без того, чтобы) использовать компьютер. Какие действия мы должны для этого выполнить?” Очевидный ответ типа: “Раз заказчик хочет программу – надо ее написать”, – на самом деле порождает еще больше вопросов. Как минимум: “А как? С чего начать?” Попытаемся разобраться.

Прежде всего, поскольку задача математическая (надеемся, этот факт не вызывает у Вас сомнения), начать наверное надо с попытки ее решения средствами математики.

Вспоминая школьный курс экономики, отметим, что зависимость спроса S и предложения D от цены товара P задаются некоторыми функциями $S = S(P)$ и $D = D(P)$. По мере роста цены товара спрос падает, а предложение увеличивается. Напротив, при падении цены товара происходит рост спроса и уменьшение предложения. В результате мы имеем ситуацию борьбы продавца и покупателя, цели которых противоположны. Продавец стремиться держать цену как можно выше, покупатель – купить товар как можно дешевле. При этом при чрезмерном завышении цены товара покупатель перестает его приобретать, и, наоборот, при ее занижении продавец не хочет более торговать этим товаром. В результате работы рыночных механизмов происходит стабилизация цены товара на некотором уровне, приемлемом для всех субъектов рынка. Для нахождения этой равновесной цены необходимо решить уравнение $S(P) = D(P)$, которое соответствует ситуации, когда весь товар распродается (спрос равен предложению).

Допустим, что вид функций S(P) и D(P) нам известен. Перенеся D(P) в левую часть, мы обнаруживаем, что задача определения соответствующей цены P* сводится к нахождению нулей некоторой известной функции f(P) = S(P) – D(P)[1].

Из курса математики известно, что, если функция y = f(x) является *непрерывной* на [a, b] и f(a) · f(b) < 0, то эта функция имеет *корень* на [a, b], т.е. такое x = x_0, что $x_0 \in [a, b]$, $f(x_0) = 0$.

Проблема нахождения значения x_0 состоит в том, что отнюдь не всегда уравнение f(x) = 0 может быть решено *аналитически*, то есть выведены формулы расчета его корней. Если аналитическое решение невозможно, то уравнение решают *численно*, – некоторым способом подбирают x′ такое, что значение функции f(x) в этой точке достаточно близко к нулю. При этом, как правило, не удается найти точное решение, но удается отыскать его с некоторой *удовлетворительной точностью*.

Формально вышесказанное может быть записано так: для решения задачи задается некоторое ε > 0, достаточно близкое к нулю, и требуется найти такое

$$x' \in [a, b], \text{ что } |f(x')| \le \varepsilon. \qquad (1.1)$$

Значение ε обычно вытекает из специфики задачи и определяет требуемую точность решения.

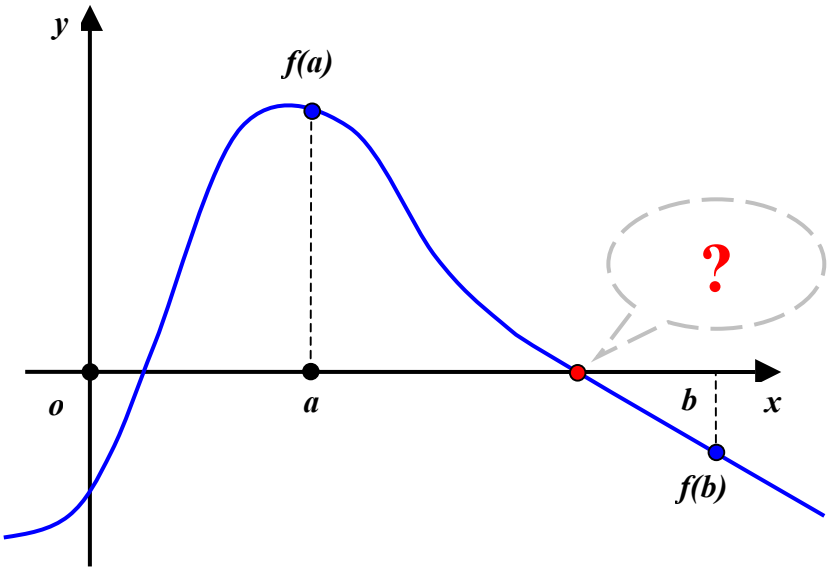


Рис. 1.1. Корни уравнения f(x) = 0

Для численного нахождения корней уравнения f(x) = 0 разработано немало *методов*. Среди них широко известными являются так называемые *итерационные методы*, которые строят цепочку точек $x_1, x_2, \ldots, x_n$, последовательно приближаясь к решению.

Общий вид формулы, задающей такой метод, выглядит следующим образом:

$$x_{n+1} = f^*(x_1, x_2, \ldots, x_n), \qquad (1.2)$$

где $x_1, x_2, \ldots, x_n$ – последовательность “кандидатов” на почетное звание корня, а f* – некоторая функция, определяющая сам метод.

В качестве примера таких методов можно привести: метод половинного деления (деления отрезка пополам, дихотомии), метод касательных, метод секущих.

---

[1] Заметим, что есть и более эффективные методы определения равновесной цены.

Рассмотрим кратко один из методов – *метод половинного деления*.

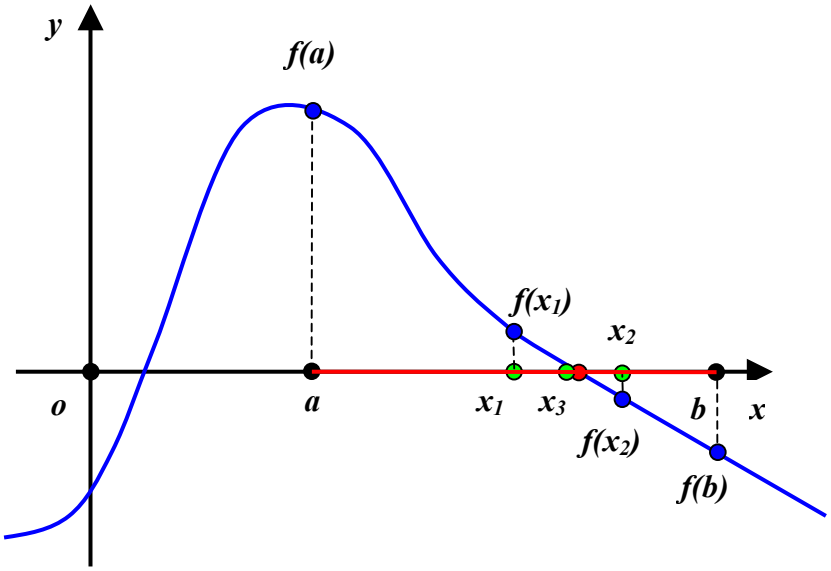


Рис. 1.2. Метод половинного деления

Формула, задающая метод, выглядит следующим образом:

$$x_{n+1} = \frac{a_n + b_n}{2}, \qquad (1.3)$$

где a_n и b_n есть текущие значения границ отрезка, на котором происходит поиск корня.

В начале работы метода $a_0 = a$, $b_0 = b$.

На каждом шаге метода происходит вычисление середины отрезка x_{n+1} по указанной формуле, после чего производится вычисление функции в точке x_{n+1}. Если выполнено условие останова ($|f(x_{n+1})| \le \varepsilon$), то найденная точка x_{n+1} считается решением, и метод заканчивает работу.

В случае если условие не выполнено, в качестве очередного отрезка для поиска выбирается тот из отрезков $[a_n, x_n]$ или $[x_n, b_n]$, для которого функция на концах принимает значения разных знаков.

Теперь дело за малым. Осталось реализовать приведенный метод – записать его на языке понятном компьютеру – и получить *программу* для нахождения корня уравнения $f(x) = 0$ на отрезке $[a, b]$.

Все просто, не правда ли? Помимо того, что мы не объяснили, что же такое “язык понятный компьютеру”, и как на нем что-нибудь написать, кажущаяся легкость связана еще и с тем, что в результате проведенного анализа нам стали хорошо известны ответы на следующие вопросы:

1. В чем суть задачи, что нам дано, и что нужно найти?
2. Какими математическими соотношениями описывается связь между исходными данными и требуемым результатом?
3. Какой метод необходимо применить для решения этой системы математических соотношений? В чем суть этого метода?

Думается, для того, чтобы ответить на эти вопросы, в рассмотренной задаче должно быть достаточно знаний выпускника средней школы. В реальной ситуации все будет значительно сложнее, и получение ответов может поставить серьезные проблемы даже перед

квалифицированными специалистами. Однако лишь после этого можно двигаться дальше, то есть, собственно, переходить к процессу, который обычно называют “программирование”. О деталях этого процесса мы поговорим позже, а пока подведем промежуточные итоги.

Итак, решение любой сколько-нибудь серьезной задачи с использованием компьютера отнюдь не начинается с составления программы (справедливости ради надо сказать, что и не заканчивается на этом). Прежде, чем можно будет приступить непосредственно к программированию, нужно, отталкиваясь от формулировки задачи, пройти ряд обязательных этапов. Далее в этой главе мы рассмотрим эти этапы, их назначение и выполняемые на каждом из них действия. Когда программа, наконец-то, будет готова, наступит очередь других важных этапов, о которых мы тоже поговорим.

Что касается самого процесса программирования (написания текста программы), то он в чем-то сродни написанию литературного произведения (хотя и более формализован), и является процессом творческим. Таким образом, для овладения им отнюдь не достаточно выучить “язык понятный компьютеру”. Так же как Александру Сергеевичу Пушкину для создания поэмы “Евгений Онегин” отнюдь не достаточно было глубокого знания русского языка, но потребовался и его великолепный талант и предшествующий опыт написания стихов и поэм и, что не менее важно, владение методами и техникой стихосложения, так и для создания, или как говорят профессионалы “разработки”, программ требуется освоение определенных приемов и методов, в совокупности образующих технологии программирования. О технологиях речь пойдет, начиная с пятой главы книги. А до этого мы успеем обсудить этапы решения задач с использованием компьютера, средства, которые помогают нам в этом процессе, поговорим о вещах, не связанных, казалось бы, с программированием напрямую, но имеющих, тем не менее, весьма важное значение (процессор, оперативная память, операционная система и т.д.), а также научимся составлять простейшие программы.

Впереди длинный путь, надеемся, что нам удастся сделать его достаточно интересным для Вас. Итак, приступим.

Постановка задачи

У разработчика программ не было бы особых проблем, если бы задачу ему формулировали примерно в таком виде: “Напиши оператор ввода трех чисел, вычисли значение по такой-то формуле, сравни результат с нулем…”, то есть прямо задавали некоторый план (*алгоритм*) действий.

Нетрудно догадаться, что на практике дело обстоит по-другому. Вот как, например, описывает один французский специалист данную ему Национальным географическим институтом постановку задачи: “Имеются соответствующие данные о самолетах, экипажах, доступном оборудовании, аэропортах, полетных задачах (пункты назначения, высота полета, скорость, степень срочности и т.д.) и карты ежедневных метеорологических наблюдений со спутников. Программная система должна предлагать эффективные решения по распределению самолетов, персонала и оборудования на каждый день работы и допускать оперативное изменение параметров и перераспределение ресурсов” [40].

В таком весьма общем виде формулируется множество заданий на разработку программных комплексов, по крайней мере, изначально. И хоть они выглядят несколько более внятно, чем известное “Пойди туда – не знаю куда, найди то – не знаю что”, все же разработать программу по такой постановке, конечно, невозможно. Добиться от заказчика ответов на все необходимые для

воплощения его потребностей в жизнь вопросы (а заодно и выяснить, что же он в реальности хочет – часто это не вполне совпадает с тем, что он говорит) – Ваша центральная, как исполнителя, задача. При этом в ходе получения ответов первоначальная постановка, скорее всего, существенно изменится.

Принципиальные вопросы, которые должны быть решены, прежде чем можно будет приступить к реализации программы, мы частично озвучили выше. Далее мы более детально рассмотрим все этапы решения задачи с использованием компьютера на конкретном достаточно простом примере.

Итак, пусть заказчику требуется программная система для выполнения регулярных расчетов *арендной платы за земельные участки*. Примем, что *арендная плата* – произведение стоимости одного квадратного метра земли на площадь участка. На самом деле это не всегда так, поскольку участок может состоять из земель разных типов, стоимость квадратного метра которых может быть различна. Таким образом, мы уже сделали первое *допущение*, которое на самом деле должно быть согласовано с заказчиком – должна ли наша будущая программа быть рассчитана на такую ситуацию или нет.

Допустим, заказчик клятвенно заверил нас, что учитывать возможную разную стоимость квадратного метра не требуется. Следующая наша задача – вычисление *площади участка*. Начиная ее решение, мы переходим к следующему этапу – *построение модели*.

Модель

Модель – формальное (как правило приближенное) описание изучаемого объекта или явления, отражающее интересующие нас аспекты.

Зачем нужна модель? Реальные объекты, о которых идет речь в задаче, чаще всего достаточно сложны, описываются массой параметров, существенной частью которых можно и нужно пренебречь. Например, пусть земельный участок имеет форму прямоугольника. Означает ли это, что его площадь есть произведение длины на ширину? Да? Вы хорошо подумали? А если участок имеет вид холма? Никто не говорил, что уровень земли по всему участку одинаков. Вот и еще одно *допущение*. Все вместе подобные допущения, ограничения, не принимаемые в расчет параметры и составляют модель объекта или явления.

Итак, формализуем условие нашей задачи, т.е. введем обозначения для исходных данных, требуемого результата и результатов промежуточных вычислений. Прежде всего, требуется формализовать понятие *земельный участок*.

Для начала, допустим, что в результате изучения плана местности и бесед с заказчиком, выяснилось, что участки имеют прямоугольную форму и уровень земли по всему участку одинаков. Тогда анализ постановки задачи приводит к следующей *системе параметров*.

Исходные данные:

a, ***b*** – размеры участка (стороны прямоугольника);

Price – стоимость одного квадратного метра земли.

Требуемый результат:

Rent – арендная плата.

Важные промежуточные результаты:

S – площадь участка.

Попробуем построить модель для этого случая. Так как участок имеет прямоугольную форму, вычисление его площади не представляет труда.

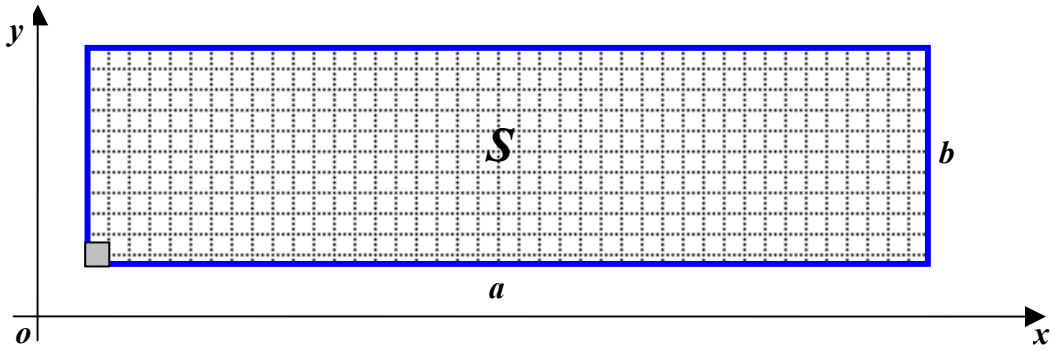


Рис. 1.3. Модель земельного участка прямоугольной формы

Как известно, площадь прямоугольника с учетом принятых обозначений вычисляется следующим образом:

$$S = a \times b \qquad (1.4)$$

Посмотрим теперь, что мы должны получить на выходе модели? Арендную плату ***Rent***. При принятых нами допущениях она вычисляется по следующей формуле:

$$Rent = S \times Price \qquad (1.5)$$

Формулы (1.4) и (1.5) совместно составляют *математическую модель* для решаемой задачи (для случая прямоугольной формы участка), связывая исходные данные и требуемый результат.

В принципе уже можно переходить к следующему этапу, но поскольку вариант с прямоугольной формой участка слишком прост, давайте рассмотрим чуть более общую ситуацию, имеющую к тому же под собой реальное обоснование – краевые участки, одна из сторон которых примыкает к реке или дороге и представляет собой кривую. Как сообщил нам заказчик, ни владелец земли, ни арендаторы не согласны “спрямить” эту сторону и настаивают на точном расчете.

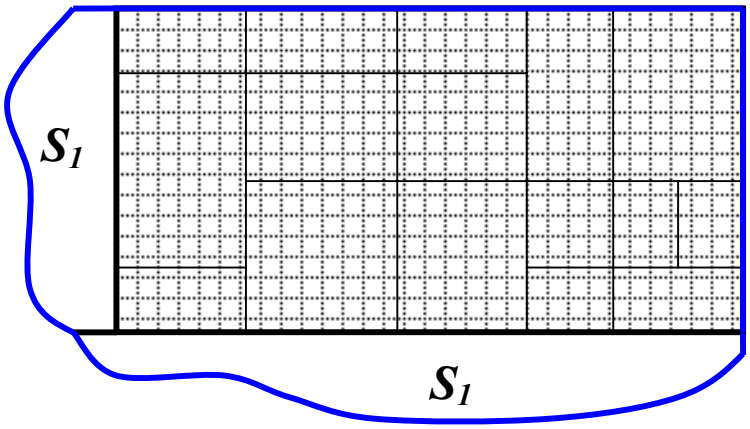


Рис. 1.4. Модель земельного участка общего вида

Чтобы справиться с этой проблемой, требуется немного больше знаний. Геометрической моделью объектов такого вида является так называемая *криволинейная трапеция*.

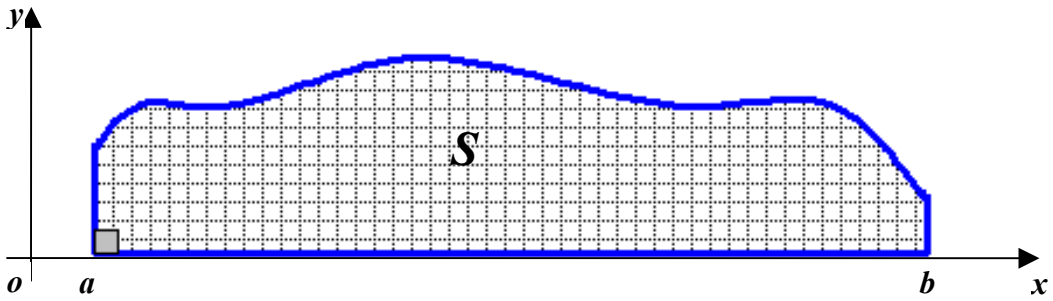


Рис. 1.5. Криволинейная трапеция

Внесем изменения в систему параметров.

Исходные данные:

a, ***b*** – границы участка;

f(x) – функция, описывающая “криволинейную часть” участка;

Price – стоимость одного квадратного метра земли.

Требуемый результат:

Rent – арендная плата.

Важные промежуточные результаты:

S – площадь участка.

Площадь криволинейной трапеции выражается при помощи определенного интеграла, требующего знания параметров трапеции, например, f(x) – функции, задающей график кривой (одной из “сторон” трапеции).

$$S = \int_a^b f(x)dx \qquad (1.6)$$

Читателю, не знакомому с понятием интеграла, достаточно знать, что существуют специальные правила интегрирования – их искусное применение позволяет получить формульное выражение, подставив в которое пределы интегрирования (параметры участка) мы получим численное значение площади. Соотношение (1.6) и формула вычисления арендной платы (1.5) задают математическую модель нашей задачи для случая криволинейных участков. Математическая модель является основой построения *информационной модели* задачи. Любой обрабатываемый в программе объект, помимо параметров, участвующих в расчетах, может характеризоваться информационными (описательными) параметрами. Например, в дополнение к данным, определяющим площадь участка и стоимость одного квадратного метра, заказчик может потребовать включить в информационную модель некоторые данные, идентифицирующие участок: его номер и фамилию владельца. Разумеется, это требование отразится на списке параметров, которыми мы описываем объекты задачи.

Построив модель, то есть, определившись, в конечном счете, со схемой получения из исходных данных требуемого результата, мы должны теперь для каждого выбранного для реализации варианта четко сформулировать способ расчета, то есть построить *метод вычислений*.

Метод

На этапе построения модели мы выделили два возможных варианта: участки прямоугольной формы и участки с одной криволинейной стороной.

Метод вычислений для первого случая тривиален, он в точности определяется формулами (1.4) и (1.5), расчет которых не представляет никаких проблем при любых исходных данных.

Со вторым вариантом несколько сложнее. Он тоже задается двумя формулами: (1.5) и (1.6), однако расчет площади по формуле (1.6) уже не так прост. Если для входящей в *исходные данные* функции f(x) известно аналитическое (формульное) выражение, то можно попытаться проинтегрировать формулу (1.6) и получить способ вычисления площади. Правда, функция f(x) может оказаться сложной и применить правила интегрирования будет непросто, но главная проблема даже не в этом. Дело в том, что точную формулу, выражающую кривизну реки или дороги на всех участках, взять просто неоткуда, и столь красивая математическая модель оказывается практически бесполезной.

Выходов из сложившейся ситуации три: первый – найти (придумать, если его не существует) метод расчета для выбранной модели, второй – изменить модель так, чтобы метод расчета для нее был известен, третий – вернуться к предыдущему этапу и попытаться построить другую модель. В данном случае мы пойдем по второму пути.

Начать надо с ответа на вопрос: “Чем вызвана возникшая проблема?”. Кажется, тем, что сторона участка имеет вид, не позволяющий выполнить точный расчет площади. Это конечно правильно, однако напомним – на самом деле мы работаем не с самим *объектом*, а с его *моделью*. А в модель криволинейная сторона попала потому, что владелец и арендатор не соглашаются ее “спрямить”. А можем мы сделать так, чтобы согласились? Можем. Для этого надо разбить криволинейную сторону на некоторое количество частей так, чтобы кривизна каждой части была достаточно мала, тогда ее можно будет спрямить с приемлемой погрешностью. Конечно, в результате мы не найдем точную площадь участка, однако если суммарная погрешность от замены исходной формы криволинейной стороны на ломаную линию будет достаточно мала, то такой подход можно использовать.

В результате мы приходим к следующему *методу вычисления* площади участка. На основе изучения карты или обследования на местности для каждого участка проводится разбиение криволинейной стороны на фрагменты, пригодные для замены отрезками прямой. Это разбиение порождает разбиение всего участка на обычные трапеции, в которых одна из боковых сторон является высотой. Их площади легко вычисляются по известной формуле, а площадь всего участка полагается равной сумме площадей трапеций. Число трапеций и длины их высот индивидуальны для каждого участка и определяются как разумный компромисс между точностью приближения кривой и трудоемкостью измерений.

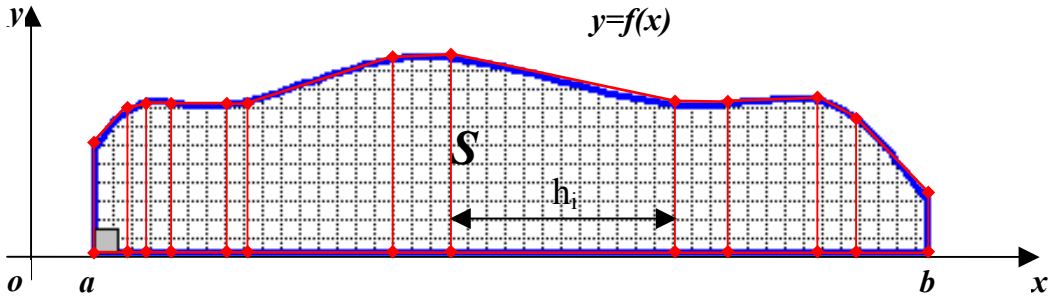


Рис. 1.6. Приближенное вычисление площади криволинейной трапеции

Представленный нами *метод* порождает новую *модель*, описывающую задачу, в соответствии с этим меняется и система используемых нами параметров.

Исходные данные:

a, ***b*** – границы участка;

N – число трапеций;

y_i – длины оснований трапеций;

h_i – высоты трапеций;

Price – стоимость одного квадратного метра земли.

Требуемый результат:

Rent – арендная плата.

Важные промежуточные результаты:

S – площадь участка.

В сделанных обозначениях формулу расчета площади участка мы можем записать так:

$$S = \sum_{i=0}^{N-1} h_i \times \frac{(y_{i+1} + y_i)}{2} \ , \qquad (1.7)$$

Теперь, имея в распоряжении исходные данные, мы сможем без труда выполнить все расчеты.

Итак, *метод вычислений* найден. Пора переходить к реализации? Не совсем так. Прежде нам предстоит пройти еще один достаточно важный этап – *построение алгоритма*.

Алгоритм

Алгоритм – точный план действий по решению задачи.

На этапе построения *модели* мы определили систему параметров, описывающих задачу, и установили соответствие между исходными данными и требуемым результатом. На этапе построения *метода вычислений* мы уточнили модель и сформировали точную схему выполнения расчетов. Но раз так, значит алгоритм, то есть план решения, готов! Берем исходные данные, подставляем в формулы расчета и получаем результат. Правильно? На самом деле нет. Подумайте, что значит: “Берем исходные данные”? Откуда берем? Не забудьте, что выполнять расчет будет программа, а она не сможет попросить пользователя: “Дай-ка, дружок, мне вон те циферки, я их сложу и умножу”. Итак, тот факт, что мы решили задачу математически, не означает, что мы сформировали алгоритм, который можно будет превратить далее в программу.

Чего же не хватает? Не хватает учета возможностей исполнителя, то есть компьютера. Возможности эти естественно ограничены, так что, несмотря на непрекращающийся с момента создания первой ЭВМ рост мощности компьютеров, в каждый текущий момент времени существуют задачи, которые современной вычислительной технике не под силу. Эти и множество других ограничений, присущих компьютерам в силу их внутреннего устройства, необходимо учитывать для грамотного составления алгоритмов, для чего может потребоваться снова вернуться к этапам построения модели и метода. Однако подробные рассуждения на эту тему уведут нас далеко в сторону, поэтому мы их отложим. Пока же достаточно отметить, что любой алгоритм, предназначенный для последующего воплощения в программу, должен

предусматривать действия по получению исходных данных, выполнению на их основе указанных расчетов, и выдаче с возможной предварительной обработкой результатов.

В нашей задаче исходные данные будут формироваться в результате измерений, выполняемых на участках. Будем считать, что эти результаты накапливаются в *текстовом файле*, а наша программа должна будет этот файл “читать” при запуске. Осталось лишь определиться с формой представления данных в этом файле. Например, она может быть следующей: номер участка, фамилия и инициалы владельца, количество отрезков разбиения, длины оснований трапеций (в метрах) и, наконец, высоты трапеций (в метрах).

Фрагмент такого файла может выглядеть следующим образом:

```
154
Иванов И.И.
3
40 37 50 45
7 20 10
```

Теперь нужно решить вопрос с представлением результатов расчетов. Например, они могут требоваться в виде справки на бумаге (то есть программа должна вывести их на принтер) и содержать идентифицирующую информацию участка, значения площади и арендной платы.

Итак, все необходимые решения приняты, можно записывать алгоритм. Вот только как? Модель и метод описываются с помощью общепринятой математической символики, а также просто словесно. А как выглядит язык записи алгоритма? Во-первых, алгоритм можно записать обычным “человеческим” языком. Однако каждое действие алгоритма должно пониматься однозначно, а разговорные языки обычно “грешат” многозначностью. Во-вторых, формальными языками записи алгоритмов являются *языки программирования*. О них речь пойдёт в следующих главах книги. Однако изложить алгоритм на формальном языке сразу бывает довольно сложно, требуется предварительная неформальная его запись в промежуточном, “черновом” варианте. В качестве такого промежуточного языка используется *язык блок-схем*. Надо отметить, что детальная запись сложного алгоритма на этом языке – дело трудоемкое, а результат, представляющий собой нечто вроде принципиальной схемы телевизора, не так уж и нагляден. Поэтому блок-схемы применяются в основном для изображения укрупненной общей схемы алгоритма, либо отдельных его фрагментов. Вот пример такого укрупнённого изображения алгоритма для нашей задачи:

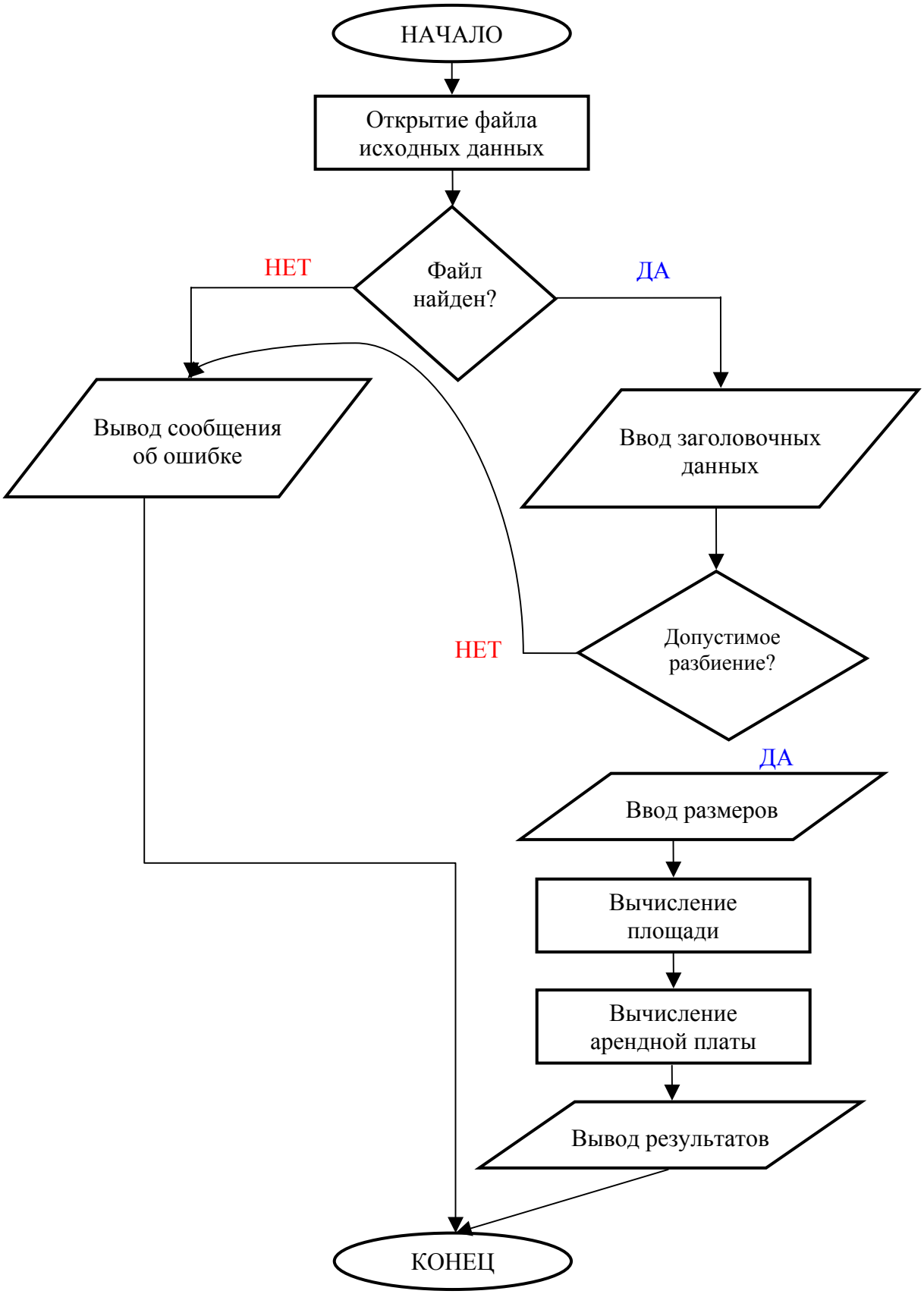


Рис. 1.7. Блок-схема алгоритма расчета арендной платы

Вообще, для представления алгоритма в литературе в настоящий момент чаще всего применяется следующая форма описания: словесное изложение с элементами того или иного языка программирования, как правило, Pascal. Мы в дальнейшем тоже будем пользоваться такой формой.

В заключение раздела, отметим, что разработка алгоритма, так же как построение модели и метода – это, конечно, творческий неформализуемый процесс. Вместе с тем, за время существования программирования как научной и технической дисциплины наработаны некоторые правила и рекомендации, облегчающие "тяжелую программистскую долю". Одним из таких фундаментальных приёмов является иерархическое разбиение сложной задачи на ряд подзадач. Современные системы программирования обеспечивают пошаговую, поэтапную реализацию алгоритма, как говорят, разработку "сверху-вниз", то есть от общего, укрупненного представления алгоритма ко все более детальному виду. Вопросы полномасштабного освещения подобных технологий выходят за рамки данной книги. В то же время, положенные в их основу технологии модульного и структурного программирования, вопросы конструирования новых типов данных, введение в объектно-ориентированное программирование будут нами рассмотрены.

Программа

Ну вот, наконец, мы добрались и до этапа составления *программы*, то есть записи *алгоритма* на языке, "понимаемом" компьютером. Разговор о языках мы снова отложим (до следующей главы), отметим только, что одним из таких языков является *язык программирования* Pascal. А сейчас в полном соответствии с высказыванием "Лучше один раз увидеть, чем сто раз услышать" давайте сразу посмотрим на текст программы, реализующей решение рассмотренной в предыдущих разделах задачи. Из него мы сможем составить первое представление о языке, с которым будем работать далее на протяжении всей книги. Для того чтобы облегчить процесс знакомства, мы сопроводили весь текст пояснениями, оформив их в виде комментариев, заключённых в фигурные скобки. Жирным шрифтом выделены так называемые ключевые слова языка, выражающие его основные синтаксические конструкции. Более подробно с этими и другими элементами языка мы познакомимся чуть позднее.

```
{ ========================================================================== }
{ Пример 1.1 }
{ Вычисление арендной платы за земельный участок }
Program  Rent; { программа Rent (заголовок) }

{$APPTYPE CONSOLE}

uses  Printers, SysUtils; { использует стандартные библиотеки }

{ декларативная часть программы – различные объявления }

{ константы }
const
  MaxSeg = 20; { максимальное число отрезков разбиения }
  Rate = 100;  { плата за 1 кв. метр в рублях          }

{ типы данных }
type
```

```
  Coord = array [0..MaxSeg] of Real; { вектор вещественных     }
                                     { координат из MaxSeg + 1 }
                                     { элементов               }

{ переменные и их типы }
var
  AreaData: Text; { текстовый файл на диске            }
  y, h: Coord;    { векторы длин оснований трапеций    }
                  { и длин отрезков разбиения          }
                  { (высот трапеций)                   }
  N, i: Integer;  { текущее число отрезков разбиения   }
                  { и переменная цикла (целые числа)   }
  Number,         { номер участка                      }
  Name: string;   { и имя владельца (текстовые строки) }
  Area,           { площадь и                          }
  Cost: Real;     { стоимость (вещественные числа)     }

{ начало исполняемой части программы }
begin
  AssignFile(AreaData, 'AREADATA.TXT'); { присвоение значения  }
                                        { переменной типа файл }
                                        { конкретного          }
                                        { имени файла на диске }
                                        { (связывание)         }

  {$I-} { директива компилятору: отключить автоматическую }
        { проверку результата операции ввода/вывода       }

  Reset(AreaData);  { поиск и открытие файла на чтение }
  if  IOResult <> 0 then { если файл не найден, то вывод на дисплей }
  begin
    Writeln('ОШИБКА 1: Не найден файл AREADATA.TXT');
    Halt                 { завершение работы }
  end;
  {$I+}

  { чтение с диска }
  Readln(AreaData, Number); { номер участка }
  Readln(AreaData, Name);   { имя владельца }
  Readln(AreaData, N);      { количество отрезков разбиения }

  if N > MaxSeg then { если количество отрезков больше максимально }
                     { допустимого, то вывод на дисплей           }
  begin
    Writeln('ОШИБКА 2: Недопустимо большое число отрезков');
    Halt             { завершение работы }
  end;

  { чтение с диска }
  for i := 0  to  N  do
    Read(AreaData, y[i]); { длины оснований трапеций }
  Readln(AreaData);       { перейти на следующую строку файла }

  for i := 1  to  N  do
    Read(AreaData, h[i]); { длины высот трапеций }
```

```
  { вычисление площади участка }
  {       "метод трапеций"       }
  Area := 0;
  for i := 1  to  N  do
    Area := Area + (y[i - 1] + y[i]) * h[i];
  Area := Area / 2;

  Cost := Area * Rate;    { арендная плата }

  { вывод на принтер }
  with Printer do
  begin
    BeginDoc;
    Canvas.TextOut(10, 10, 'Участок ' + Number);
    Canvas.TextOut(10, 210, 'Владелец ' + Name);
    Canvas.TextOut(10, 410, 'Площадь участка ' + FloatToStr(Area) +
                            ' кв. м');
    Canvas.TextOut(10, 610, 'Арендная плата ' + FloatToStr(Cost) + ' руб.');
    EndDoc;
  end;
  { Закрытие файла }
  CloseFile(AreaData);
  { конец программы }
end.
{ ========================================================================== }
```

Читатель, взявший на себя труд разобрать представленный текст программы, должен заметить, что собственно расчетная ее часть занимает всего лишь пять строк из общего текста. Может быть эта ситуация вызвана простотой решаемой нами задачи? На самом деле нет. Опыт показывает, что почти всегда “обслуживающая” часть программы превышает по размеру (иногда многократно), собственно содержательные преобразования исходных данных в требуемый результат. Тому много причин. Во-первых, любая программа должна не просто работать, а работать надежно, то есть проверять все потенциально опасные для правильной работы ситуации (например, отсутствие файла с исходными данными). Во-вторых, существенную часть любой программы обычно составляет реализация интерфейса с пользователем (в простом случае это организация ввода и вывода информации). В-третьих, о чем мы неоднократно будем упоминать дальше, правильно написанная программа обязательно должна обладать тем, что на профессиональном языке называется “самодокументированность”, и что означает использование при оформлении текста программы общепринятых стилистических правил (отступы, именование переменных и т.п.). Есть и в-четвертых и в-пятых. Но об этом в свое время.

Итак, кажется, мы добрались до конца процесса – программа готова, можно пользоваться. В принципе, все верно, однако есть одно “но”. Задача, которую мы рассматривали на протяжении пяти разделов, достаточно проста, но даже для ее реализации нам понадобилась программа, содержащая почти сотню строк. А то ли еще будет. Программа среднего по нынешним меркам размера это десяток другой тысяч строк, что равносильно книге страниц на триста. Спросите себя, сколько раз вы ошибетесь просто при наборе ее текста? Перефразируя классика, можно сказать: “О, сколько нам *ошибок* чудных готовит просвещенья дух”[1]. Но не пугайтесь, на самом деле не все так плохо, просто вот так плавно мы переходим к следующему “наиболее горячо любимому” всеми без исключения программистами этапу – *отладке* программы.

[1] А.С. Пушкин: “О, сколько нам открытий чудных готовят просвещенья дух…”

В разработке программ имеются свои вопросы наподобие первенства курицы или яйца. Так, многие люди считают, что главное – правильно провести анализ предметной области, построить модель, разработать алгоритм, а уж программирование – кодирование алгоритма – дело десятое. Другие уверены, что именно кодирование есть самая главная часть работы. На самом деле, как это часто бывает правы и те и другие – все перечисленное одинаково важно, а в целом *промышленное программирование* – сложный технологический процесс со своими специальностями, технологиями, проблемами.

Конечно, вопрос успешной разработки программного продукта в существенной степени зависит от качества подбора персонала и бюджета. Известен так называемый закон Лермана: "*Имея достаточно времени и денег, можно преодолеть любую техническую проблему*". Но вместе с законом известно также и его следствие: "*Вам всегда будет не хватать либо времени, либо денег*". Отсюда вывод – необходимо не просто учиться программировать, а учиться делать это *быстро* и *качественно*.

Отладка

Ошибки, возникающие в процессе создания программы, помимо упомянутой выше причины могут быть вызваны и неадекватным моделированием, и некорректностью метода или алгоритма, и, наконец, неправильным применением самих средств программирования.

В целом типы ошибок принято разделять на два неравнозначных класса. Один из них – это класс *синтаксических ошибок*, то есть ошибок, связанных с неправильной записью или употреблением языковых конструкций. Эти ошибки легко исправимы, так как соответствующее программное обеспечение – транслятор языка – осуществляет автоматический контроль синтаксической правильности программы пользователя, а с помощью программы контекстно-зависимой помощи можно получить как разъяснение ошибки, так и узнать правильный вид языковой конструкции.

Другой вид ошибок, действительно представляющий проблему программирования, – *смысловые ошибки*. Обнаружение и исправление их, что собственно и представляет собой процесс *отладки*, дело сложное, а порой, как это ни парадоксально звучит, и безнадёжное. Как определить, что программа имеет смысловую ошибку? В лучшем случае программа не работает, то есть её работа прерывается в некоторый момент и система выдаёт какое-нибудь туманное сообщение типа "исчезновение порядка числа с плавающей точкой". В худшем случае программа успешно завершает свою работу и выдаёт результаты, отвечающие интуитивным представлениям о характере решения задачи, а о наличии ошибки в программе мы узнаём только после практического внедрения результатов, например, когда по нашим прочностным расчётам построили мост, а он тут же обвалился под собственной тяжестью.

Как обнаружить такие скрытые ошибки? Самый популярный метод – так называемое *тестирование*. Следует взять такие исходные данные, правильный результат расчёта для которых известен заранее, и выполнить программу с этими данными. Если полученный результат совпадает с известным, то, как говорят, "тест прошёл". Беда в том, что, это совсем не означает, что программа не содержит ошибок. Рассмотрим простой пример. Допустим, нас попросили реализовать программу умножения двух вещественных чисел, и мы предложили следующий вариант на языке Pascal:

```
{ ========================================================================= }
{ Пример 1.2 }
program Mult;                          { заголовок                        }

{$APPTYPE CONSOLE}

var                                    { объявление                       }
  a, b, c: Real;                       { вещественных переменных          }
begin
  Writeln('Введите сомножители');      { вывод запроса на дисплей         }
  Readln(a, b);                        { чтение с клавиатуры              }
  c := a + b;                          { ОШИБКА!!! + вместо *             }
  Writeln('Произведение равно ', c);   { вывод результата на дисплей      }
end.                                   { конец программы                  }
{ ========================================================================= }
```

Чтобы доказать заказчику правильность предложенной программы, мы предлагаем ему тест: **2 x 2 = 4**, который, очевидно, проходит. Тем не менее, программа содержит существенную ошибку. Можно, конечно, сказать, что надо провести несколько тестов, но это означает, что мы должны знать все особые случаи. Для сложного алгоритма такая информация, как правило, неизвестна.

Ситуация усугубляется тем, что часто получение тестовых данных выливается в самостоятельную программистскую работу. Возьмём нашу задачу о земельных участках. Как получить значение площади участка с криволинейной стороной, если именно ради этого мы и составляли программу? Единственный выход – имитация. Следует “придумать” участок с криволинейной стороной, вид которой задаётся некоторой формулой, и вычислить его площадь методом аналитического интегрирования. Затем выбрать отрезки разбиения, рассчитать длины оснований трапеций и, применив нашу программу, сравнить результаты. Однако ручной расчет длин оснований трапеций для нескольких вариантов теста вполне может оказаться задачей трудоёмкой, следовательно, надо написать программу. Но её тоже надо отладить! Замкнутый круг! Наконец, никто не может гарантировать, что мы не ошибемся и при аналитическом интегрировании.

Проблема получения тестовых данных настолько серьёзна, что иногда сдерживает разработку сложных систем. Например, один из аргументов противников разработки американской системы противоракетной обороны космического базирования (СОИ) состоял в том, что проверить правильность работы сложнейшей компьютерной системы управления крайне трудно. Реальный тест – запустить все ракеты потенциального противника и сбить положенное количество боеголовок – просто невозможен.

На практике разработчики сложных программных систем следующим образом решают проблему, связанную с заранее очевидной “недотестированностью” программ. Программы продаются (сдаются в эксплуатацию), а разработчики берут на себя обязательства по так называемому *сопровождению* программного продукта. Другими словами, разработчики обязуются при выявлении кем-нибудь из пользователей ошибок вносить необходимые исправления и предоставлять потребителям обновлённые версии программы.

Принципиально другой подход к выяснению корректности программы состоит в методе формального доказательства её правильности по типу доказательства теорем. Однако на пути практического применения этого подхода стоят большие трудности, и он не играет заметной роли в приложениях.

Итак, программа написана, предварительно отлажена и сдана в эксплуатацию. Неужели есть что-то дальше? Да, есть. Причем, чем лучше и серьезнее разработанная программная система, тем более долгая жизнь ждет ее в потенциале, а значит, тем длиннее будет это “дальше”.

Модификация

Крупные программные комплексы разрабатываются с расчетом на достаточно длительную эксплуатацию, измеряемую как минимум годами. В то же время динамичность компьютерной индустрии настолько велика, что даже пара лет является огромным сроком, в течение которого существенно меняется аппаратная база, требования к функциональности и качеству программных продуктов. А значит, разработанную программную систему придется модернизировать, добавлять в нее новые возможности, не предусмотренные первоначальной постановкой. Естественно этот процесс должен быть максимально быстрым и простым, что в свою очередь накладывает определенные довольно существенные требования на первоначальную реализацию. Она должна быть достаточно гибкой, чтобы необходимость ее модификации не приводила к полному перепрограммированию. В то же время в стремлении к универсальности нельзя заходить слишком далеко, иначе программная система станет чересчур громоздкой, а ее внутреннее устройство (как говорят, “архитектура”) запутанным и излишне сложным. В общем, необходим разумный компромисс между стремлением как можно раньше создать первую полнофункциональную версию и желанием облегчить процесс дальнейшей ее модификации. Найти этот компромисс – весьма непростая задача.

Различные приемы, помогающие в достижении указанного компромисса, мы явно или косвенно будем рассматривать далее в книге, а пока отметим в качестве иллюстрации один достаточно простой момент. В примере 1.1 параметр модели ***Rate*** мы представили как константу с некоторым заданным значением. Решение это основано на вполне разумном предположении, что плата за кв. метр изменяется достаточно редко, и пользователю программы будет не слишком удобно вводить одно и то же число при каждом ее запуске. В то же время, если плата все-таки изменится, без участия программиста и наличия исходного текста программы обойтись не удастся. Как свести воедино эти противоречащие друг другу (на первый взгляд) требования? Решение может состоять, например, в следующем. Параметр ***Rate*** представляется в программе как переменная. В начале программы ей автоматически присваивается некоторое значение. Затем программа пытается найти и прочитать файл инициализации, в котором может храниться измененное значение параметра. Если файл найден, то ***Rate*** устанавливается равным значению, считанному из файла. Теперь при изменении платы за кв. метр пользователь программы сможет внести новое значение в файл инициализации, и программа будет функционировать правильно.

Выводы

Завершая первую главу, попытаемся подвести некоторые итоги. Мы рассмотрели – достаточно подробно для начального ознакомления – основные этапы на пути от возникновения задачи, для решения которой необходима вычислительная техника, до ее воплощения в программный код, а также обсудили некоторые моменты дальнейшей жизни получившейся программы. Отметили существенную взаимосвязанность всех этапов и общий циклический характер процесса, когда с каждого из этапов может потребоваться вернуться к одному из предыдущих для уточнения

постановки задачи, адаптации модели, выбора более подходящего метода, формирования более эффективного алгоритма. Каждый из представленных этапов важен и занимает свое место в индустрии программирования. Вместе с тем, мы в нашей книге основное внимание сосредоточим на части общей технологической цепочки: “алгоритм – программа – отладка – модификация”.

2. Современная система разработки программного обеспечения

*Плохой работник ненавидит **свои** инструменты. Хороший работник ненавидит **плохие** инструменты. Результаты труда специалиста в значительной степени определяются его **инструментами**.*

Джеральд Вейнберг

В предыдущей главе мы познакомились с тем, каким образом компьютер можно привлечь к решению задач, возникающих в различных областях человеческой деятельности. В процессе этого знакомства мы сформулировали некоторую схему, в соответствии с которой надлежит действовать: с чего начинать, через какие этапы двигаться к результату. Часть этих этапов, как было отмечено, содержат немалую долю творчества, однако в целом процесс применения вычислительной техники для решения реальных практических задач может быть до некоторой степени формализован.

У Вас не возникло никаких вопросов? Как минимум, а что мы собственно понимаем под *реальной практической задачей*? И чем эта задача отличается от “нереальной” или “непрактической”? К примеру, мы встретили школьных друзей и после бурного наплыва ностальгических воспоминаний о любимой школе приняли решение реализовать программную систему для нахождения корней квадратного уравнения, взяв за основу мощный математический аппарат, изученный нами много лет назад. Будет ли это в точности то, что мы здесь и далее понимаем под реальной практической задачей?

Конечно, нет. По счастью, в России все еще есть довольно много людей, решающих квадратные уравнения если не в уме, то, по крайней мере, при помощи ручки и бумаги и без существенных временных затрат. А, следовательно, вряд ли наша программная система будет востребована. Никто не станет ее использовать ни в коммерческих, ни в исследовательских, ни в учебных целях. Почему это так? Вспомним, откуда появилась идея о написании программы? Вспомнили? Правильно! Основная предпосылка была: “давайте что-нибудь напишем”.

Серьезные программы не пишут просто так. Обычно, до момента начала разработки имеется довольно точное представление, зачем нужна эта программа, и кто ей будет пользоваться. Таким образом, идея о написании программы не возникает сама по себе, ее порождает некоторая *задача*, для которой заранее очевидны выгоды от использования компьютера. Так, вряд ли Вы захотите вручную обсчитывать весь комплекс задач бухгалтерии крупного предприятия или вычислять орбиту, по которой должен двигаться спутник. Вот такие задачи можно смело отнести к *реальным* и уж безусловно к *практическим*. Итак, запомним: *сначала реальная задача, в которой требуется привлечение компьютера, потом программа*.

О средствах разработки

Взглянем под другим углом зрения на рассмотренную в предыдущей главе схему. Весьма важный вопрос: “Каких специалистов необходимо привлекать для успешного решения проблем, возникающих на разных этапах этой схемы?” Иначе говоря, кто будет анализировать предметную область, кто строить модель, кто создавать алгоритм и т.д.?

Необходимо отметить, что в настоящий момент разработка программного обеспечения фактически является *технологическим процессом*, на разных стадиях которого действуют подготовленные специалисты, применяющие в своей повседневной производственной практике различные технологии. Среди таких специалистов выделяются аналитики, маркетологи, менеджеры, кодировщики, тестеры, специалисты по созданию документации и многие другие.

Отрасль, связанная с разработкой программного обеспечения, в которой работают авторы данной книги, а также, возможно, собираются работать ее читатели, очень молода. Действительно, ей немногим более полувека, однако накопленный в ней объем знаний, технологических решений, методик и просто практических рекомендаций к действию воистину огромен! Рассмотреть весь процесс производства программного обеспечения в рамках одной книги невозможно, и мы не собираемся этого делать. В данной книге мы предлагаем нашим читателям познакомиться с тем, как пишется программный код, на каких принципах основан этот процесс, какие технологии применяются, в чем их содержательный смысл.

Вопросов, которые нам потребуется рассмотреть, достаточно много. С чего же начать? Остановитесь на секунду и подумайте, с чего бы начали Вы? Пока не знаете? Тогда давайте представим себе ситуацию: нас страшно заинтересовала задача нахождения среднего арифметического двух чисел. Наши действия: находим алгоритм ее решения, осознаем всю глубину заложенного в алгоритме математического аппарата, с торжествующим возгласом мчимся к компьютеру и… А что дальше? Начинаем писать программу? А как или, как говорят программисты, на чем (имея на самом деле ввиду, на каком языке)? Оказывается, прежде нам понадобится решить для себя, *на каком языке программирования*[1] *мы будем писать программу*. Решение это в реальной ситуации зависит от многих факторов, в нашем же случае ответ содержится на обложке книги. В рамках данной книги изучается язык программирования Pascal (или Object Pascal, что точнее, но далее для краткости первое слово мы будем опускать).

Хорошо, язык “общения” с компьютером мы выбрали[2]. Догадались, какой вопрос следующий? А где писать текст программы? Быть может великолепный текстовый редактор Microsoft Word, в котором Вы привыкли создавать такие красивые открытки для своих друзей и такие интересные рефераты по разным областям знания (как говорится: “Блажен, кто верует”) подходит для написания текста программы? Может быть и так, но профессиональные программисты почему-то программы в нем не пишут. Для этого используются… Впрочем, немного подождите.

Еще один вопрос: “А как превратить текст программы в нечто, что можно выполнить?” Здесь ответ уже далеко не тривиален, но, немного терпения, и мы преодолеем и это препятствие.

---

[1] расшифровка этого важного понятия будет приведена чуть ниже.

[2] на самом деле необходимость такого выбора должна вызывать у вдумчивого читателя вопрос: “А зачем?”. Разве не достаточно было создать один единственный язык для “общения” человека с компьютером? Ответ на этот вопрос мы немного отложим.

Перечень вопросов можно продолжить и, прочитав книгу до конца, Вы обнаружите их еще немало, однако пока нужно остановиться. Сказанного должно быть достаточно, чтобы подвести Вас к важной мысли: *для поддержки деятельности программиста необходимы специальные средства*[1].

> Эта мысль на самом деле важна. Будет ли токарь работать без станка? Будут ли на лесопилке пилить дерево лобзиком или старинной двуручной пилой? Будут ли для переноса бетонных свай на стройке использовать десятки тысяч человек, как в Древнем Египте? Ответы очевидны. В любой отрасли существуют свои средства, упрощающие труд специалиста. Есть такие средства и в разработке программного обеспечения. Средства эти от года к году совершенствуются, причем, процесс их развития, как и всего остального в программной индустрии, необычайно стремителен.

В данной главе мы рассмотрим, какие именно средства помогают программистам в решении их профессиональных задач. При этом основное внимание будет уделено не конкретным образцам, а их классам (так, мы рассмотрим, что такое компилятор вообще, а не конкретный компилятор Borland Pascal 7.0 Compiler). Готовы? Тогда за дело.

Основные средства разработки

Прежде всего, поговорим о тех средствах создания программ, без которых этот процесс вообще невозможен (точнее говоря практически невозможен) в настоящий момент времени.

Язык программирования высокого уровня

Что значит написать программу? Написать программу – *реализовать* алгоритм, иначе говоря, представить его в виде понятных компьютеру указаний того, что необходимо делать. К сожалению, компьютеры не умеют понимать человека с полуслова, а это значит, что словесное описание алгоритма необходимо превратить в абсолютно точный набор инструкций, однозначно интерпретируемых машиной.

> Представьте себе, что Ваш друг никогда не слышал про метод половинного деления. Представьте теперь, что Вам жизненно необходимо довести этот метод до его сведения. Какие средства для этого имеются в Вашем распоряжении?
>
> Ну конечно, прежде всего, это великий и могучий русский язык. Вы просто объясняете другу суть метода, пользуясь некой математической терминологией и обиходными словами русского языка. Надеемся, что Вы хорошо изучили русский язык. Надеемся, что Вы в состоянии не только изъясняться на бытовом уровне, но и грамотно писать связный текст, рассказывать о чем-то и т.д. и т.п. Люди, считающие, что они освоили Microsoft Visual Studio .Net, C++, C#, Object Pascal, Visual Java, Microsoft SQL Server, Windows, Linux и прочие умные слова, но, в то же время, не способные изъясняться на родном языке так, чтобы их понимали хотя бы коллеги, вряд ли смогут найти хорошую работу. А что, если друг англичанин (в наше время развития Интернет-технологий этот вариант

[1] средства в данном случае имеются в виду программные, понятно что без компьютера как такового все остальное вообще бессмысленно.

более не представляется экзотическим)? Надеемся, что Вы активно изучаете также и английский язык, в противном случае Вас ожидают большие трудности в изучении технической документации, чтении электронной справки и т.д. К сожалению, большая часть информации, необходимой программистам, присутствует именно на английском языке. Если Вы более или менее владеете языком, то для Вас не составит особого труда рассказать по-английски, как работает метод.

Итак, основное средство передачи информации от одного человека к другому – некоторый понятный обоим язык общения.

А как объяснить что-либо машине? Увы, пока что для компьютера и русский, и английский, и суахили – одинаковая тарабарщина[1]. Поэтому, для того чтобы иметь возможность "общаться" с компьютером, люди создали специальные языки. Классификация этих языков и история их возникновения подробно описаны в литературе [5]. Мы же здесь лишь отметим, что процесс написания программ за последние полвека прошел путь от программирования в инструкциях процессора (в машинных кодах) через программирование на низкоуровневых языках (ассемблер) до программирования на *языках высокого уровня*. Вот на них мы и остановимся чуть подробнее.

Что такое язык программирования высокого уровня? Чем он отличается от естественного языка?

С формальной точки зрения *Язык программирования = Синтаксис + Семантика*.

Обратимся к литературе и посмотрим, как расшифровываются понятия *синтаксиса* и *семантики* для естественного языка:

- *Синтаксис* – раздел грамматики, изучающий внутреннюю структуру и общие свойства предложения [3].
- *Семантика* – раздел языкознания, изучающий значения единиц языка [3].

Для языков программирования справедливы следующие определения:

- *Синтаксис* – набор правил построения фраз алгоритмического языка, позволяющий определить осмысленные предложения в этом языке [18].
- *Семантика* – система правил истолкования отдельных языковых конструкций. Семантика определяет смысловое значение предложений алгоритмического языка [18].

Заметим, что определения достаточно похожи по своему смыслу. Действительно, язык программирования – искусственный (формальный) язык, предназначенный для записи алгоритмов [18]. Язык программирования задается своим *описанием* и реализуется в виде специальной программы: *компилятора* или *интерпретатора* [18].

Таким образом, если обычные (естественные) языки предназначены для общения людей между собой, то языки программирования – для общения программиста с компьютером.

Что же означает словосочетание "высокого уровня"? Чем языки программирования высокого уровня отличаются от языков "низкого уровня"? Быть может, кто-то всерьез считает, что языки "высокого уровня" – хорошие, а "низкого уровня" – плохие, неразвитые… Разумеется, это заблуждение. Говоря об уровнях, мы ведем речь, прежде всего, о степени приближенности языка к машине. Уровень в данном случае – *уровень машинного восприятия*. Так, языки

[1] На заре развития вычислительной техники бытовало мнение, что через несколько лет компьютеры научатся понимать человеческий язык. Современные промышленные гиганты, такие как Microsoft, вкладывают большие средства в исследования в области распознавания речи и голосового управления, однако, до полноценного диалога или, даже, до полноценного голосового управления еще очень далеко.

низкого уровня (ассемблер) по возможности приближены к машине, что делает соответствующие программы особенно эффективными с точки зрения их быстродействия. Однако существенная проблема использования таких языков заключается в том, что программист – прежде всего человек, и его способы восприятия информации весьма далеки от машинных, что чрезвычайно затрудняет написание программ на ассемблере. В настоящий момент на ассемблере реализуются сравнительно небольшие участки программного кода, связанные преимущественно с программированием аппаратуры (например, драйверы устройств). Подавляющее большинство программ пишется на том или ином языке программирования высокого уровня. Такие программы существенно ближе к восприятию человека, наделенного некоторыми профессиональными навыками – программиста. Следующая таблица иллюстрирует некоторые достоинства и недостатки языков программирования низкого и высокого уровня.

Свойство	Программа на языке программирования низкого уровня	Программа на языке программирования высокого уровня
Легкость создания	–	+
Понятность текста при беглом просмотре	–	+
Удобство отладки (поиска и исправления ошибок)	–	+
Удобство модификации	–	+
Удобство коллективной разработки	–	+
Быстродействие	+	–

К настоящему времени создано большое число разных языков программирования высокого уровня, однако реально используются лишь некоторые из них. К числу активно применяемых языков относятся C и C++, Pascal и Object Pascal, Fortran, Java, Basic (Visual Basic). В данной книге рассматривается язык программирования высокого уровня *Object Pascal*.

Транслятор. Интерпретатор. Компилятор

Под *транслятором* (*translator*) обычно понимают специальную программу, которая переводит текст программы в последовательность машинных команд. Напомним еще раз: текст программы понятен человеку, набор команд понятен компьютеру.

Заметим, что трансляторы языков высокого уровня, таких как Pascal, C, Fortran и других, обычно называют *компиляторами* (*compiler*). Этим подчёркивается общепринятый для промышленных языков режим трансляции, при котором в начале осуществляется перевод программы в двоичное представление, а лишь затем программа передаётся на исполнение. Другой способ трансляции, называемый *интерпретация*, состоит в совмещении перевода и исполнения программы (в этом случае исполняемый модуль не сохраняется и его, соответственно, нельзя повторно использовать). Метод интерпретации используется при выполнении программ на языке Basic.

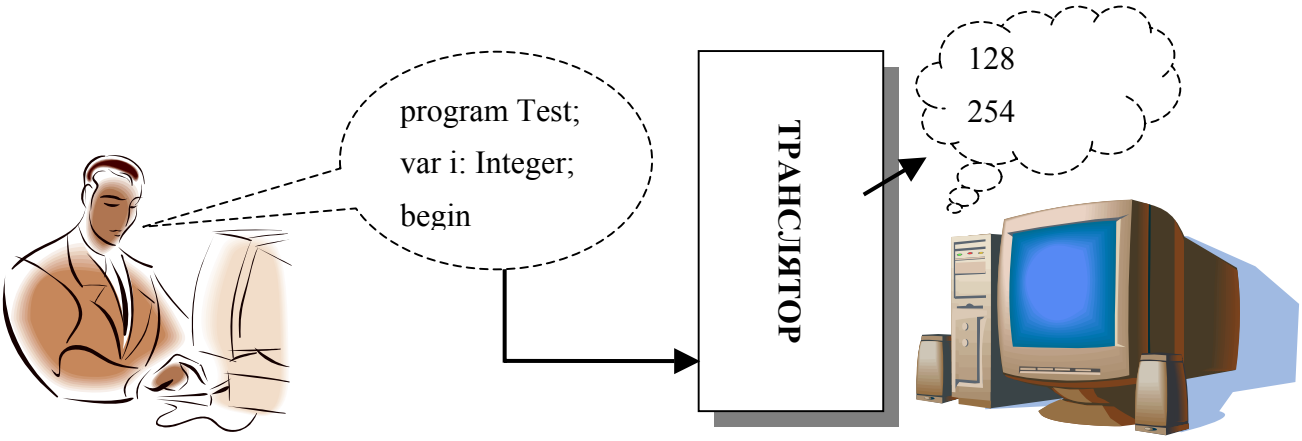


Рис. 2.1. Роль транслятора в создании программ

К числу основных достоинств компилируемых языков по сравнению с интерпретируемыми относятся:

- в компилируемых языках процесс построения (создания) исполняемого модуля выполняется один раз, а не при каждом запуске, что экономит время.
- в компилируемых языках обнаружение синтаксических ошибок происходит до запуска программы на выполнение, а не в его процессе.

Несмотря на очевидные недостатки интерпретируемых языков, они применяются в разных специфических задачах, а также в тех случаях, где простота программы важнее ее производительности (а программы на интерпретируемых языках почти всегда проще[1] своих аналогов на языках транслируемых).

Редактор связей

Уже в самом начале развития методов программирования стал применяться простой и эффективный приём выделения часто используемых алгоритмов в самостоятельные программы, получивших название стандартных *подпрограмм*. Примером могут служить подпрограммы вычисления элементарных функций (синус, косинус и др.), а также процедуры обмена с внешними устройствами компьютера. Однажды составленные и откомпилированные, они в дальнейшем могут применяться программистами в своих задачах путём подсоединения их к разработанному коду основного алгоритма. В более широком плане эта идея нашла своё выражение в технологии *модульного программирования*, которую мы будем рассматривать в нашей книге. В данный момент для нас важно обратить внимание на тот факт, что для обеспечения комплектации оттранслированной программы вспомогательными подпрограммами требуются специальные средства.

В систему программирования входит программа, называемая *редактор связей* (сборщик, динамический компоновщик), которая обеспечивает поиск вспомогательных подпрограмм в специальных библиотеках программ и их присоединение к основной программе пользователя. Результатом работы редактора связей является полностью готовый к исполнению двоичный код программы, называемый *загрузочным модулем*.

[1] простота здесь означает не то, что программа меньше “умеет”, а то, что она легче воспринимается (говорят “читается”) человеком.

Загрузочный модуль может быть немедленно инициирован на исполнение, а может быть записан на диск и в дальнейшем многократно вызываться на исполнение с помощью специальной программы – загрузчика.

Отладчик

В систему программирования входит также программа, облегчающая *отладку*, а точнее, *поиск ошибок*. При всём многообразии реализаций отладчиков их основные возможности заключаются в так называемой *трассировке* работы программы.

Трассировка – это отслеживание (ведение протокола) работы программы. В процессе трассировки программист может проследить порядок исполнения операторов, а также динамику изменения значений переменных программы.

В современных условиях отладка программ является не менее, а зачастую и более важным этапом разработки, чем собственно программирование (написание кода). Реальные задачи, пришедшие из разных областей человеческой деятельности, как правило, являются очень сложными. Объем программ, реализующих их решение весьма велик. Такой код обычно создается большим коллективом разработчиков, в связи с чем возникает много дополнительных проблем (в частности, необходимость поддерживать передачу информации между разными участниками проекта и согласовывать их деятельность). В конечном счете, сложность задач приводит к росту числа ошибок в программе. Известна старая шутка о том, что “любая программа содержит хотя бы одну ошибку”. Известно продолжение этой шутки: “если ошибок в программе не обнаружено, ищи ошибку в компиляторе”. В обоих положениях есть большая доля правды. На этапе разработки крайне редко удается полностью промоделировать реальную обстановку, в которой будет функционировать программа (представьте, например, как Вы создаете у себя тестовый полигон, размером с автомобильный завод), а также отследить все могущие возникнуть ситуации.

Все это, конечно, хорошо, но как же все-таки определить, есть в программе ошибка или нет? Кажется, что самый простой способ можно сформулировать так: “сейчас запустим и проверим!”. К сожалению, этот принцип применим лишь для очень ограниченного спектра задач. Конечно, если Вы создаете программную систему, которая должна нарисовать на экране тигра, Вы легко можете проверить ее работоспособность. Запускаете, смотрите – “да это же не тигр, это вовсе кролик!” Значит, вывод элементарен – не работает. Оставим на минуту вопрос о том, как теперь получить ответ на вопрос, почему не работает. Представим себе, что Вы пишете программу для расчета траектории движения спутника, который будет запущен в космос. Представили? “Сосчитали, попробуем запустить спутник. Ой, какой кошмар! Спутник улетел в неизвестном направлении…” Теперь поняли? Итак, необходимо уметь проверять работоспособность программы до внедрения ее в эксплуатацию. Если вдобавок к этому вернуться к вопросу о поиске причины возникновения обнаруженной ошибки, мы поймем, что диагностика и исправление ошибок в программах – важнейшая задача. В реальных программистских компаниях существует целый штат сотрудников, которые занимаются этими проблемами (тестеры, контролеры качества,…). Более того, теория в данной области не стоит на месте. Разработано несколько разных подходов к решению рассмотренных проблем. В нашей книге мы не будем подробно останавливаться на этих подходах, это предмет для серьезного разговора, заслуживающий отдельной книги.

Редактор кода

Мы начали рассмотрение системы программирования с её ключевых частей – транслятора и редактора связей, которые существовали с самого начала развития систем автоматизации программирования. А вот процесс составления программ долгое время оставался ручным. Программист записывал программу на специальном бланке, относил в отдел перфорации, где операторы с помощью специального оборудования наносили программу на перфокарты или перфоленты. С них программа загружалась в ЭВМ, запускалась, в ней обнаруживались ошибки, программист их исправлял, снова "набивал" перфокарты и так далее.

В настоящее время – время персональных компьютеров – этот рутинный процесс ушёл в прошлое. Современный программист, как правило, не пользуется бумагой для записи программ, а сразу заносит её текст в компьютер "из головы", пользуясь так называемыми *редакторами кода* (редакторами текстов) или *текстовыми процессорами*.

Редактор кода – это программная система, обеспечивающая первоначальную подготовку исходного текста программы и его исправление в процессе разработки. В отличие от универсальных текстовых процессоров (самым известным из них является уже упоминавшийся выше Microsoft Word) редакторы кода специализированы для работы именно с исходными текстами программ, поэтому они не имеют массы функций обычных редакторов (вроде работы с таблицами и рисунками), зато предоставляют другие функции не менее полезные. Существует довольно большое количество различных редакторов кода, список их возможностей также весьма обширен, начиная от простого набора текста, комбинирования отдельных фрагментов, поиска по образцу, выделения цветом различных элементов программы и заканчивая автоматическим форматированием в соответствии с устоявшимися правилами оформления кода для того или иного языка программирования (эти правила часто называют *стилем*).

Подготовленная с помощью редактора текстов программа запоминается в виде одного или нескольких файлов и в дальнейшем служит входной информацией для транслятора.

Дополнительные средства разработки

В предыдущем разделе были рассмотрены компоненты системы программирования, без которых невозможно обойтись. Здесь мы поговорим о некоторых других средствах, использование которые не является чем-то обязательным, но, в то же время, способно упростить процесс создания программ, сделать его более эффективным по разным критериям.

Средства автоматизированной генерации кода

Помните, как, сидя в школе на уроке, Вы мечтали о том, чтобы какой-нибудь джин прилетел и сделал за Вас такую нудную контрольную работу? Или написал сочинение? Или нарисовал изометрическую проекцию? Или сделал что-нибудь еще, предоставив Вам возможность просто посмотреть в окно, помечтать…

Вы будете удивлены, но подобные мысли неоднократно приходили в голову авторам этой книги. Значит ли это, что мы лодыри, лентяи и тунеядцы? Надеемся, что нет. Просто, периодически бывают случаи, когда очевидно, что именно нужно сделать, но эта работа является скучной,

неинтересной, особенно грустно, если подобная работа до того уже выполнялась много раз. Что же делать?

К счастью, в разработке программ существует возможность частично избавить себя от рутины с помощью так называемых *средств автоматизированной генерации кода*, в некоторых случаях они умеют самостоятельно создавать программный код, выполняющий определенные стандартные действия. Примером таких средств может служить, например, Delphi IDE, которая автоматически создает и заполняет полями класс “Форма” при создании нами нового окна и наполнении его различными компонентами. Если в предыдущем предложении Вы поняли ровно половину, не беда. Постепенно мы со всем разберемся, сейчас же важно усвоить следующие: в некоторых случаях современные системы программирования способны автоматически создавать фрагменты текста программы.

Оптимизирующий компилятор

Все люди любят, когда программы работают быстро. В то же время, далеко не все задаются вопросом, как этого добиться.

Прежде всего, для решения одной и той же задачи часто можно предложить несколько алгоритмов, отличающихся по эффективности (как именно ее оценивать мы узнаем чуть позже). Далее, для одного и того же алгоритма могут быть созданы несколько реализаций, также различной эффективности.

Известно, что для обеспечения максимальной производительности имеет смысл писать программы на Ассемблере. Однако, этот подход приемлем только для небольших задач (в больших написать на Ассемблере всю программу просто не удастся). Итак, мы пишем программу на языке программирования высокого уровня. Допустим, у нас есть неплохой алгоритм. От чего теперь будет зависеть качество программного кода? Только ли от нас? Конечно, нет. Качество получаемого кода весьма сильно зависит от компилятора, ибо именно он преобразует текст программы в набор машинных команд. От того, как он это сделает, напрямую зависит, как быстро будет работать наша программа.

Многие современные компиляторы генерируют достаточно эффективный код. Именно к этой категории относится компилятор с языка Object Pascal от фирмы Borland. Однако стоит заметить, что этот компилятор ничего не знает о наличии самых современных процессоров (последние процессоры линейки Pentium), и, следовательно, о новых командах, появившихся в этих процессорах. Значит ли это, что программа не будет работать на таких процессорах? Кончено, нет. Обратную совместимость еще никто не отменял. Новая аппаратура, как правило, работает со старыми средствами, просто работа эта недостаточно эффективна. Немногие компиляторы можно причислить к числу истинно оптимизирующих. Основные представители данного семейства – компиляторы языков C++ и Fortran от фирмы Intel. В отдельных расчетных задачах эти компиляторы (при использовании современной аппаратуры) позволяют получить выигрыш в производительности на порядок и больше.

К дополнительным функциям оптимизирующего компилятора относятся исключение неиспользуемых участков кода из исполняемого модуля, замена неэффективных конструкций более быстрыми и т.д.

Профилировщик

Представьте себе, что Вы в сотрудничестве с другими программистами создали некую программную систему. После установки ее на реальном объекте (завод, магазин, склад) выяснилось, что при увеличении объемов обрабатываемых данных Ваша программа работает слишком медленно. Вопрос: как узнать, почему это происходит. Где в коде те “узкие места”, на которые приходится основное время выполнения? Как повысить быстродействие?

Для того, чтобы получить ответ на этот вопрос, существуют специальные программы, называемые *профилировщиками*. Одним из лучших профилировщиков на настоящий момент является Intel® VTune™ Performance Analyzer. При помощи этой и других подобных программ Вы можете серьезно повысить производительность Вашей программы, внеся в нее необходимые изменения. Процесс использования профилировщиков описан, например, в [6].

Средства документирования

Осознаете ли Вы необходимость создания документации к разрабатываемой программной системе? Считаете ли Вы, что каждый программист должен уметь создавать документацию и делать это параллельно с разработкой программы? Если сейчас Вы на оба вопроса ответили “Нет”, отнесем это на счет Вашей неопытности. Однако если Вы не измените своего мнения в будущем, найти работу по нашей специальности Вам вряд ли удастся.

Необходимость в создании документации неизбежно возникает в любом технологическом процессе. Процесс разработки программного обеспечения не является исключением. На всех этапах этого процесса создается масса документов различной направленности. Это документы управленческие (инструкции, приказы), юридические (лицензии), постановочные (техническое задание, требования к системе), проектные (проект программного комплекса), описательные, предназначенные для конечного пользователя (руководство пользователя), и, наконец, внутренняя документация, описывающая, как создавалась система, какова ее архитектура, какие модули в ней есть, что в них находится и т.д. и т.п.

На первых порах Вам может показаться, что такое обилие документации есть следствие неизбежной тяги человека к бюрократии. На самом деле это не так. Вы, разумеется, не верите и скептически качаете головой. Попробуем Вас разубедить! Для этого приведем пару примеров.

Зададимся для начала одним, казалось бы, несложным вопросом: много ли программных продуктов Российского производства Вы знаете? Скорее всего, не очень. В чем же дело? Быть может, в России плохие программисты? Очевидно, это не так – наши студенты побеждают в международных олимпиадах по программированию, наши программисты находят применение своим силам во многих западных компаниях, более того, эти компании с большим удовольствием принимают их на работу. В чем же дело? Не претендуя на подробный анализ, укажем лишь одну причину (не самую главную, разумеется). Одним из факторов успешного коммерческого распространения чего-либо (в том числе и созданного программного обеспечения) является наличие хорошей инструкции. Хорошая инструкция – это не то, что напечатано 8-м шрифтом на папиросной бумаге. Это не то, что может понять только автор инструкции. Это не книга в 2000 страниц, глядя на которую будущему пользователю хочется побыстрее забыть про Ваш программный продукт. Хорошая инструкция – это то, при помощи чего можно быстро и самостоятельно научиться пользоваться приобретенной продукцией (в том числе и программным обеспечением). Как обстоит дело с написанием этих инструкций? Как правило, программисты

любят писать программы, но не любят писать инструкции. Специально приглашаемые для этого люди неизбежно сталкиваются с проблемой: сначала кто-то должен обучить их самих пользоваться программой, досконально осветив все аспекты ее применения. Получается замкнутый круг: для написания инструкции нужен технический писатель, которому для работы также нужна инструкция. К сожалению, этому обычно уделяется слишком мало внимания. Дополнительные трудности вызывает процесс перевода написанной инструкции (руководства пользователя) на разные иностранные языки (эффект "сломанного телефона" – один писал программу, другой – инструкцию, третий ее переводил).

Второй пример связан с созданием документации во время разработки программы (описания того, как устроена эта программа "изнутри"). Представьте себе, что у Вас есть коллектив программистов, который пишет программу, но не создает документацию. Представьте себе, что один из программистов уволился, а на его место был вынужденно принят на работу новый сотрудник. Как теперь он сможет разобраться, как устроено то, что ему надлежит доделывать и, возможно, в чем-то переделывать? Без наличия документации попытка разобраться в чужой программе во многом равносильна ее переписыванию с нуля.

Таким образом, создание документации – задача не менее важная, чем создание программного кода, и в ее решении нам помогают различные программные средства. Для примера сошлемся на одно из таких средств – "Система генерации проектной документации Rational SoDA" [15].

Понятие интегрированной среды разработки

Интегрированная среда разработки (IDE, integrated development environment) – специальная программа, предоставляющая возможность удобной совместной работы с различными компонентами системы программирования.

В предыдущих разделах мы рассмотрели большое количество видов программ, входящих в систему программирования. Это и редакторы кода, и компиляторы, и сборщики, и отладчики, и многие другие. При первом же знакомстве со всеми этими программами становится понятно, что каждая из них может работать с разными начальными установками. Так, например, Вы можете настроить множество параметров для редактора кода: цвет фона, цвет шрифта, шрифт, размер символа табуляции и еще сотню разных характеристик. Для компилятора Вы можете указать, как Вы предпочитаете оптимизировать код: по скорости, по размеру, никак не оптимизировать, а также есть возможность управления многими другими параметрами. Аналогично обстоит дело практически со всеми составляющими системы программирования.

Теперь представьте себе, как Вы по очереди запускаете все эти программы с огромным количеством разных параметров в командной строке. Т.е. сначала Вы запускаете редактор кода, и пишите в нем программу. После этого Вы ее сохраняете, а затем закрываете редактор. Далее Вы запускаете компилятор, указав ему в командной строке файл с текстом программы и все необходимые настройки. Компилятор отработал и нашел 4 ошибки в строках 27, 31, 110 и 547. Вы снова запускаете текстовый редактор, открываете текст программы и в результате титанических усилий находите эти строки и ошибки в них. После этого Вы снова сохраняете программу, закрываете редактор и опять запускаете компилятор. В результате компилятор создает нечто (объектный код), на этот раз по счастью без синтаксических ошибок (это Вам повезло!). Теперь Вы запускаете сборщик, указывая ему в командной строке кучу параметров и тот самый объектный файл. Если Вам опять повезло и ошибок нет (это бывает отнюдь не всегда), то Вы, наконец-то, получаете исполняемый файл, запускаете его и… О ужас! Программа

запустилась и “повисла”. Или не повисла, но вместо тигра нарисовала Вам слона. Или ничего не нарисовала, но сказала, что Ваше уравнение не имеет корней, хотя Ваши друзья математики с пеной у рта не далее чем вчера доказывали Вам, что решение точно есть! Что это значит? Это значит, что с семантикой что-то не то, иначе говоря, программа работает неправильно и ее необходимо отлаживать, искать ошибки. А как это делать? Ага, для этого у нас есть отладчик. Все закрываем, запускаем отладчик, передавая ему в командной строке кучу параметров, исходный файл (текст программы), исполняемый файл, отладчик запускается и… О ужас! Он говорит Вам, что Вы при компиляции и сборке не включили так называемую *отладочную информацию*, оптимизировав исполняемый файл по размеру, а значит, не сможете нормально его отлаживать. Что делать? Снова запускаем компилятор, сборщик, потом отладчик. И т.д. и т.п.

Вам понравился процесс? Авторы книги застали момент, когда процесс именно так и выглядел. Поверьте, это очень неудобно. Для устранения неудобств и повышения эффективности процесса разработки создатели систем программирования стали строить их в виде так называемых *интегрированных сред*. Термин “интегрированная” в названии среды означает, что она включает в себя в качестве элементов все необходимые инструменты для выполнения полного цикла работ над программой: написания, компиляции, построения исполняемого модуля, запуска, отладки. Кроме того, интегрированные среды позволяют выполнять следующие операции:

- визуально (в диалоге) производить быструю настройку параметров каждого из компонентов системы программирования;
- сохранять разные системы настроек и загружать их по мере необходимости;
- нажатием нескольких клавиш или выбором соответствующих пунктов меню осуществлять запуск одного или сразу нескольких компонентов системы программирования, автоматизируя процесс передачи им необходимых параметров.

Так, в любой интегрированной среде исполняемый модуль из исходного текста программы можно получить нажатием пары кнопок на клавиатуре[1]. Единственный минус таких сред является прямым следствием их главного плюса – собрав “под одной крышей” большой набор инструментов, интегрированная среда сама становится весьма сложной программой. Однако время, потраченное на ее изучение с лихвой окупается в дальнейшем. И, наконец, еще один положительный момент – устройство большинства сред одинаково в концептуальном плане, различия наблюдаются лишь в комбинациях клавиш для того или иного действия да в названиях пунктов меню. Таким образом, освоив первую в своей жизни интегрированную среду, на все остальные Вы потратите в разы меньше времени.

Визуальные среды

Одно из последних достижений человеческой мысли в области разработки программного обеспечения – визуальные среды программирования (самые известные – Borland® Delphi™ с базовым языком Object Pascal и многоязыковая среда Microsoft® Visual Studio.NET). Их появление связано с двумя важными факторами. Первый уже упоминался выше в этой главе – стремление человека максимально автоматизировать собственный труд. Компьютеризация человеческой цивилизации становится всеобщей, компьютеры, “родившись” в стенах научных

[1] Конечно, не все так безоблачно. Дело ограничится парой кнопок только, если в программе не оказалось ошибок и если предварительно были сделаны необходимые настройки среды. Впрочем, справедливости ради надо отметить, что настроек по умолчанию обычно бывает достаточно.

учреждений, выбрались оттуда в промышленность, проникли в сферу обслуживания и, наконец, прочно завоевали себе место рядом с человеком в его собственном доме. Аппетит, как известно, приходит во время еды – вот и люди, получив в свое распоряжение такого помощника, требуют от него все больше и больше возможностей. Неизбежно растет число необходимых программ. Обеспечить такой объем потребностей можно только одним способом – стандартизировав производство. Визуальные среды – еще один шаг на этом пути. Они содержат в себе заготовки, из которых можно собирать работоспособный скелет программ, дополняя его впоследствии необходимой функциональностью.

Второй фактор связан с тем, что современный пользователь в большинстве своем не станет работать с программой, которая не удовлетворяет его “чувство прекрасного”. Говоря серьезно, сейчас при создании программ их внешнему виду уделяется не меньшее значение, чем внутреннему содержанию. Однако здесь разработчика поджидает серьезная дилемма. С одной стороны, чем больше новая программа визуально отличается от других, тем лучше – ее легче запомнить, она, что называется, “бросается в глаза”. А с другой, нужно быть очень осторожным – стоит чуть переборщить, и программа станет слишком сложной для восприятия. А талантливых дизайнеров среди программистов ничуть не больше, чем среди остальных профессий. Как же быть остальным “простым смертным”? Визуальные среды и тут приходят на помощь. На самом деле нетрудно понять, что внешний вид программы можно собрать (да-да именно собрать как в конструкторе) из некоторого количества стандартных элементов. И если Вы не чувствуете в себе таланта архитектора, то этот путь для Вас. Кстати, появление визуальных сред в середине 90-х годов прошлого века привело, в числе прочих эффектов, к взрывообразному росту числа программ, написанных программистами-одиночками. Некоторые из этих программ стали (вполне заслуженно) всемирно известными и используются массой людей.

Выводы

Вот и закончилась вторая глава, в которой мы познакомились с составом и функциональным назначением современных систем разработки программного обеспечения.

Мы узнали, что работа над текстом программы выполняется в редакторе кода, далее в дело вступают компилятор и редактор связей, перерабатывающие код в исполняемый модуль, состоящий из набора машинных команд. Запуская исполняемый модуль, можно исследовать его на наличие ошибок и обнаруживать их в программе при помощи отладчика. Кроме того, были рассмотрены дополнительные средства разработки, такие как средства автоматизированной генерации кода, оптимизирующий компилятор, профилировщик, средства документирования.

В завершающей части главы мы обсудили интегрированные среды разработки, объединяющие в себе указанные выше компоненты и делающие процесс написания и отладки программы максимально комфортным для программиста.

3. Среда исполнения программ. Программа в среде Microsoft Windows

Кому и командная строка – дружественный интерфейс.

Программистский фольклор

В предыдущих главах мы попытались разобраться с тем, зачем нам нужна вычислительная техника, что такое алгоритм, программа и чем они отличаются друг от друга, какими инструментами мы в настоящий момент располагаем для того, чтобы эти самые программы создавать. Также мы должны были осознать, почему техника без программ представляет собой лишь мертвую "груду железа", а программы без своего воплощения – более или менее строгую (чаще менее) абстракцию. Очевидный следующий шаг – начать изучать, как же собственно правильно превратить алгоритмическое решение конкретной задачи в текст программы на выбранном языке программирования. Однако к этому мы перейдем в следующей главе, а здесь рассмотрим, достаточно кратко, еще один весьма важный вопрос.

> Вообще говоря, в контексте обсуждения методов программирования словосочетание "*вычислительная техника*" требует расшифровки. Вроде бы очевидно, что к вычислительной технике относятся компьютеры. А что еще? Можно ли считать "вычислительной" стиральную машину с программным управлением? А цифровой фотоаппарат? А сотовый телефон? Ведь их назначение вовсе не в том, чтобы складывать и умножать числа. Однако это точка зрения потребителя. А для разработчика программного обеспечения, к каковым, мы надеемся, желает присоединиться и Читатель, важно лишь то, способна ли та или иная техника выполнять программы, поскольку, если способна, то кто-то должен для нее эти самые программы создавать. К счастью, при всем неизмеримом многообразии видов и моделей современной техники написание программ для нее основано на тех же базовых принципах, которые используются при работе с классическим "вычислителем", более знакомым всем под именем "компьютер". Итак, *с точки зрения программиста к вычислительной технике относится все, что имеет возможность выполнять программы*.

Что нужно для того, чтобы программа, которая, как мы уже должны были усвоить, есть выраженный на языке программирования алгоритм, могла быть выполнена? Вроде бы ответ очевиден – нужен тот, кто способен шаг за шагом (инструкцию за инструкцией) выполнять сформулированные в алгоритме действия. Поскольку действий много, нам потребуется место для их хранения и последующего считывания. Кроме того, любая программа оперирует данными (входными и результирующими) – их тоже необходимо хранить. Наконец, входные данные программе обычно поставляет человек, он же "забирает" результаты, а, значит, требуются средства ввода/вывода (обмена информацией).

Здесь, кстати, стоит упомянуть о глобальном противоречии, которое до сих пор определяет развитие всей программной индустрии – удобные способы представления информации у

человека и у компьютера, мягко говоря, весьма различны. Человек свободно оперирует образами: это тигр, а это кот, хотя те же “усы, лапы и хвост”; вот эта конструкция о четырех ногах, вон та на колесиках и даже та, что с одной вычурно изукрашенной подставкой – все это стол. Для компьютера же информация, а еще точнее данные, есть всего лишь последовательность, короткая или длинная, нулей и единиц. В самом начале компьютерной эры (по меркам истории человечества буквально вчера – каких-то полвека назад) мощности ЭВМ едва хватало на то, ради чего их создавали – помочь человеку в выполнении численных расчетов, к которым, так или иначе, сводятся большинство реальных задач. Естественно, что как особ королевских кровей, ЭВМ освобождали от всех побочных дел, вроде перевода информации из вида удобного человеку в вид, понятный машине – на их долю оставались чистые вычисления. Однако подобно тому как на подрастающих детей родители начинают перекладывать обязанности по уходу сначала за собой, а потом и за семьей в целом, так и на долю компьютеров с ростом их мощности падало все больше и больше задач, не связанных напрямую с выполнением расчетов. И если когда-то программирование велось в машинном коде, потом на ассемблере, затем на языках высокого уровня, то сейчас компьютер пытаются научить “понимать” обычную человеческую речь. Вполне возможно, что в будущем основным занятием программиста будет не “стучать по клавишам”, а с не меньшей скоростью “молоть языком”.

Повышение уровня “дружественности” компьютера к человеку ведется, конечно же, не только в области средств разработки программ, а точнее даже не столько, сколько в области прикладного использования компьютера как еще одного инструмента в руках человека. Без этого компьютер никогда не занял бы того места рядом с нами, которое он занимает сейчас, и так и остался бы инструментов “высоколобых” ученых. И в числе прочих эффектов это повышение дружественности привело к появлению целого класса специальных обслуживающих программ, реализующих промежуточный слой между “голым железом” и “полезными” программами, теми, что выполняют конкретные задачи пользователя. В результате возникла настоящая “среда обитания” программ, что привело, в свою очередь, к существенному усложнению самого процесса программирования, который люди снова начали упрощать, видимо, до следующего витка спирали.

Таким образом, цель данной главы – разобрать, в максимально облегченном варианте, из чего в настоящий момент складывается рабочая среда, в которой выполняется любая прикладная программа, с чем ей приходится взаимодействовать, и что, в конечном счете, должен иметь в виду программист, желающий чтобы написанная им программа не просто выполняла то, что от нее требуется, но и делала это по возможности эффективно.

Некоторые из изложенных здесь сведений пригодятся нам в дальнейшем, другие более не будут в этой книге затронуты, однако, хоть для того, чтобы водить автомобиль и не обязательно знать устройство двигателя внутреннего сгорания, но представлять себе его функциональные обязанности, возможности и потенциальные проблемы все же весьма полезно.

Процессор

Еще раз напомним: изначальное предназначение вычислительной техники, в точном соответствии с названием, – выполнение расчетов, следовательно, главный функциональный элемент любого вычислительного устройства должен… что? Считать? Да. Но не только. Вторая не менее важная задача – управление (здесь вполне уместна аналогия с мозгом человека – он тоже одновременно является и центром принятия решений и центром управления). Управлять

приходится самим собой, устройствами хранения инструкций и данных, устройствами ввода/вывода, и всеми остальными “участниками концессии”. Итак, знакомьтесь…

> Процессор, он же центральный процессор, он же ЦП, он же Central Processing Unit, он же CPU, он же “проц”, он же “камень” – основное *действующее* лицо любой вычислительной системы. Точен, исполнителен, трудолюбив, имеет привычку “гореть на работе”. В силу внутреннего устройства понимает только двоичную логику. Должностные обязанности: “координатор” и “вычислитель”. Слабости: при выполнении некорректных программ имеет тенденцию к “зацикливанию”, жить не может без “мамы”[1].

Современный процессор представляет собой очень сложное устройство – даже упрощенная структурная схема содержит десятки элементов. Однако не пугайтесь. Сколько-нибудь подробное рассмотрение этого вопроса выходит за рамки данной книги. Мы посвятим несколько слов лишь тем элементам и функциям процессора, понимание которых будет полезно нам в дальнейшем.

Но прежде небольшое отступление. Вспомним, какие виды чисел известны человечеству. Натуральные, целые, рациональные, иррациональные, вещественные (они же действительные). Возможно, напрягшись, кто-то припомнит еще и замечательные комплексные числа. Из всего, что перечислено, в компьютере с его двоичной системой счисления[2] достаточно просто могут быть представлены лишь натуральные числа, чуть посложнее с целыми отрицательными, а вот с вещественными совсем плохо (про комплексные же и вовсе умолчим – работа с ними это уже высшая математика, причем как в прямом, так и в переносном смысле – ни в одном из известных авторам процессоров работа с комплексными числами “в железе” не реализована). Из математического анализа известно, что количество вещественных чисел бесконечно не только, так сказать, “в длину” (вправо и влево по координатной оси), но и “в глубину”, то есть на любом сколь угодно малом отрезке [a, b] вещественных чисел также бесконечно много[3]. Таким образом, если целые числа “растут” только в одну сторону, слева от десятичной точки, то вещественные еще и вправо от нее. В результате для представления вещественных чисел используется специальный формат, получивший название “*число с плавающей запятой*”.

Из предыдущего нетрудно заключить, что в процессоре *вычислительных* блоков должно быть как минимум два: один для целых чисел, один для вещественных. На самом деле и тех и других (блоков) побольше, чем по одному, но это уже тема другой книги. Итак, запомним: “*Целые и вещественные числа обрабатываются в процессоре по-разному и даже в разных его частях*”. И еще одно: “*Кроме целых и вещественных чисел процессор ничего другого обрабатывать не умеет*”. К этим двум положениям мы не раз будем возвращаться в дальнейшем, освещая вопросы представления и интерпретации данных.

С расчетами разобрались. Теперь вторая задача процессора – управление. Каждая программа содержит набор команд (инструкций), указывающих процессору, что он должен сделать, откуда взять исходные данные, куда поместить результат. Понятно, что каждая такая команда должна быть процессору известна, то есть содержаться в его *системе команд*. Общее их количество относительно невелико (в последних процессорах компании Intel – пара сотен). Условно все команды можно разделить на два типа: *вычислительные* и *управляющие* (системные). С вычислительными вроде все ясно. А управляющие зачем? Представим, что у нас имеется горячее

---

[1] “мама” – материнская плата (motherboard).

[2] кто не в курсе, что это такое, подождите до следующей главы – пока достаточно знать, что компьютер в состоянии оперировать только двумя цифрами: нулем и единицей.

[3] да простят нас математики за столь вольную трактовку.

желание заставить процессор сложить два числа. Мы находим в списке команд ту, которая отвечает за сложение (не забывая, что для целых чисел команда будет одна, а для вещественных совсем другая), и… Возникает законный вопрос: Как же нам объяснить процессору, какие именно числа складывать? “Поговорить” с ним напрямую не удастся. Мы оперируем словами, процессор электрическими импульсами – не поймем друг друга. Вспоминаем, что вроде бы где-то слышали о том, что компьютер имеет устройство ввода информации. А, так вот же оно – клавиатура! С криком “Эврика!” нажимаем несколько цифровых клавиш и… Ничего не происходит. Даже с помощью клавиатуры передать данные процессору напрямую невозможно. Как именно это сделать – позднее, а пока допустим, что мы все-таки сумели. Следующий вопрос: куда их положить, чтобы процессор сумел их “достать”? К счастью, в самом процессоре существует *блок регистров* – часть процессора, специально предназначенная для хранения данных. Регистров этих немного, да и размер их невелик, но для нашей суперзадачи их хватит. Итак, прежде чем мы сможем выполнить команду сложения двух чисел, их необходимо *загрузить* в регистры, вот для этого, в частности, нам и понадобятся системные инструкции. Естественно есть множество других ситуаций, в которых процессор должен выполнять не вычисления, а функции управляющего, “говоря” остальным устройствам компьютера: “пойди туда”, “сделай то”, “подай это” и т.д.

Следующий важный момент – поступление данных в процессор. Взаимодействие с внешним миром процессор производит через *шину данных* – набор соединений, по которым данные в двоичной системе передаются в виде электрических сигналов. Чем больше соединений, тем большими порциями может передаваться информация. Размер “порции” называется *разрядностью шины*. Например, в процессоре Intel® Pentium® 4 шина данных 64-разрядна, то есть за единицу времени этот процессор может воспринять (или передать) блок из шестидесяти четырех нулей и единиц.

Процессор – устройство с дискретным “восприятием” времени. Моменты времени для него следуют друг за другом в виде *тактов*, в каждый такой *такт* “помещается” одно элементарное действие. Чем мельче такт, тем быстрее “живет” процессор, а значит, тем выше его быстродействие. Такт современных процессоров составляет менее одной миллиардной секунды. Такие числа не слишком удобны для восприятия, поэтому быстродействие процессора принято характеризовать его *тактовой частотой* – числом тактов в секунду (герц). Таким образом, конкретная модель процессора может быть описана, например, так: процессор Intel® Pentium® 4 с тактовой частотой 3,4 Ггц (гигагерц или GHz).

Оперативная память

В каждый момент времени (такт) процессор работает ровно с одной командой и средствами хранения команд не обладает[1].

Большинство команд представляют собой операции над некоторой порцией данных. Как мы выяснили выше, небольшое количество данных процессор все-таки в состоянии разместить “внутри себя”, в регистрах. Однако, кроме самых простейших случаев, регистров для хранения всех данных, которые должны быть обработаны в конкретной программе, недостаточно. Таким образом, и сами данные и команды для их обработки должны быть куда-то записаны. Из этого

[1] На самом деле это не совсем верно, а точнее совсем не верно, однако с точки зрения программиста ситуация именно такова, поскольку никаких средств прямого использования внутренней памяти процессора (кэш-памяти) нет.

“куда-то” процессор будет их извлекать, выполнять указанные действия и в это же “куда-то” сохранять полученные результаты. Итак, знакомьтесь…

> Оперативная память, она же ОП, она же Random Access Memory, она же RAM, она же “мозги” – второе по важности *действующее* лицо вычислительной системы. Как всякая женщина, весьма непостоянна – при выключении питания “забывает” все, что содержала. В отличие от процессора представляет собой совокупность физических и программных элементов. Должностные обязанности: “хранитель оперативной информации”. Для любой системы справедлив лозунг: “Памяти много не бывает”.

Дать точное определение *оперативной памяти* весьма непросто. Если процессор – это конкретное физическое устройство, которое можно подержать в руках, то на вопрос, что такое ОП, разные специалисты ответят Вам совершенно по-разному. Сборщик компьютеров скажет, что оперативная память – это одна, две или три небольших по размеру платы, устанавливаемые в соответствующие слоты на материнской плате. Программист, создающий прикладные программы, сообщит Вам, что для него оперативная память есть место хранения данных, при этом порции данных имеют имена, а размещение данных и связывание с именами осуществляется системой программирования. Программист, разрабатывающий обслуживающие (системные) программы, с уверенностью скажет, что оперативная память – это адресное пространство, то есть непрерывная последовательность пронумерованных ячеек одного и того же размера, в которых размещается код программы и обрабатываемые данные.

Каждое из приведенных определений отражает некоторые аспекты понятия оперативной памяти. Попробуем разобраться в причинах такого многообразия точек зрения и выделить то существенное, что понадобится нам в дальнейшем для правильного понимания процесса составления программ.

Итак, положение первое – устройство оперативной памяти неразрывно связано с операционной системой, поскольку без последней сегодня не функционирует ни один компьютер. Об операционной системе более подробно мы поговорим чуть позднее, а пока важно отметить одно следствие указанного факта.

Положение второе – оперативная память неоднородна. На верхнем уровне она состоит из двух частей. Первая, память физическая. Именно она в первую очередь используется для хранения кода исполняемых программ и их данных. Понятно, что сколь бы ни был велик объем физической памяти, всегда найдутся задачи, для которых ее требуется еще больше. Здесь, правда, существует принципиальное ограничение, накладываемое свойством процессора, которое называется *разрядность*. Современные процессоры используют “плоскую” модель адресации, то есть адрес любого байта оперативной памяти состоит из одного числа. В связи с этим разрядность процессора фактически представляет собой количество бит, отведенных на представление адреса ячейки оперативной памяти. Поскольку минимально адресуемая ячейка имеет размер в один байт, то объем памяти, с которым может напрямую работать процессор, определяется, как два в степени количество бит на адрес. Например, для 32-разрядных процессоров это число составляет 2^{32} байт, то есть 4Gb.

Объем физической памяти, устанавливаемой на компьютеры, обычно бывает существенно меньше того, что может адресовать процессор. Недостающую часть составляет память виртуальная, размещаемая на жестком диске. При нехватке физической памяти операционная система задействует виртуальную, что, естественно, сказывается на быстродействии, но зато позволяет запускать “прожорливые” программы, требующие больших ресурсов.

Положение третье – оперативная память однородна. Противоречия со вторым положением здесь нет. Неоднородность памяти имеет место с точки зрения операционной системы, которая, как заботливая хозяйка, скрывает этот факт от исполняемой программы. Таким образом, если

программа работает на 32-разрядном процессоре, то она совершенно спокойно может рассчитывать на объем памяти равный 4 Гб[1], как на линейное адресное пространство.

И, наконец, положение четвертое. При программировании на языке высокого уровня практически никогда не возникает необходимость обращаться к ячейкам оперативной памяти по абсолютным адресам. Более того, многие операционные системы это явным образом запрещают. В программе мы оперируем переменными, а их размещением в памяти и преобразованием имен в адреса занимается компилятор.

Долговременное хранение информации

Как уже было сказано, оперативная память хранит информацию, только пока на нее подается питание. Очевидно, в компьютере должно присутствовать и устройство, способное "не забыть" ее, если вдруг (какой ужас!) кто-то выдернет "шнур из розетки". Итак, знакомьтесь…

> Жесткий диск, он же винчестер[2], он же Hard-Disk Drive, он же HDD, он же "винт" – самое ценное *действующее* лицо вычислительной системы. Нетороплив (с точки зрения процессора и даже оперативной памяти), всегда готов принять на хранение любые Ваши секреты. Впрочем, при беспечном отношении с той же легкостью отдаст их кому-нибудь другому.

В отличие от двух предыдущих элементов вычислительной системы устройства жесткого диска мы даже касаться не будем. С точки зрения программиста значение имеет лишь то, *как* осуществляется долговременное хранение информации. Ключевым понятием в этом процессе является *файл*. Дать четкое определение этому понятию также не просто, как и оперативной памяти.

Итак, приближение первое: *файл – это поименованная область на диске*. Любой файл имеет имя. Чаще всего существуют ограничения на длину имени и допустимые символы, из которых оно может быть составлено. Любой файл некоторым образом располагается на диске, имеет начало (в "системе координат" жесткого диска) и длину в байтах (или более крупных единицах).

Приближение второе: нередко файл записывается на диск частями и в разные моменты времени, как следствие физически он может состоять из отдельных фрагментов дискового пространства. Таким образом, более точно можно сказать, что *файл – последовательность областей диска, логически связанных и имеющих общее имя*.

Подавляющее большинство долговременно хранящейся информации представляется в виде файлов, в том числе и сами программы.

Объемы жестких дисков уже довольно давно стали достаточными для размещения весьма большого числа файлов. В связи с чем возникла потребность обеспечить их группировку по произвольным признакам. Эта задача решается с помощью введения на диске еще одного

---

[1] На самом деле не вся эта память доступна программе – сколько именно определяет операционная система

[2] В 1973 году компания IBM представила диск "IBM model 3340 disk drive", который считается "отцом" современных жестких дисков. Эта модель имела два разделенных шпинделя, каждый с емкостью в 30 мегабайт. По этой причине диск часто называли "30-30", что и породило кличку "винчестер", в силу похожего названия ружья "винчестер 30-30".

важного объекта – *каталога* (папки). Каталог, также как и файл, имеет имя. Содержимым каталога является список файлов, которые считаются в нем размещенными.

С точки зрения прикладного программиста работа с жестким диском целиком и полностью происходит через высокоуровневые операции с файлами и каталогами, предоставляемыми в его распоряжение системой программирования, на которой пишется программа. Эти операции реализуются через функции операционной системы, под которую программа компилируется и на которой далее выполняется. Функции ОС в свою очередь обращаются к драйверу жесткого диска – специальной программе, переводящей управляющие команды в последовательность инструкций электронно-механической начинке винчестера. Изготовление драйверов обычно берет на себя фирма-производитель.

На этом мы закончим наш краткий экскурс в мир компьютерного “железа” и перейдем к “софту” – программному обеспечению.

Классификация программных средств

Существенная часть потребностей современного человека связана с восприятием и обработкой информации во всех ее видах: от текстового до мультимедийного, от формул до музыки.

Поскольку компьютер представляет собой универсальное вычислительное средство, то есть способен выполнить любую программу, которая может быть составлена на основе системы команд процессора, естественным является тот факт, что программ, удовлетворяющих наши потребности существует несколько больше, чем одна. А там, где есть множество объектов, человека всегда неодолимо тянет создать их классификацию. Итак, знакомьтесь…

> Программное обеспечение, оно же ПО, оно же Computer Software, оно же “софт” – совокупность всех программных средств, имеющихся в данной вычислительной системе. Если “железо” вычислительной системы – это мозг, то программы, хранящиеся на диске, – умения и навыки, а программы выполняющиеся – мысли.

Вернемся к задаче о сложении двух чисел. Есть команда процессора, есть оперативная память, в которую нужно разместить аргументы, есть клавиатура, на которой можно их набрать, есть жесткий диск, куда можно сохранить результат. Как связать все это в единое целое? Программа для выполнения этой простейшей операции должна быть способна обработать электрические импульсы от клавиатуры, “перекодировать” их в числа, разместить эти числа в оперативной памяти, передать адреса в команду сложения, получить адрес результата, считать его, найти место на жестком диске, куда записать файл с результатом, осуществить запись. Это если вкратце. Понравилось? Как Вы думаете, какой процент от размера такой программы составит собственно команда сложения?

Очевидно, что взаимодействие с аппаратурой вычислительной техники, – задача, которую необходимо решать в любой программе, – должно быть запрограммировано отдельно, то есть между программами, выполняющими задачи пользователя системы, и аппаратурой должен располагаться промежуточный слой из специальных обслуживающих программ. Этот промежуточный слой называется *системное программное обеспечение*. Все остальные программы относятся к *прикладным*. Классы программ пересекаются, то есть одна и та же программа может в зависимости от точки зрения быть отнесена либо к классу системных, либо к множеству прикладных.

В качестве примера: в системном программном обеспечении существуют так называемые *драйвера* – специальные программы, осуществляющие эффективное управление функциональными блоками вычислительной системы: жестким диском (о чем мы говорили выше), видео-, звуковой и сетевой картой и т.д.

Мы в нашей книге вопросов разработки программ системного уровня явным образом касаться не будем. Соответственно в дальнейшем, употребляя термин *программа*, мы будем иметь в виду уровень прикладной. Такие программы по имени класса, к которым они относятся, нередко называются *приложениями*.

Операционная система

Задача обеспечения удобного взаимодействия человека и вычислительной системы уже довольно давно ставится во главу угла при разработке программного обеспечения. При этом первые реальные пользователи вычислительной системы – это программисты. А среди программистов первыми “вступают в контакт” с аппаратурой программисты системные – создатели программ системного уровня. Однако при всем нашем уважении к “системщикам”, программистов прикладных все же существенно больше. Прикладные программисты пишут программы на языках высокого уровня, обычно способны путем некоторых манипуляций с набором исходных текстов получить *исполняемый модуль*, готовый к запуску, и ожидают, что все дальнейшее возьмет на себя кто-то еще. Итак, знакомьтесь…

> Операционная система, она же ОС, она же Operating System, она же OS, она же “операционка”, она же “ось” – основное *управляющее* лицо любой современной вычислительной системы. Представляет собой совокупность программных средств системного уровня. Должностные обязанности: обеспечение взаимодействия пользователя и вычислительной системы, обеспечение эффективного использования ресурсов последней, организация надежного функционирования программного обеспечения.

Устройство современных операционных систем не менее, а скорее даже более сложно, чем устройство самого компьютера. Различные аспекты их функционирования и использования освещаются в книгах толщиной во многие сотни страниц [16, 17]. Мы здесь рассмотрим лишь минимально необходимые моменты этой огромной и интересной темы.

Итак, момент первый – любая операционная система предоставляет пользователям некоторый интерфейс. На сегодняшний день наиболее распространены два варианта: интерфейс командной строки и интерфейс графический.

В интерфейсе командной строки основной режим работы – текстовый, способ взаимодействия с операционной системой – ввод команд с клавиатуры. Примерами ОС с интерфейсом командной строки являются операционные системы типа DOS, огромное число Unix-подобных систем, в том числе ставшая весьма популярной в последнее время Linux.

В графическом интерфейсе основной режим работы, как не трудно догадаться, графический, способ взаимодействия с ОС – выбор действий с помощью мыши, работа с окнами, меню, кнопками, панелями задач и другими элементами управления. Пример – любая из операционных систем семейства Microsoft Windows.

Достаточно очевидно, что графический интерфейс пользователя более удобен, однако требует и больше усилий со стороны разработчика.

Момент второй – операционные системы различаются по способу выполнения программ.

Однозадачные позволяют в каждый момент времени исполняться только одному приложению, предоставляя ему в распоряжение все ресурсы: процессорное время, оперативную память, экран монитора и т.д.

Многозадачные не накладывают ограничений на число одновременно выполняемых программ (в реальности их количество определяется лишь соотношением совокупных затрат оперативной памяти к объему доступной памяти системы). При наличии одного процессора действительная одновременность выполнения всех запущенных программ, конечно, невозможна. В результате она организуется разделением времени процессора между задачами. Квант времени, который отводится каждой задаче, достаточно мал, чтобы чередование доступа к процессору давало пользователю иллюзию параллельной работы. Большинство современных операционных систем являются многозадачными.

Момент третий – нередко операционная система предоставляет некоторые услуги не только пользователям, но и программистам, а именно обеспечивает их набором функций системного уровня – так называемым API (Application Programming Interface). Эти функции могут быть использованы при разработке программ под данную ОС, что, конечно, делает такие программы системозависимыми или, как говорят, непереносимыми, то есть неспособными функционировать на других операционных системах, но зато повышает их эффективность.

Операционные системы семейства Windows

Среди операционных систем семейство Microsoft Windows занимает особое положение. От версии 1.0, появившейся в 1985 году и умещавшейся на одной дискете (на самом деле Windows 1.0 операционной системой не являлась, а представляла собой графическую оболочку для DOS), произросло целое весьма развесистое дерево. Сегодня ОС Windows функционирует на персональных компьютерах (на подавляющем их большинстве) в виде линеек 9x – Windows 95/98/Me и NT –Windows NT 4.0/2000/XP; на серверах – Windows NT 4.0/2000/2003 Server; на мобильных устройствах – Windows CE и т.д.

Не вдаваясь в детали, отметим основные моменты, которые необходимо иметь в виду программисту при работе с Windows.

Прежде всего, Windows – операционная система с *графическим интерфейсом пользователя*. Основным объектом в этой операционной системе является окно, через которое “смотрит на мир” любая программа. Вследствие этого факта программы под Windows часто называют *оконными приложениями*. Windows обеспечивает некоторую стандартную функциональность по управлению окном: перемещение по экрану, изменение размеров, сворачивание/разворачивание и так далее. Все остальное, то есть собственно внешний вид – “лицо” Вашего приложения, необходимо программировать самостоятельно. Операционная система предоставляет для этих целей Windows API. Необходимо отметить, что создание полноценного графического интерфейса приложения на API требует весьма существенных усилий. Правда, для любителей, а также для ситуаций, в которых пользовательский интерфейс не имеет большого значения, существует возможность создания приложений *консольных*, функционирующих в старом добром текстовом режиме.

Во-вторых, Windows – система многозадачная, а значит, при написании программы Вы не можете рассчитывать на “единоличное” использование ресурсов вычислительной системы: процессорного времени, оперативной памяти, экрана монитора и т.д. Правда, существенную

часть необходимой работы для обеспечения разделения ресурсов берет на себя сама операционная система.

В третьих, Windows каждой программе при запуске предоставляет “отдельное” *виртуальное адресное пространство* (ВАП), размером в 4 Гб[1], из которых половину резервирует под себя, все остальное программист может свободно использовать[2]. Поскольку большинство компьютеров обладают меньшим объемом оперативной памяти, в реальности это пространство обеспечивается механизмами поддержки виртуальной памяти.

В четвертых, Windows – система событийно-управляемая. Приближенная схема функционирования ОС и исполнения приложений в ней выглядит следующим образом. Каждое действие, которое инициирует пользователь, каждая команда, которую операционная система “хочет” выдать выполняющимся приложениям, помещается в общую очередь. Точнее в очередь помещается сообщение, содержащее информацию, детально описывающую происходящее. Например, одним из типичных событий Windows является ситуация, когда некоторое окно, перемещаясь по экрану, открывает часть другого окна, находившуюся под ним. В этом случае система формирует сообщение с идентификатором WM_PAINT, содержащее указание открывшемуся окну перерисовать свое содержимое.

Все стандартные события имеют соответствующие им сообщения. Реализацию обработки некоторой части сообщений в каждом приложении операционная система выполняет автоматически. Остальные, те из них, которые необходимы данной программе, программист должен обрабатывать самостоятельно – Windows предоставляет для этого стандартный механизм.

В заключение раздела отметим, что разработка пользовательского интерфейса и обработка событий хоть и является интересной и подчас сложной задачей, все же не имеет непосредственного отношения к методам программирования, потому мы ее касаться не будем. Все примеры программ в данной книге будут выполнены нами в виде консольных приложений.

Выводы

Этой главой заканчивается вводная часть данной книги, в которой мы рассмотрели основные этапы разработки программ, инструменты, необходимые в этом процессе, а также немного охарактеризовали основные составляющие среды, в которой выполняются типичные прикладные программы. Содержание этих глав, на первый взгляд, не имеет непосредственной очевидной связи с процессом написания программ. Чтобы осознать, что эта связь есть, и она весьма существенна, требуется пройти в программировании некоторый путь. Научиться составлять простейшие, потом простые, потом средней сложности программы, после чего понять, что дальнейший рост сложности приводит к качественному скачку, когда на первый план выходят факторы, бывшие в простых ситуациях несущественными. Такие качественные скачки, пройденные сообществом программистов, приводят к появлению и утверждению технологий разработки, о которых и пойдет речь в дальнейшем.

---

[1] речь идет о 32-разрядных версиях Windows, в 64-разрядных Windows XP и Windows 2003 Server объем ВАП существенно больше.

[2] существует возможность “уговорить” Windows выделить приложению еще один гигабайт из четырех доступных.

4. Программа на языке Object Pascal

Pascal is very elegant. It's certainly still alive. It is prolific of successors and it has influenced language design profoundly.
Паскаль весьма изящен. Конечно, он до сих пор жив. У него тьма последователей, и он существенно повлиял на разработку языков программирования.

Dennis M. Ritchie
Bell Labs/Lucent Technologies

Свершилось! Пробившись сквозь дебри анализа предметной области, преодолев изучение классификации средств разработки, ознакомившись с некоторыми особенностями аппаратуры и современных операционных систем, мы, наконец-то, добрались до детального знакомства с языком программирования Object Pascal.

В данной главе мы заложим весьма прочный фундамент в здание под названием "Программирование с использованием языка Pascal", а в последующих главах возведем на этом фундаменте несколько этажей. Впрочем, как известно, чем выше окно, тем дальше из него видно, так и в нашем случае, чем больше мы будем узнавать, тем большая перспектива будет перед нами открываться. Надеемся, что к концу книги Вы сможете с удовлетворением сказать: "Как много пройдено! Какое счастье, что впереди еще больше!".

Историческая справка по языку Pascal

В ноябре 2000 года исполнилось 30 лет с момента первой официальной публикации описания языка Pascal. Произошло это событие в стенах Швейцарского федерального технологического института (Eidgenoessische Technische Hochschule – ETH), а сама публикация представляла собой недоступный широкой аудитории технический отчет. В том же 1970 году был написан первый компилятор языка Pascal (ETH Pascal). А в конце года Никлаусом Виртом[1], по праву считающимся отцом-основателем языка, было опубликовано первое официальное его описание с изложением синтаксиса и семантики.

В самом начале 1971 года упомянутый выше отчет был перепечатан в первом номере журнала "Acta Informatica" [42]. Так что рождение нового языка можно отсчитывать и с этого момента [1].

---

[1] Никлаус Вирт (Niklaus K. Wirth) – швейцарский ученый, профессор Швейцарского федерального технологического института (ETH – Eidgenoessische Technische Hochschule), идейный вдохновитель и непосредственный участник процесса эволюции семейства основанных на Pascal языков программирования.

Созданный изначально как язык для обучения студентов, позднее Pascal получил признание и в качестве коммерческого средства разработки программ. Так, в период с 1975 по 1980-е годы активно использовалась реализация под названием UCSD Pascal (Кеннет Боулес) [27].

Однако настоящую массовость использованию языка сумела обеспечить компания Borland International. Созданные под руководством профессора Андерса Хейльсберга среды программирования под общим названием Turbo Pascal (последняя версия среды имела номер 7.0) являлись одним из основных средств разработки программ для архитектуры x86 и операционных систем семейства DOS. Borland существенно расширила разработанный Виртом язык, сделав его мощным и удобным инструментом разработки, имевшем развитые средства поддержки технологий структурного и модульного программирования, а также несколько упрощенные возможности программирования в рамках объектно-ориентированного подхода. В последующих главах мы подробно рассмотрим основные положения данных технологий, причины их появления и преимущества, получаемые от их использования на практике.

С появлением первой завоевавшей массовую популярность операционной системы семейства Windows – Windows 3.1 компания Borland адаптировала под нее среду Turbo Pascal 7.0, переименовав ее попутно в Borland Pascal, и предоставив разработчикам инструментарий для создания оконных приложений в виде библиотеки OWL – Object Windows Library.

К счастью и на этом разработчики компании Borland не остановились – в начале 90-х годов язык Pascal подвергся, наверное, самой серьезной переработке с момента своего возникновения и с полным основанием был переименован в Object Pascal, став основой новой среды разработки Borland Delphi. Описание всех нововведений потребовало бы от нас отдельной главы, упомянем лишь весьма значительные изменения в поддержке объектно-ориентированного программирования и, конечно, качественную и продуманную реализацию новых на тот момент концепций визуального и компонентного программирования[1], что привело, с одной стороны, к огромной популярности среды разработки, а с другой, впервые сделало создание программ под Windows по-настоящему доступным[2].

Сегодня система Borland Delphi уверенно занимает свой сегмент рынка, по-прежнему используя Pascal в качестве базового языка (хотя компания Borland в последнее время называет сам язык Delphi).

Наряду с описанной ветвью развития языка, продолжались научные разработки и в области создания новых языков программирования на основе Pascal. Так, хорошо известна разработанная под научным руководством Вирта линейка “Pascal – Modula-2 – Oberon – Oberon-2” [32, 43, 44].

В последние годы начала активно распространяться новая платформа разработки .NET, созданная и продвигаемая на рынок компанией Microsoft. Технология .NET позиционируется как средство создания переносимых распределенных (в том числе Web) приложений и предоставляет в числе прочих возможности разработки многоязыковых программ. В рамках технологии .NET в институте ETH под руководством профессора Дж. Гудкнехта (J. Gutknecht) разрабатывается компилятор нового языка программирования Zonnon, основанного на Pascal и предназначенного для использования в рамках платформы .NET [29, 30, 31, 2]. Стоит заметить, что компания Borland также не осталась в стороне. Последняя их среда разработки Borland Delphi 2005 не

[1] Справедливости ради необходимо отметить, что первыми реализацию “визуального” подхода к построению программ осуществила компания Microsoft для языка Visual Basic.

[2] Тот, кто хоть немного знаком с WinAPI – этим краеугольным камнем, на котором зиждется программирование под операционные системы семейства Windows, – нас поймет.

только полностью интегрирована с .NET, но и содержит два базовых языка: Object Pascal и C#[1] – новый язык, разработанный компанией Microsoft специально под платформу .NET [25].

Синтаксическая характеристика языка

Любой язык представляет собой средство общения и передачи информации. В подавляющем большинстве естественных языков существуют две иногда весьма отличающиеся друг от друга формы: устная и письменная. Причем первый язык общения каждый человек изучает в устной форме и лишь затем узнает о существовании письменности, формальных правил словообразования, построения предложений, орфографии, пунктуации и т.д.

Языки программирования есть “средство общения человека и компьютера”, а потому должны быть максимально формализованы, любая конструкция языка должна иметь четкий однозначно понимаемый смысл. И, конечно же, языки программирования существуют исключительно в письменной форме – по крайней мере, до тех пор, пока компьютер на “научат” понимать обычную человеческую речь.

В отличие от первого языка общения, освоение которого приходится у человека на фактически бессознательный возраст, изучение второго и последующих языков обычно начинается именно с письменного варианта: алфавит, набор базовой лексики, грамматика. Именно в таком порядке и мы будем знакомиться с языком Pascal.

Методы описания синтаксиса (БНФ, синтаксические диаграммы)

Под *синтаксисом языка* формально понимается набор правил, разъясняющих, какие последовательности из символов алфавита языка являются допустимыми.

Под *алфавитом* понимается набор “атомарных” символов (букв), из которых может быть построен текст на алгоритмическом языке.

Описание синтаксиса языка представляет собой некоторую проблему. Если определить алфавит можно, просто перечислив все символы, число которых конечно и более того относительно невелико, то указать все допустимые цепочки символов, т.е. перечислить все правильные программы, практически невозможно. Что же делать? Решение – сформулировать *правила построения* допустимых цепочек.

Формальная спецификация абсолютно необходима для любого языка программирования высокого уровня, поскольку программы, написанные на этом языке должны автоматически переводиться транслятором в машинный код. Для подавляющего большинства популярных языков количество трансляторов несколько больше, чем один, хотя бы потому, что популярные языки используются на разных платформах, то есть под разными операционными системами, а иногда и на разной аппаратной базе. Поскольку одно из основных достоинств языков программирования высокого уровня – независимость или, говоря по-другому, переносимость, необходимо, чтобы любая реализация транслятора совершенно одинаково интерпретировала любую конструкцию языка.

---

[1] Читается “си шарп”.

Для строгого описания синтаксических правил изобретены специальные языки, получившие наименование *метаязыки* (“надязыки”). Такое название связано с их функцией “языка для описания языка”. Наиболее широко употребляемыми из них являются *металингвистические формулы Бэкуса-Наура* (язык БНФ) и *синтаксические диаграммы*.

Форма Бэкуса-Наура – формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие.

Впервые была использована для описания языка программирования Algol-60. Известна также *Расширенная форма Бэкуса-Наура*, в которой для удобства изложения добавлено некоторое количество конструкций.

Полное изложение БНФ увело бы нас далеко в сторону, здесь мы ограничимся лишь парой примеров.

Вот как на языке БНФ может быть описано правило, определяющее цифру: *<цифра> ::= 0|1|2|3|4|5|6|7|8|9*

Здесь знаки “<”, “>”, “::=” и “|” являются метасимволами языка БНФ и используются в следующем смысле: угловые скобки ограничивают понятия описываемого языка (метапеременные), знак “::=” читается как “по определению есть” или просто “это”, а вертикальная черта читается как “либо” (“или”).

А вот так может быть описано правило, определяющее синтаксически допустимую последовательность символов алфавита (в данном случае арабских цифр), для изображения целого числа без знака:

<целое без знака> ::= <цифра> | <целое без знака><цифра>

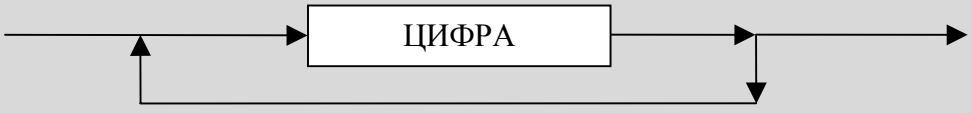
Приведенная формула задает рекурсивное определение целого числа без знака, т.е. определение через само себя. Рассмотрим, как “раскручивается” такое определение.

Во-первых, вспомним, что “цифра” уже определена нами ранее.

Во-вторых, заметим, что первым (до знака “|”) стоит определение целого числа без знака как цифры. Тогда становится понятно, что вторая часть определения поясняет, что для построения правильного целого числа следует справа к числу (а первый раз это одна цифра) приписывать какую-нибудь цифру.

Очевидным недостатком языка БНФ с точки зрения обучения является невозможность, например в нашем случае с целым числом, указать, сколько же может быть цифр в числе. С некоторой точки зрения это неважно (ниже мы поясним, о чем идет речь), но человека, изучающего язык по металингвистическим формулам, такой способ определения целого числа может ввести в заблуждение.

Синтаксические диаграммы, которые часто используются для описания именно языка Pascal, по существу являются *графической формой записи металингвистических формул*. Определение того же понятия “целое без знака” может выглядеть следующим образом:



где метапеременная ЦИФРА заключена в прямоугольник, а ее повторяемость графически изображена обратной стрелкой, задающей цикл повторений.

Не останавливаясь на обсуждении достоинств и недостатков приведенных метаязыков, укажем главное. Основное их назначение не связано с изложением языка для пользователя. Они решают другую задачу – обеспечивают возможность формального описания элементов и конструкций языка для их однозначной интерпретации. При этом понятно, что разработчику транслятора не надо, например, пояснять, сколько может быть цифр в целом числе. Он реализует представление целых чисел в соответствии с принятыми спецификациями.

Для пользователя же языка более приемлемым является словесное неформальное описание языковых конструкций, сочетающее синтаксис и семантику, т.е. правила истолкования, смысл предложений. Именно таким способом мы и будем пользоваться в дальнейшем, иногда прибегая к угловым скобкам для обозначения метапеременных, а также к еще одним метасимволам – квадратным скобкам, обозначающим, что заключенная в них конструкция может быть опущена. Заметим, что и угловые скобки (символы “меньше” и “больше”), и квадратные скобки входят в алфавит языка Pascal.

Алфавит языка

Чтобы получить общее представление об *алфавите языка*, достаточно взглянуть на клавиатуру компьютера – большинство из изображенных там символов входит в “азбуку” любого языка программирования. Конечно же, и Pascal не является исключением из этого правила.

Мы не будем перечислять здесь все допустимые символы, а сделаем лишь несколько общих замечаний.

Во-первых, обратим внимание на использование русских букв. Их допускается применять только в комментариях к программе и в строковых и символьных константах. Никакие другие языковые конструкции не могут содержать русские буквы.

Во-вторых, отметим, что за исключением символьных и строковых значений компилятор в полном соответствии с правилами языка не различает заглавные и строчные буквы, то есть слова **`Program`** и **`program`** в языке Pascal являются идентичными. Таким образом, использование букв в разных регистрах является лишь вопросом стиля записи программы и дополнительной возможностью повышения ее читабельности, о чем мы будем далее неоднократно упоминать.

Спецсимволы

Спецсимволы – символы и пары символов, которые имеют специальное, заранее определенное значение.

К числу спецсимволов языка Pascal относятся следующие символы и пары символов:

```
#   $   &   '   (   )   *   +   ,   -   .   /   :   ;   <   =   >   @   [   ]   ^   {   }
(*   (.   *)   .)   ..   //   :=   <=   >=   <>
```

Следующие символы и пары символов являются эквивалентными:

```
[     (.
]     .)
{     (*
]     *)
```

Ключевые слова

В известном смысле в алфавит языка входят и ряд так называемых *ключевых (зарезервированных, служебных) слов* – в примерах программ, которые рассматриваются на протяжении данной книги, они выделены жирным шрифтом. Эти слова не могут быть использованы ни в каком другом смысле, кроме изначально приписанного разработчиками языка. Пояснять значение этих слов мы будем по мере необходимости. Заметим, что выделение нами ключевых слов с помощью жирного шрифта служит лишь иллюстративным целям. В реальной практике характер выделения ключевых слов (или отсутствие такового) целиком зависит от текстового редактора, который используется для работы с кодом программы.

automated	**at**	**and**	**array**	**as**	**asm**
begin	**case**	**class**	**const**	**constructor**	**destructor**
dispinterface	**div**	**do**	**downto**	**else**	**end**
except	**exports**	**file**	**finalization**	**finally**	**for**
function	**goto**	**if**	**implementation**	**in**	**inherited**
initialization	**inline**	**interface**	**is**	**label**	**library**
mod	**nil**	**not**	**object**	**of**	**or**
on	**out**	**packed**	**procedure**	**program**	**property**
raise	**record**	**repeat**	**private**	**protected**	**public**
published	**resourcestring**	**set**	**shl**	**shr**	**string**
then	**threadvar**	**to**	**try**	**type**	**unit**
until	**uses**	**var**	**while**	**with**	**xor**

Идентификаторы

Любой объект программы (переменная, константа, тип данных, процедура и т.д.) должен иметь имя. В языках программирования имена называются *идентификаторами*.

Правила записи идентификаторов отличаются не только от языка к языку, но и в разных версиях одного и того же языка. На данный момент в языке Object Pascal эти правила состоят в следующем:

- идентификатор может начинаться только с буквы или символа подчеркивания;
- далее могут следовать буквы, цифры и знак подчеркивания;
- формально, длина, т.е. количество символов, идентификатора не ограничена, однако только первые 255 являются значащими. Другими словами, два разных идентификатора должны иметь различия в первых 255 символах, что, конечно, не является серьезным ограничением с практической точки зрения.

Сразу отметим, что хороший стиль программирования подразумевает выбор имен не “с потолка”, а в соответствии со смыслом обозначаемого объекта. При этом рекомендуется пользоваться заглавными буквами и символом разделения – подчеркиванием. Не следует также бояться (а точнее говоря, лениться) записывать длинные имена – это окупится большей ясностью программы.

Приведем примеры идентификаторов: `ListOfPersons`, `Window_12`, `cpu86`.

Еще раз обратим внимание, что в именах нельзя использовать русские буквы, а также на то, что имена не должны совпадать с ключевыми словами.

Заметим следующий факт: многие начинающие программисты (конечно, мы далеки от мысли, что Вы считаете себя начинающим ☺. В этом случае, Вам проще будет осознать то, что сейчас будет написано…) не придают никакого значения проблеме правильного именования.

Удивительно, как много программ содержат имена вроде `aaa`, `qqq`, `qweqwe`, `qq11`, `a1`, `a2`, `a22`, `Button1`, `Form1`, и т.д. Неизбежным следствием этого безобразия является превращение текста программы в некую шифрограмму, что нивелирует сам смысл применения языка программирования высокого уровня. Мы берем хороший инструмент и пренебрегаем его возможностями. Что можно понять, глядя на следующую строку текста программы?

```
qqq := (a1 + a2) / 2 * StrToFloat(Edit.Text);
```

С определенностью лишь одно – здесь что-то складывается, делится и умножается, а вот что именно, зачем и почему, можно выяснить только путем “долгого и мучительного” анализа контекста для выяснения смысла, заложенного автором в каждую из участвующих в выражении переменных. Непонимание или хуже того игнорирование этого факта, конечно же, выйдет боком самому горе-программисту, когда он через месяц попытается модифицировать свою программу, добавляя в нее новую функциональность или устраняя внезапно обнаруженные ошибки.

Хотелось бы сразу избавить читателя от иллюзий: мысль о том, что “мы сейчас сделаем как-нибудь, а вот когда понадобится, будем делать по-человечески” на практике нереализуема. Когда доходит до дела, люди, привыкшие работать по принципу “тяп-ляп”, с большим трудом перестраиваются. Таким образом, с нашей точки зрения совершенно необходимо сразу привыкнуть присваивать всем объектам содержательные имена – в дальнейшем эти усилия окупятся сторицей.

Объявления

Объявления – специальные выражения языка, указывающие на то, что мы хотим далее в своей программе использовать идентификатор в тех или иных целях. Сам идентификатор и цели использования вытекают из содержимого объявления.

```
var
  Number: Integer;
const
  MinNum = 10;
  MaxNum = 1000;
type
  TElement = Word;
  TMyArray = array[MinNum..MaxNum] of TElement;
```

Общая идея, реализованная в языке Pascal повсюду (существующие исключения мы рассмотрим по мере необходимости) заключается в том, что мы должны “сначала объявить – затем использовать”. Так, обязательны предварительные объявления переменных, констант, новых типов данных, подпрограмм и др.

Операторы

Понятие оператора является, с одной стороны, основным в большинстве языков программирования, с другой дать ему четкое определение весьма непросто. Попробуем дать определение конструктивное. Прежде всего, *оператор* – любая *исполняемая* конструкция языка программирования. Некоторые такие конструкции являются составными, то есть включают в себя другие операторы. Довольно часто набор из нескольких операторов, синтаксически или семантически связанных, также называется оператором. Как видим, определение получается рекурсивным. Подводя итог, можно сказать, что оператор есть выраженное средствами языка программирования действие.

Естественным образом возникает вопрос: “Сколько операторов должно быть в языке?”. Разговор о том, сколько операторов *достаточно* мы отложим до следующей главы, а сейчас подумаем над таким возможным ответом: “Чем больше, тем лучше”. Казалось бы, ответ разумен, чем больше операторов, тем более выразительным становится язык программирования. Однако здесь скрывается существенная проблема.

Представьте себе, что Вы, вместе с коллективом программистов работаете над большим проектом. Представьте, что средством разработки служит воображаемый язык “SuperPascal nonsense edition”, в котором определена пара сотен операторов. Будете ли Вы знать их все? Честно говоря, на этот счет нас “терзают смутные сомнения”. Скорее всего, дело ограничится десятком другим операторов, тех, что наиболее часто используются, а за остальными придется при необходимости обращаться к справочникам. В результате в коллективе неизбежно возникновение несовпадающих подмножеств операторов, известных разным людям, что существенно затруднит совместную работу над проектом.

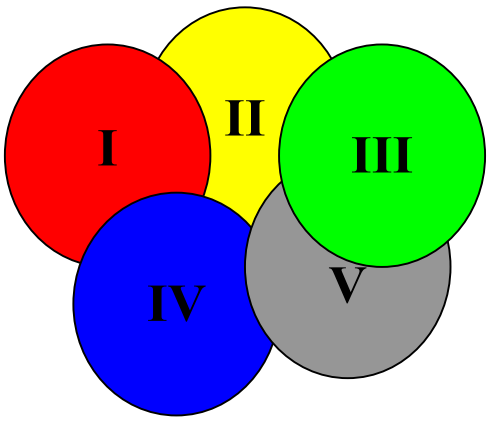


Рис. 4.1. Перекрытие областей знаний программистов для операторов языка “SuperPascal nonsense edition”

Отсюда вывод – да здравствует разумная достаточность!

Откроем маленький секрет – этот лозунг обычно выдерживают и создатели языков программирования, большая их часть имеет сравнительно небольшой набор операторов. В следующей главе мы узнаем, чем этот набор определяется.

Комментарии

Комментарии – это пояснения к программе. Комментарии не анализируются компилятором и никак не влияют на ее исполнение. Комментарий может быть записан в любом месте программы, где может стоять пробел. Правила его записи следующие: комментарий заключается либо в фигурные скобки: `{` и `}`, либо в ограничители, состоящие из круглых скобок со звездочками, `(*` и `*)`, внутри скобок могут находиться любые символы, включая и русские буквы.

```
(* Все это комментирующий текст *)

{ И это тоже }

(*
  И даже это
*)

{
  И, наконец, это
}
```

С появлением систем визуального программирования Delphi, стало допустимым использование в тексте программы на языке Object Pascal комментариев в стиле C++, в частности, текст в строке, следующий за `//` является комментарием.

```
// Комментарий
```

> Тот, кто изучал программирование, наверное, уже не один раз слышал и читал настоятельную рекомендацию: “Пишите комментарии!”. Причина этого призыва понятна. Программа – это не только инструкция для работы компьютера, но и документ, фиксирующий идеи решения задачи, часто весьма нетривиальные. Программа может быть и часто бывает использована как способ сообщения знаний другому лицу. Кроме того, и сам программист через некоторое время забывает то, как он реализовывал те или иные моменты и, если приходится возвращаться к старой некомментированной программе, испытывает определенные затруднения в ее понимании.
>
> С другой стороны, авторы знают и на собственном опыте и из бесед с другими программистами, имеющими большой стаж работы, как, несмотря на понимание необходимости использования комментариев, и даже, несмотря на свой печальный опыт их игнорирования, тем не менее, трудно заставить себя отвлечься на некоторую вспомогательную “черновую” работу, будучи увлеченным составлением хитроумного алгоритма и его оперативной отладкой. Здесь можно дать только один совет. Вставляйте комментарии не сразу, а после того как записан и отлажен некоторый фрагмент программы. Но делайте это обязательно!

Наконец, последнее замечание. Комментарии должны дополнять текст программы, а не разъяснять правила работы операторов. Например, комментарий типа

```
{ складываются числа Segment и Offset }
```

следует признать крайне неудачным, так как предполагается, что читающий программу и так знает язык и, соответственно, может понять, как работает оператор, особенно если за идентификаторами закреплен известный читателю смысл.

Если говорить о некоторой общепринятой практике, то обычно в первую очередь записываются комментарии к программе в целом, к отдельным подпрограммам, а также дается содержательное описание объектов программы.

Директивы компилятору

Язык Pascal содержит специальную синтаксическую конструкцию, позволяющую управлять компиляцией программы, точнее выдавать команды компилятору по включению в исполняемый модуль или исключению из него тех или иных групп вспомогательных машинных команд. Пусть, например, программа должна обрабатывать некоторую информацию, хранящуюся на жестком диске в виде файла. Если имя этого файла должен задать пользователь программы, он вполне может ошибиться при его вводе. В этом случае при отсутствии специальных указаний компилятор автоматически вставит в исполняемый модуль команды, анализирующие характер завершения операции поиска файла и вызывающие принудительное завершение программы в случае, если файл не был найден. Очевидно, что в подавляющем большинстве случаев ситуация неверного ввода имени файла не является критической и вместо аварийного выхода из программы пользователю нужно предложить ввести имя файла заново. В этом случае нам и понадобятся директивы, чтобы “объяснить” компилятору, что мы не нуждаемся в автоматической обработке, а будем выполнять ее сами.

Синтаксически директивы в языке Pascal записываются в фигурных скобках, подобно комментариям, однако сразу же после открывающей скобки должен стоять символ “знак доллара”, т.е. $. В этом случае компилятор не игнорирует данный текст, а рассматривает его как указание к действию. После знака $ следует имя директивы – одна буква или развернутое наименование (некоторые директивы имеют оба варианта) – и признак включения или выключения действия: знаки плюс, минус или слова `ON`, `OFF`. В дополнение к этому можно записать произвольный текст, скажем для пояснения директивы. В случае неверного написания имени директивы компилятор выдаст сообщение об ошибке. Конкретные директивы мы будем рассматривать по мере необходимости. А сейчас приведем пример директивы отключения контроля выполнения операций ввода-вывода, использованной, кстати, еще в Примере 1.1:

```
{$I- отключить автоматический контроль ввода-вывода}
```

Директива может также выглядеть следующим образом:

```
{$WARNINGS OFF}
```

Структура программы

Синтаксис языка Object Pascal значительно более “либерален”, чем тот, что был определен изначальным стандартом языка Pascal. Программа в Object Pascal состоит из операторов или предложений языка, которые разделяются между собой точкой с запятой “;” и записываются в свободном формате. Компилятор рассматривает программу как последовательность строк, каждая из которых содержит последовательность символов и включает ряд операторов, между которыми может находиться сколько угодно пробелов. Пользуясь возможностями свободного формата, программист произвольным образом разбивает текст на строки, руководствуясь размерами экрана (или размера листа бумаги, на котором будет печататься текст) и соображениями хорошего стиля записи программ. Для большей наглядности программы в нее вставляются пустые строки, отделяющие фрагменты алгоритма, используются отступы при

записи операторов (все примеры в книге выдержаны в некотором общем стиле, включающем, помимо прочего, и данное правило).

В общем случае последовательность записи элементов программы должна выглядеть следующим образом (впоследствии эта схема будет уточняться):

```
[Program <Имя программы>;]
[uses <список имен используемых библиотек>;]
[<объявления меток>]
[<объявления констант>]
[<объявления типов>]
[<объявления переменных>]
[<объявления и описание процедур и функций>]
begin
  [<исполняемые операторы>]
end.
```

Программа начинается (если не считать заголовочных комментариев) с ключевого слова **Program**, за которым следует имя программы – идентификатор. В языке Object Pascal это предложение можно опустить, о чем говорят метасимволы квадратные скобки. Надо сказать, что имя программы, задаваемое в заголовке, с точки зрения ее выполнения не играет никакой роли (поэтому его и можно опускать). Тем не менее, мы настоятельно рекомендуем начинать текст программы с оператора **Program** и использовать его для указания в названии программы ее назначения.

После заголовка программы следует необязательная секция подключения библиотек, начинающаяся с ключевого слова **uses** (“использует”). Более подробно мы поговорим об этом в главе 7. На практике только очень простые программы могут обойтись без использования этой секции.

Далее по порядку следуют разделы различных объявлений. Еще раз отметим, в языке Pascal *все* объекты в обязательном порядке должны быть *объявлены*. Ни один объект не может быть использован ни в каком качестве без предварительного объявления. Каждый из указанных выше разделов объявлений мы рассмотрим достаточно подробно. Пока же отметим, что в отличие от стандарта языка в версии Object Pascal разделы объявлений могут располагаться в произвольном порядке и даже вперемешку, т.е. например, между разделами объявления переменных может находиться раздел с определениями типов. Единственное правило должно соблюдаться неукоснительно: “сначала объяви, потом используй” (правда, как было сказано ранее, и здесь есть исключения).

Наконец, самым последним (это правило нарушено быть не может) следует раздел операторов – *тело программы*. Он начинается с ключевого слова **begin** и завершается ключевым словом **end** с последующей точкой, которая собственно и является признаком конца текста программы, так как слово **end** может встретиться и в другом контексте.

Из приведенной схемы нетрудно заключить, что минимальной программой на языке Pascal (ничего не делающей, но и не содержащей ни одной ошибки), является программа:

```
begin
end.
```

В заключение отметим, что совокупность разделов описаний и операторов (без предложения **Program** и заключительной точки) называется *блоком*.

Типы данных

Информация и формы представления данных

Цель решения любой задачи – получение новой *информации*. Информация заключается в данных и извлекается из них путем их анализа. Другими словами, данные и информация, вообще говоря, не одно и тоже. Информация дополнительно подразумевает существование некоторого способа интерпретации данных. Вследствие чего возникает необходимость в наиболее информативном их представлении, т.е. представлении в виде, максимально облегчающем интерпретацию.

Минимальной единицей измерения информации является *бит* (сокращение от *binary digit*).

Один бит соответствует двум различным значениям, обычно обозначаемым 0 и 1. Восемь бит составляют 1 *байт*, 2^{10}=1024 байт – 1 *килобайт*, 2^{10}=1024 килобайт – 1 *мегабайт*, 2^{10}=1024 мегабайт – 1 *гигабайт* и т.д.

Понятие типа данных

Как известно, данные в памяти компьютера должны быть представлены в виде последовательностей двоичных цифр, другими словами в алфавите {0, 1}. Очень не многие из нас способны связно мыслить в таком формате. Во всяком случае, тексты, графики, диаграммы, изображения, звуковые сигналы намного более привычны для нас. Таким образом, говоря о данных, мы должны различать два способа их представления: внутреннее – в памяти компьютера и внешнее – на мониторе, принтере, через звуковую карту и динамики. При этом естественно должны существовать средства, осуществляющие перевод данных из одной формы представления в другую. Например, для текстовой или числовой информации такой перевод – задача операторов ввода/вывода.

Остановимся на способах внутренней кодировки более подробно.

Прежде всего, отметим – данные, представляющие различные виды информации, конечно же, и кодируются по-разному, хотя и в одном и том же алфавите.

Классическим примером является различие в подходах к кодировке целых и вещественных чисел, о чем мы частично говорили в предыдущей главе. Казалось бы, поскольку множество целых чисел есть подмножество чисел вещественных, было бы разумно все расчеты вести исключительно в них. На самом деле это, конечно же, не так. Во-первых, аппаратная реализация команд для работы с целыми числами существенно более проста и эффективна с точки зрения быстродействия, чем реализация команд для вещественных чисел. Во-вторых, вещественные числа в их строгом математическом понимании реализовать на сугубо дискретном устройстве, каковым является процессор, просто невозможно, в связи с чем организация корректных вещественных вычислений на компьютерах представляет собой отдельную весьма непростую задачу.

Проиллюстрируем эту проблему на простом примере. Как известно, количество бит (физических носителей двоичной информации в оперативной памяти) для представления числа всегда ограничено аппаратными возможностями машины. Можно провести аналогию с “окошком” калькулятора, в котором можно высветить только определенное количество цифр. Если мы

наберем на калькуляторе число 1, а затем разделим на 3, то в окошке появится результат 0.3333333 (если калькулятор рассчитан на представление 8 десятичных цифр). Далее при умножении этого числа на 3 мы получим 0.9999999, что, строго говоря, не равно 1. Итак, (1/3)*3 не равно 1. Нечто подобное происходит и при выполнении операций над двоичным кодом.

Итак, целые и вещественные числа представляются в различных внутренних форматах и имеют разные наборы машинных команд для выполнения арифметических операций.

Обобщим сказанное. Обработка данных выполняется с использованием оперативной памяти. Данные должны быть некоторым образом закодированы в двоичный вид, что автоматически подразумевает наличие различных способов интерпретации одной и той же области памяти в зависимости от вида размещенных в ней данных. Объем памяти, который может быть выделен под хранение любого данного, естественным образом конечен. Наконец, вид данных автоматически определяет допустимые операции их обработки.

Все сказанное и составляет понятие *тип данных*:

4. множество значений;
5. набор операций, применимых к значениям данного типа;
6. способ представления и интерпретации данных в памяти компьютера;
7. размер оперативной памяти, необходимый для хранения данных в памяти компьютера.

Заметим, что не все указанные пункты независимы (очевидно, что пункты 1 и 4 взаимосвязаны). Однако представляется важным указать их все. Далее с каждым пунктом мы познакомимся подробнее.

Представление чисел. Системы счисления

В данном разделе мы рассмотрим важный теоретический материал, который поможет в дальнейшем разрешать вопросы, связанные с представлением чисел в оперативной памяти, способами интерпретации располагаемых в ней данных, вычислением объема памяти, необходимого для хранения данных.

Понятие системы счисления

Со школьных времен многим из нас известно волшебное словосочетание *система счисления*. Правильное понимание этого понятия имеет в нашем контексте весьма существенное значение. А потому попробуем приоткрыть завесу тайны, скрывающую под собой его смысл.

Давным-давно люди поняли, что для адекватного осуществления разных бытовых операций (обмен, учет, торговля и т.д.) необходимы какие-то абстрактные единицы измерения. Так появились *числа*. Трудно сказать однозначно, кто и когда первым ввел это понятие. Однако достоверно известно, что Евклид в своих “Началах” уделил вопросу о числе большое внимание. Большая советская энциклопедия определяет число так: “Число – важнейшее математическое понятие. Возникнув в простейшем виде еще в первобытном обществе, понятие числа изменялось на протяжении веков, постепенно обогащаясь содержанием по мере расширения сферы человеческой деятельности и связанного с ним расширения круга вопросов, требовавшего количественного описания и исследования. На первых ступенях развития понятие числа определялось потребностями счета и измерения, возникавшими в непосредственной

практической деятельности человека. Затем число становится основным понятием математики, и дальнейшее развитие понятия числа определяется потребностями этой науки".

С появлением чисел возникла и новая как для математики, так и для лингвистики проблема – как записывать числа? Для обозначения чисел понадобилось придумать графические изображения элементарных количеств. Эти графические обозначения в настоящий момент мы с Вами называем *цифрами*. Но наличия одних лишь цифр, конечно же, недостаточно. Необходимы еще правила построения (записи) из них "больших" чисел. Именно эти правила – набор соглашений, принятых людьми, относительно графического изображения числовых величин и способа их интерпретации – и составляют в совокупности *систему счисления*.

Все развитые цивилизации обладали своими системами счисления; так историки и археологи обнаружили такие системы в Древнем Египте, Греции, Вавилоне, Риме, Иудее, Индии, у славян, индейцев майя и т.д. К слову Вавилонская система используется всеми нами до сих пор. Да, да, не удивляйтесь! Именно из Вавилона к нам пришли 60 минут в часе, 60 секунд в минуте, 12 месяцев в году, примерно 30 дней в каждом месяце.

Непозиционные и позиционные системы счисления

Анализируя разные системы счисления, использовавшиеся в древности или применяемые в наши дни, можно обнаружить два вида таковых: *позиционные* и *непозиционные*.

В непозиционных системах положение цифры не оказывает влияния на ту величину, которую она обозначает. Классическим примером непозиционной системы является Римская система счисления. Проиллюстрируем определение на примерах:

```
XI = 10 + 1 = 11
IX = 10 - 1 = 9
XXV = 10 + 10 + 5 = 25
```

Из примеров видно, что знак **X** в любом случае обозначает 10, вне зависимости от того, на каком месте он располагается.

Главным недостатком непозиционных систем является сложность и неудобство осуществления арифметических операций.

В позиционных системах, одним из существенных элементов которых является понятие "разряд", ситуация принципиально упрощается, поскольку появляется возможность задать четкие однозначные алгоритмы выполнения операций – известные нам с детства сложение, вычитание и умножение "столбиком", деление "уголком".

Математические основы систем счисления

Рассмотрев содержательную сторону дела, перейдем теперь к формальному математическому описанию проблемы.

Пусть $p \in N$, $p \geq 2$ – *основание* (количество цифр) системы счисления[1].

Пусть $A_p = \{c_0, c_1, \ldots, c_{p-1}\}$ – *алфавит* системы счисления.

Будем называть такую систему счисления *p-ичной системой счисления* с алфавитом A_p, а $c_0, c_1, \ldots c_{p-1}$ – *цифрами* системы счисления.

[1] Здесь и далее рассматриваются только позиционные системы счисления

В качестве графических обозначений цифр обычно используют символы 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 и, если их не хватает, заглавные буквы латинского алфавита (если Вам и этого не хватит(!), напишите нам, мы поможем Вам обойтись в задаче системой счисления с меньшим основанием).

В соответствии с этой трактовкой рассмотрим примеры наиболее употребительных в течение второй половины XX-го века систем счисления:

- $p = 2$, $A_2 = \{0, 1\}$ – *двоичная* система счисления;
- $p = 8$, $A_8 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ – *восьмеричная* система счисления;
- $p = 10$, $A_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ – *десятичная* система счисления;
- $p = 16$, $A_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ – *шестнадцатеричная* система счисления.

Десятичная система счисления знакома каждому из нас с детства, именно в ней подавляющее большинство человечества выполняет расчеты. Двоичная система знакома “с детства” почти каждому вычислительному устройству, т.к. именно в этой системе представляются данные в памяти компьютера. Отметим простой факт (тем, кому он неизвестен со школы, станет ясно, в чем дело, через пару страниц): чем больше основание системы, тем короче запись числа. Так, шестнадцатеричная система очень удобна для записи машинных адресов.

Важным для математики вопросом является вопрос о связи значения числа с его изображением в виде системы цифр. Проиллюстрируем решение этого вопроса для натуральных чисел.

Известна теорема о том, что любое десятичное натуральное число N можно единственным образом разложить по степеням p так, что

$$N_{10} = a_k p^k + a_{k-1} p^{k-1} + \ldots + a_1 p + a_0 \quad (4.1)$$

где $a_k \neq 0$; $0 \le a_i \le p-1$, $0 \le i \le k$ и все a_k и p – десятичные числа.

Заменим теперь каждое десятичное число a_i (любое из них меньше чем p) соответствующей ему p-ичной цифрой c_i и выпишем все цифры слева направо по убыванию степеней числа p. Полученная в результате запись по определению есть *позиционная запись числа N в p-ичной системе счисления*.

$$N_{\mathbf{10}} = c_k c_{k-1} \ldots c_1 c_0{}_{\mathbf{p}} \quad (4.2)$$

Во избежание неоднозначности индекс при числе есть основание системы счисления, в которой записано число.

Перевод чисел из десятичной системы счисления в другую и наоборот

Из формул (4.1), (4.2) следуют алгоритмы преобразования между десятичной и p-ичной системами счисления. Рассмотрим их подробнее.

1. Для преобразования числа N из системы с основанием 10 в систему с произвольным основанием p необходимо получить разложение (4.1) и заменить все коэффициенты на цифры p-ичной системы счисления. Для вычисления коэффициентов a_i рекомендуется делить число N на p “уголком”, затем частное от деления снова делить на p, и так до тех пор, пока очередное частное не окажется меньше, чем p. При этом необходимо запоминать остатки от деления. После этого справедливо следующее: последнее частное равно a_0, последний остаток – a_1, …, первый остаток равен a_k. После этого необходимо осуществить замену коэффициентов на p-ичные цифры.

2. Для преобразования числа N из системы с произвольным основанием p в систему с основанием 10 необходимо записать разложение (4.1), предварительно заменив все цифры p-ичной системы счисления на их десятичные эквиваленты и произвести операции возведения в степень, сложения и умножения, вычислив результат.

Перевод чисел из системы счисления с основанием p в систему счисления с основанием q

В общем случае перевод чисел между двумя произвольными системами счисления выполняется через промежуточные операции перевода в десятичную систему, что на практике для больших чисел может оказаться довольно проблематичным (по крайней мере, при решении “на бумаге”). Известны некоторые частные случаи, а именно: перевод чисел между двоичной, восьмеричной и шестнадцатеричной системами счисления.

Рассмотрим **перевод чисел между двоичной и восьмеричной системами**. Используемый для этого прием называется *метод триад* (троек). Суть его заключается в том, что каждой восьмеричной цифре ставится в соответствие комбинация нулей и единиц по следующей таблице:

Восьмеричная цифра	Двоичное представление
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Таблица 4.1. Метод триад

Так число $127_8 = 001\ 010\ 111_2$, где ведущие нули, разумеется, необходимо отбросить. При переводе из двоичной системы в восьмеричную необходимо дополнить число слева нулями так, чтобы все его двоичные цифры можно было разбить на группы по три, после чего произвести замену в соответствии с таблицей 4.1.

Рассмотрим теперь **перевод чисел между двоичной и шестнадцатеричной системами**. Для этого предназначен *метод тетрад* (четверок). Суть его заключается в том, что каждой шестнадцатеричной цифре ставится в соответствие комбинация нулей и единиц по следующей таблице:

Шестнадцатеричная цифра	Двоичное представление
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Таблица 4.2. Метод тетрад

Так число $12A7_{16} = 0001\ 0010\ 1010\ 0111_2$, где ведущие нули, разумеется, необходимо отбросить. При переводе из двоичной системы в шестнадцатеричную необходимо дополнить число слева нулями так, чтобы все его двоичные цифры можно было разбить на группы по четыре, после чего произвести замену в соответствии с таблицей.

При переводе **между восьмеричной и шестнадцатеричной системами** представляется разумным выполнять эту операцию не через десятичную, а через двоичную систему счисления, используя рассмотренные выше методы.

Классификация типов данных в Object Pascal

Изучив основы представления чисел, перейдем теперь к рассмотрению типов данных, включенных в язык Object Pascal, но прежде сделаем одно замечание общего характера.

На аппаратном уровне компьютер поддерживает существенно ограниченный набор типов данных. В то же время программист, составляя программу на языке высокого уровня, использует более широкий и содержательно интерпретируемый набор типов и записывает операции над ними в символическом виде, а не в машинных командах. Заботу о реализации этих типов на основе имеющихся примитивов берут на себя разработчики компилятора. Такие новые типы иногда называют *абстрактными*, тем самым подчеркивается, во-первых, отсутствие их на аппаратном уровне, во-вторых, тот факт, что программист может отвлечься (абстрагироваться) от их происхождения и пользоваться ими, не задумываясь о способе их реализации (разумеется, настоящий программист должен знать, как все устроено изнутри).

Множество типов данных, конечно, не ограничивается типами, обеспечиваемыми языком программирования. Скажем, в задаче обработки изображений в качестве типа данных неплохо было бы иметь тип допустимых “картинок” с набором соответствующих операций. Современные языки программирования и в их числе Object Pascal имеют синтаксические средства создания и оформления абстрактных типов данных, которые мы рассмотрим в главе 6.

В целом в редакции языка Object Pascal, являющейся основой среды визуального программирования Delphi 7, на которую ориентируется данная книга, принята следующая классификация типов данных:

- простые: целые, вещественные, символьный, логический, перечислимый, тип-диапазон;
- строковые;
- указатели;
- процедурный тип;
- специальный тип `Variant`;

- структурированные – типы, создаваемые программистом с помощью средств конструирования новых типов данных: множества, массивы, записи, файлы, классы, ссылки на класс, интерфейсы.

Далее мы изучим некоторые стандартные (встроенные) типы языка.

Встроенные типы данных

Каждый встроенный тип данных характеризуется приписанным ему разработчиками языка именем (идентификатором), набором допустимых значений и синтаксической формой представления констант, а также набором операций над значениями.

Отметим, что три из них – целый, вещественный и строковый – обладают следующей особенностью: каждый тип фактически представляет собой некоторый набор концептуально близких типов, отличающихся между собой множеством допустимых значений.

Работа с целыми числами

Рассмотрим типы данных, используемые в Object Pascal для представления целых чисел, а также их параметры. В соответствии с документацией разработчика компилятора фирмы Borland, наилучшую производительность (скорость вычислений) при обработке целых значений обеспечивают типы данных `Integer` и `Cardinal`. Кроме того, реализована поддержка типов данных `ShortInt`, `SmallInt`, `LongInt`, `Int64`, `Byte`, `Word`, `LongWord`.

Рассмотрим следующую таблицу, характеризующую типы данных.

Название типа данных	Множество значений	Размер
Типы данных, обеспечивающие наилучшую производительность		
Integer	-2147483648..2147483647	32 бит
Cardinal	0..4294967295	
Дополнительные типы данных		
ShortInt	-128..127	8 бит
SmallInt	-32768..32767	16 бит
LongInt	-2147483648..2147483647	32 бит
Int64	$-2^{63}..2^{63}-1$	64 бит
Byte	0.255	8 бит
Word	0..65535	16 бит
LongWord	0..4294967295	32 бит

Внимательное изучение таблицы позволяет сделать несколько выводов.

Первый, имеются “типы-синонимы”: `Integer` – `LongInt`, `Cardinal` – `LongWord`. Вызвана такая ситуация соображениями обратной совместимости (для поддержки программ, написанных на основе предыдущих версий языка).

Второй, для размеров памяти в 1, 2 и 4 байта существуют знаковый и беззнаковый варианты типов. Надеемся, Вам уже достаточно очевиден тот факт, что конкретное значение числа определяется не только нулями и единицами, из которых оно состоит, но и способом интерпретации этой последовательности двоичных цифр.

Такое многообразие типов прежде всего связано со стремлением предоставить программисту возможность выбора наиболее подходящего под конкретную ситуацию диапазона. В самом деле, если из существа задачи вытекает, что обрабатываемые значения являются неотрицательными и не превосходят 255 (например, возраст человека в годах), то для их представления в целях экономии памяти разумно выбрать тип `Byte`. В то же время, не стоит забывать, что числа типов `Integer` и `Cardinal` обрабатываются процессором наиболее эффективно. Есть еще несколько технических тонкостей, которые могут повлиять на выбор типа данных и эффективность программы.

Конечно, саму эффективность можно понимать по-разному – эффективность по быстродействию или эффективность по затратам памяти. Более того, для специалиста по аппаратуре не является секретом тот факт, что объем используемой памяти может существенно повлиять на скорость программы. Вопросы оптимизации программ выходят за рамки данной книги[1]. Здесь мы лишь посоветуем следовать рекомендации разработчиков из фирмы Borland – везде, где представляется возможным, использовать типы `Integer` и `Cardinal`.

Перейдем теперь к вопросам практического использования рассматриваемых типов данных.

Константы целого типа записываются в языке Pascal естественным, принятым в математике, способом. Константа может иметь знак “+” или “–”, или не иметь знака. В последнем случае она считается положительной.

В языке Pascal имеются следующие *арифметические операции* над целыми числами:

Наименование	Обозначение
Сложение	+
Вычитание	–
Умножение	*
Деление (целая часть)	div
Остаток от деления	mod

Приведем примеры выполнения двух последних операций:

- `7 div 3` равно `2`,
- `1 div 2` равно `0`,
- `5 mod 3` равно `2`,
- `-7 mod 3` равно `-1`.

В языке Pascal имеется возможность применять к целым числам так называемые *битовые операции*, которые рассматривают число как последовательность двоичных разрядов.

Наименование	Обозначение
Битовое “НЕ”	NOT
Битовое “И”	AND
Битовое “ИЛИ”	OR
Битовое “ИСКЛЮЧАЮЩЕЕ ИЛИ”	XOR
Битовый сдвиг влево	SHL
Битовый сдвиг вправо	SHR

[1] Желающие могут обратиться к прекрасной книге Криса Касперски “Техника оптимизация программ. Эффективное использование памяти”, а также к документации по Intel-процессорам http://developer.intel.com

Рассмотрим принцип действия битовых операций. Поскольку эти операции работают с целым числом как с последовательностью нулей и единиц и действуют поразрядно, достаточно указать так называемые таблицы истинности операций, поясняющие, как формируется результат. Рассмотрим эти таблицы.

x	NOT x
0	1
1	0

Из всех битовых операций лишь операция NOT является унарной, т.е. имеет один аргумент. Принцип действия – во всех двоичных разрядах цифра меняется на противоположную ($0 \to 1$; $1 \to 0$).

x1	x2	x1 AND x2	x1 OR x2	x1 XOR x2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Смысл использования операций вытекает из приведенной таблицы. Операция AND возвращает 1 в очередном разряде числа тогда и только тогда, когда оба аргумента содержат в этом разряде 1. Операция OR возвращает 0 в очередном разряде числа тогда и только тогда, когда оба аргумента содержат в этом разряде 0. Операция XOR возвращает 1 в очередном разряде числа тогда и только тогда, когда аргументы содержат в этом разряде разные двоичные цифры.

Операции AND и OR чаще всего используются для того, чтобы установить (сделать равным 1), сбросить (сделать равным нулю) или протестировать (узнать значение) какой-то бит в числе. Операция XOR используется для разных “трюков”, обычно связанных с кодированием.

Рассмотрим некоторые примеры.

Возьмем числа 121 и 37.

$121_{10} = 1111001_2$

$37_{10} = 100101_2$

$$\begin{array}{r} 1111001_2 \\ \text{AND}\ \ 100101_2 \\ \hline 100001_2 = 33_{10} \end{array}$$

$$\begin{array}{r} 1111001_2 \\ \text{OR}\ \ 100101_2 \\ \hline 1111101_2 = 125_{10} \end{array}$$

$$\begin{array}{r} 1111001_2 \\ \text{XOR}\ \ 100101_2 \\ \hline 1011100_2 = 92_{10} \end{array}$$

А теперь еще один пример на операцию XOR.

```
var
  a, b: Integer;
begin
  write('Input numbers: ');
  readln(a, b);
  a := a xor b;
  b := b xor a;
  a := a xor b;
  write('a = ', a, ' ', 'b = ', b);
  readln;
```

```
end.
```

Как Вы думаете, что в ней происходит? Если запустить программу на исполнение, значения переменных a и b волшебным образом меняются местами без использования дополнительной памяти.

Операции SHL и SHR предназначены для быстрого осуществления операций умножения и деления на степень двойки. Так x SHL 3 эквивалентно умножению на 2^3, т.е. на 8, а x SHR 3 – делению нацело на 8. Заметим, что сдвиговые операции работают существенно быстрее "обычных" операций умножения и деления.

Работа с вещественными числами

Для представления вещественных чисел имеется следующий набор типов: `Real`, `Real48`, `Single`, `Double`, `Extended`, `Comp`, `Currency`.

Тип `Real` (он же `Double`) является основным, `Real48` оставлен для обратной совместимости (соответствует типу `Real`[1] в ранних версиях Object Pascal). Кроме того, введен специальный тип данных `Currency`, оптимизированный для повышения точности финансовых расчетов.

Название типа данных	Множество значений	Значащих цифр	Размер
Real48	$-2.9 \cdot 10^{39}..1.7 \cdot 10^{38}$	11-12	6 байт
Single	$-1.5 \cdot 10^{45}..3.4 \cdot 10^{38}$	7-8	4 байт
Double	$-5.0 \cdot 10^{324}..1.7 \cdot 10^{308}$	15-16	8 байт
Real	$-5.0 \cdot 10^{324}..1.7 \cdot 10^{308}$	15-16	8 байт
Extended	$-3.6 \cdot 10^{4951}..1.1 \cdot 10^{4932}$	19-20	10 байт
Comp	$-2^{63}+1..2^{63}-1$	19-20	8 байт
Currency	–922337203685477.5808.. 922337203685477.5807	19-20	8 байт

Многообразие вещественных типов также вызвано соображениями повышения эффективности программы – возможность использования наиболее подходящих типов, стремление обеспечить максимальную точность вычислений.

Константы вещественных типов могут быть записаны двумя способами: с фиксированной точкой и в так называемой экспоненциальной форме, т.е. с порядком.

Константа с фиксированной точкой – обычная запись числа (со знаком или без него) с использованием точки для отделения дробной части. Например, `-23.345` или `3.14`.

Для компактной записи очень больших или очень маленьких по абсолютной величине чисел, т.е. содержащих много значащих нулей до или после точки, используется *запись с порядком* (*в экспоненциальной форме*). Например, вместо числа `12300000` можно записать константу `1.23e+7`. Конструкция `e+7`, записываемая после числа без пробела, указывает, что значение, расположенное до символа порядка `e` (в соответствии с общими правилами можно использовать и большую букву `E`), должно быть умножено на 10 в степени 7. Отметим, что эта запись не означает операцию, т.е. преобразование константы выполняется во время компиляции, а не исполнения программы. Максимальные и минимальные значения порядков указаны в вышеприведенной таблице вещественных типов.

---

[1] Любите ли Вы рыбий жир? Нет? Тогда не удивляйтесь тому, что процессор не любит числа, имеющие размер в байтах, не являющийся степенью двойки. Определенно, 6 байт – это не то, о чем мечтает CPU.

В языке Pascal имеются следующие *арифметические операции* над вещественными числами:

Наименование	Обозначение
Сложение	+
Вычитание	–
Умножение	*
Деление	/

Заметим, что, как для целых типов, так и для вещественных не определена операция возведения в степень, которая в некоторых других языках имеет свое специальное обозначение (например, “**” в Fortran или “^” в Basic). Дело в том, что операция возведения в степень не реализована аппаратно, то есть как машинная команда, а представляет собой некоторый алгоритм вычислений. Исходя из соображений эффективности, программисту предлагается самому реализовать этот алгоритм.

Логика и логический тип данных

Рассмотрим следующее утверждение, сделанное неким студентом: “У нас в группе 5 отличников”. Как отнестись к его истинности? Во-первых, с общих позиций понятно, что утверждение может быть как истинным, так и ложным. Во-вторых, сказать что-то определенное можно, лишь обладая достоверной информацией. В данном случае в качестве таковой могут выступать данные деканата об успеваемости. То есть истинность этого утверждения, с одной стороны, проверяема, с другой, не однозначна, в отличие от большинства утверждений математических. Правда, и с ними не все так просто. Чуть дальше мы обсудим эту ситуацию на одном общеизвестном примере.

С точки зрения написания программ нас в данный момент интересуют утверждения, истинность которых зависит от некоторых условий. Самое очевидное действие, связанное с такими утверждениями – проверка. Язык программирования, конечно же, должен содержать для этого необходимые средства. Каковы они и как ими пользоваться, мы обсудим в следующей главе, а пока рассмотрим, как описать результат такой проверки.

Для работы с такими значениями многие языки программирования предоставляют специальный тип данных – *логический тип*. В языке Pascal этот тип называется `Boolean`. Логический тип имеет только два возможных значения, представленными предопределенными идентификаторами: `True` – “истина” и `False` – “ложь”.

Рассмотрим теперь следующее утверждение: “У нас в группе 5 отличников и 4 троечника”. Нетрудно видеть, что эта фраза совмещает в себе два утверждения:

- “У нас в группе 5 отличников”;
- “У нас в группе 4 троечника”.

Оба утверждения объединены операцией “и”.

Для того чтобы обрабатывать подобные соотношения, объединять утверждения в составные конструкции, в языке необходимы специальные *логические операции*.

Наименование	Обозначение
НЕ (отрицание)	NOT
И (конъюнкция)	AND

ИЛИ (дизъюнкция)	OR
ИСКЛЮЧАЮЩЕЕ ИЛИ	XOR

Важно не путать эти операции с рассмотренными ранее битовыми операциями. Если те представляли целочисленные аргументы как последовательность двоичных разрядов и работали *поразрядно*, то логические операции применимы только к двум возможным значениям операндов – “Истина” и “Ложь”.

Итак, *унарная* (то есть имеющая один аргумент) операция NOT дает результат, противоположный аргументу по значению. Все остальные логические операции имеют по два аргумента, то есть являются *бинарными*. Результат операции AND есть `True` тогда и только тогда, когда оба аргумента имеют значение `True`. Операция OR возвращает `False` тогда и только тогда, когда оба аргумента имеют значение `False`. Операция XOR возвращает `True`, если значения аргументов противоположны.

В дополнение к логическим языки программирования обычно предоставляют *операции отношения*, которые в качестве аргументов могут принимать значения произвольного порядкового типа (*порядковым* является любой тип, элементы которого можно занумеровать), результат же таких операций принадлежит логическому типу. Операции отношения в языке Pascal выглядят следующим образом:

Наименование	**Обозначение**
Больше	>
Меньше	<
Больше или равно	>=
Меньше или равно	<=
Равно	=
Неравно	<>

Важно, что сравниваться могут не только числа, но и значения других порядковых типов. Так, например, тип `Boolean` является порядковым, а выражение `False < True` истинным.

Отметим также, что для совместимости с программным обеспечением, реализованным при помощи других систем программирования, язык Object Pascal содержит логические типы данных `ByteBool`, `WordBool`, `LongBool`.

В заключение раздела небольшая логическая задача.

Рассмотрим следующее утверждение: “$2 \times 2 = 4$”. Что мы можем сказать о его истинности? Первое, что приходит в голову любому, кто хотя бы слышал о таблице умножения, – конечно же, утверждение истинно! Однако, не все так просто. В этой формулировке неявно подразумевается, что речь идет о всем привычной десятичной системе счисления, где данное утверждение трудно оспорить. Но мы то с Вами теперь более грамотны и знаем, что существуют системы счисления и с основаниями, отличными от десяти, например, с основанием три. В этой системе, как не трудно догадаться отсутствует цифра 4, а значит, результат умножения будет следующий: “$2 \times 2 = 11_3$”. Таким образом, на самом деле истинность указанного выше утверждения не так уж и бесспорна.

Конечно, можно возразить, что результат операции умножения есть одно и то же число, а от системы счисления зависит лишь форма его представления. Это тоже верно. Однако представьте себе, что Вам потребовалось создать программу калькулятор, способную работать с числами в любой позиционной системе счисления. Можно ли будет в качестве теста к этой программе использовать утверждение, указанное выше? Очевидно, нет.

Мораль всего сказанного – к любым даже самым очевидным с обыденной точки зрения утверждениям при разработке программ нужно подходить критически.

Символьная информация и символьный тип данных

Константами этого типа, имеющего имя `Char`, являются символы, обрабатываемые компьютером в стандартном текстовом режиме работы. Это набор из 256 символов, представленных в памяти в коде ASCII (American Standard Code for Information Interchange). Код символа занимает 1 байт – 1 байт = 8 бит, следовательно, имеем $2^8 = 256$ различных комбинаций. В их число входят буквы (заглавные и строчные), цифры и большой набор специальных знаков, например, так называемые символы псевдографики. Входят в число допустимых символов и русские буквы, разумеется, при наличии на компьютере специального аппаратного или программного обеспечения перекодировки стандартной таблицы символов. Заметим, что все символы упорядочены по значениям их кодов. При этом буквам присвоены такие коды, чтобы сохранялся алфавитный порядок.

Константы символьного типа могут быть записаны двумя способами. Первый способ – задание в явном виде в апострофах, например, `'a'` или `'$'`. Данный способ при вводе информации, очевидно, годится лишь для символов, представленных на клавиатуре.

Второй способ задания символьной константы – это указание ее десятичного кода, перед которым записывается специальный знак #. Например, #21. В языке Pascal отсутствуют специальные операции обработки символов, однако, символьный тип является порядковым и к константам и переменным этого типа применимы операции сравнения.

Заметим, что 256 символов – не панацея. Как быть в Китае и Японии? Для того чтобы иметь возможность представления 65536 различных символов был принят стандарт Unicode, представленный в Object Pascal типом данных `WideChar`.

Строковая информация и строковый тип данных

Обработка информации, представленной в строковом виде не менее распространена, чем численная. Мы имеем дело с названиями улиц, днями недели, именами и т.д. Конечно же, языки программирования должны предоставлять соответствующий аппарат для такого рода данных.

В первом приближении строка есть последовательность символов. Язык Object Pascal содержит несколько типов данных, позволяющих представлять строковую информацию. Самый простой из них мы рассмотрим сейчас, остальные – в главе 9.

Рассмотрим следующие объявления:

```
const
  TestStr = 'Это пример строки';
var
  s: ShortString; { Это переменная – строка }
  s1: String[30]; { Это тоже строка, ограниченной длины }
```

Первое объявление задает константу строкового типа. Как видите, в этом случае значение константы заключается в апострофы. Длина заданной таким способом строки не может превышать 255 символов. Вытекает оно из того факта, что для хранения длины используется один байт, располагаемый в самом начале строки (поэтому общий объем выделяемой памяти не может быть больше 256 байт).

Объявленная далее переменная `s` также может содержать строку до 255 символов длиной, однако в отличие от константы, в памяти для нее выделено в точности 256 байт и размер этот от длины строки никак не зависит.

В случае, когда заранее известно, что строки, с которыми придется работать в программе, не превышают по длине некоторого числа, меньшего 255, можно сэкономить и объявлять переменные так, как это сделано с переменной `s1`, указывая после имени типа максимальную длину строки в квадратных скобках. Заметьте, что при объявлении `s1` использовано название типа `String`. Дело в том, что в Object Pascal тип `ShortString` оставлен для обратной совместимости с программами, написанными на основе предыдущих версий языка. Однако устройство типа `String` существенно более сложно, поэтому пока мы ограничимся рассмотренным вариантом.

Операции над строками реализованы в виде стандартных подпрограмм. Мы также рассмотрим их в главе 9. А здесь познакомимся с одной часто употребляемой операцией, которая к тому же имеет весьма простой вид. Это операция объединения строк, называемая еще *конкатенацией*. Ее результат – строка, объединяющая строки-аргументы в порядке их следования. Обозначается эта операция знаком “+”. Например, результатом операции `'Object ' + 'Pascal'` будет строка `'Object Pascal'`.

Понятие переменной

В целом смысл понятия переменной совпадает с его общепринятой математической интерпретацией как символического имени, которому в разные моменты времени могут быть приписаны различные значения. Вместе с тем, с программистской точки зрения требуются уточнения. *Переменная* есть символическое (в виде идентификатора) обозначение некоторой области памяти, в которую в результате выполнения операторов программы помещаются значения.

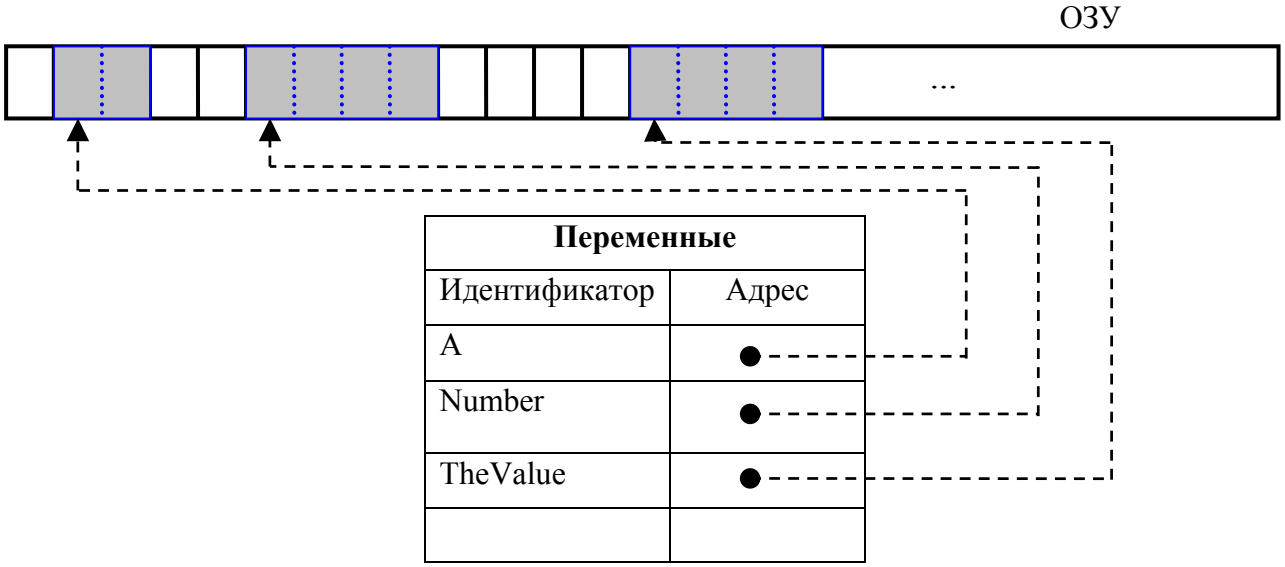


Рис. 4.2. Переменные – поименованные области памяти

Переменная характеризуется типом данных, значения которого она может хранить. Тип данных в свою очередь определяет размер области памяти, которая должна быть отведена под переменную, и способ интерпретации содержимого этой памяти. Важной особенностью многих языков программирования, в том числе и языка Pascal, является правило, согласно которому переменная в *области своего действия* может принадлежать только одному типу, т.е., например, одной и той же переменной не может быть приписано вещественное и строковое значение в процессе работы программы. В этом правиле требуют пояснения слова об области действия переменной. Коротко говоря, область действия – это блок. Более подробно этот вопрос мы рассмотрим в главе, посвященной модульному программированию.

Как мы уже обсуждали ранее, переменные объявляются в разделе объявлений, начинающемся с ключевого слова **var** (от английского variable – переменная). Объявление имеет следующий общий вид:

```
var
  <Список идентификаторов>: <Тип данных>;
```

Список идентификаторов есть список из имен переменных, которые мы хотим объявить как переменные одного типа. Сам тип указывается после двоеточия.

Тип данных – это, например, одно из тех имен встроенных типов, которые мы рассмотрели выше. Приведем примеры объявления переменных предопределенных типов:

```
var
  a, b, c: Real;
  Page_Number: Word;
  FileName: String[80];
  Key: Char;
```

Понятие константы

Мы уже знаем, как записываются константы встроенных типов. Однако в добавление к стандартному способу представления в языке Pascal имеется возможность вводить псевдонимы для констант любых типов. Эта замечательная возможность, во-первых, облегчает модификацию программы, а во-вторых, делает текст программы более понятным.

Объявления констант производятся в разделе, начинающемся с зарезервированного слова **const**. Конструкция объявления константы имеет следующий общий вид:

```
const
  <Имя константы> = <Значение>;
```

Например,

```
const
  Ln10 = 2.302585;
  SecondsInDay = 24 * 60 * 60;
  Done = True;
  Path = 'C:\MyPrograms\';
  ВМК = 'Факультет Вычислительной математики и кибернетики';
  SizeOfTable = 50;
```

Как видим, значение константы может задаваться выражением, содержащим другие константы, как явные, так и определенные ранее, а также некоторые допустимые функции от констант.

Основное правило составления таких выражений (в том числе и использования в них функций) – компилятор должен иметь все необходимое для вычисления значения константы.

Значение константы во время выполнения программы изменено быть не может.

Использование именованных констант вместо явных обеспечивает ряд преимуществ. Текст программы становится более ясным, если, например, вместо непонятно откуда взятых чисел 2.302585 или 86400, будут стоять поясняющие идентификаторы `Ln10` и `SecondsInDay`, имена которых ясно говорят, что же имеется в виду. Также текст программы может быть сокращен – без особой потери ясности – если вместо длинной часто встречающейся строковой константы записывать ее обозначение (смотри пример с константой ВМК).

Другое достоинство использования констант связано с существенным облегчением модификации программы. Допустим, некоторая программа ориентирована на обработку таблиц с максимальным числом строк, равным 50. Эта константа многократно встречается в программе, например, в алгоритмах просмотра таблицы, контроля ввода и т.п. Если по какой-либо причине необходимо сменить максимально допустимый размер таблицы, то поиск явной константы 50 в программе может вылиться в утомительное занятие. При этом не исключены и ошибочные замены, так как число 50 могло быть использовано и в других целях. Очевидно, при использовании в тексте программы вместо явной константы 50 именованной константы `SizeOfTable` достаточно будет сменить приписанное ей значение.

Типизированные константы и инициализация переменных

Описывая в предыдущем разделе способ задания именованных констант, мы опустили один довольно важный момент. Как компилятор “разбирается”, какой тип назначить указанной нами константе? Для встроенных типов данных этот вопрос может быть решен автоматически. Для целочисленных констант выбирается самый “младший” подходящий тип. Для вещественных – тип Double или Extended в зависимости от числа указанных цифр после запятой и величины константы. Однако бывают ситуации, когда программисту требуется явным образом указать тип константы. В этом случае язык Object Pascal предоставляет возможность использовать *типизированные константы*:

```
const
  Ln10: Single = 2.302585;
  SecondsInDay: Integer = 24 * 60 * 60;
```

Как видите, формат похож на объявление переменной с той лишь разницей, что после указания типа ставится знак “=” и задается значение константы. Наиболее полезны типизированные константы для типов данных, создаваемых программистом, о чем пойдет речь в главе 6.

Аналогичным способом в языке Object Pascal можно инициализировать переменные начальными значениями непосредственно при объявлении:

```
var
  Square: Single = 0;
  NumOfElements: Integer = 100;
```

В отличие от типизированных констант[1] значения инициализированных переменных, конечно же, можно изменить в теле программы.

```
begin
  Square := a * b;
```

Оператор присваивания и выражения

Одним из существенных качеств любой программы является ее общность, то есть способность выполнять заложенные в нее действия на разных наборах данных. Из этой посылки с неизбежностью вытекает тот факт, что основной объект любой программы – переменная, способная принимать и хранить различные значения. В силу важности этого понятия повторим еще раз данное выше определение: *переменная есть символическое обозначение места в памяти*. Естественно любой язык программирования должен содержать операцию установки требуемого значения в это самое место. Такая операция называется *оператор присваивания*.

В языке Pascal оператор присваивания имеет вид:

```
<переменная> := <Выражение>;
```

где "**:=**" – знак присваивания.

Примеры оператора присваивания:

```
a := 5;
x1 := (b + sqrt(sqr(b) - 4 * a * c) / (a + a);
Str := 'Это ' + 'строковое ' + 'выражение';
IsGoodStudent := AvgMark >= 4;
```

Пользуясь имеющимися в языке операциями, программист может составлять формулы или *выражения* над значениями типов данных. Существуют общие естественные правила построения выражений, которые могут включать в себя операции над константами, переменными и значениями, заданными функциями. Для указания порядка действий, отличного от порядка, определяемого приоритетами операций, используются круглые скобки.

Для лучшего понимания принципа работы оператора присваивания рассмотрим еще один вариант его записи, часто встречающийся в литературе:

```
<LValue> := <RValue>;
```

Здесь LValue – левая часть оператора присваивания – должна обязательно указывать на какую-то область памяти. Так, например, неправильны и порождают сообщения об ошибке следующие примеры:

```
5 := a; { Число 5 не связано с областью памяти! }
a + b := 7; { Результат выражения "a+b" не связан с областью памяти! }
```

Надеемся, что необходимость привязки LValue к области памяти для Вас очевидна. В самом деле, иначе было бы непонятно, куда же собственно помещать результат выполнения присваивания.

---

[1] необходимо отметить, что в версии языка Pascal, на которой основана среда разработки Borland Pascal 7.0, типизированные константы фактически являлись переменными, то есть их значения можно было менять, а инициализировать переменные при объявлении было нельзя.

RValue – правая часть оператора присваивания – в отличие от левой может содержать любые допустимые в языке выражения.

Подводя итог, укажем алгоритм выполнения оператора присваивания:

1. Вычисляется значение выражения, стоящего в правой части (RValue).
2. Определяется адрес оперативной памяти, по которому необходимо записать значение (LValue).
3. Производится запись значения, вычисленного на шаге 1, по адресу, определенному на шаге 2, в соответствии с типом данных переменной (LValue).

Преобразования встроенных типов данных

Допустим, сделаны следующие объявления переменных:

```
var
  x, a, d: Real;
  i, j: Integer;
```

Рассмотрим оператор присваивания:

```
x := a + (i div j) / d;
```

Как видим, правая часть оператора состоит из выражений разных типов. Допустима ли такая запись? Для данного примера ответ положителен. В общем же случае ответ на этот вопрос не может быть дан однозначно и требует обстоятельного рассмотрения. В данном разделе мы сделаем первые замечания по этой проблеме и разберем случай предопределенных типов данных.

Итак, как нам уже известно, тип данных определяет, помимо прочего, способ представления значений в оперативной памяти. Различия в представлении в свою очередь неизбежно ведут к необходимости каждому типу данных “иметь” собственные команды обработки значений, машинные или программно реализованные. Таким образом, если мы выполняем целочисленную операцию деления (**div**) целого числа на вещественное, то команда деления будет интерпретировать код вещественного числа как код целого, следовательно, результат операции окажется весьма далек от ожидаемого. Конечно, на практике выполнить такую операцию не удастся, поскольку компилятор выдаст сообщение об ошибке вида: “Тип операнда не соответствует операции”. В свою очередь такая диагностика возможна благодаря наличию типизации в языке Pascal.

Попытаемся теперь сложить целое и вещественное число. Вам ничего не кажется подозрительным? Как мы указали выше, операция сложения имеет одно и то же обозначение как для целых так и для вещественных аргументов: “+”. Как же ее будет рассматривать компилятор в данном случае? Очевидно, что операцию нельзя выполнять как сложение целых чисел – результат окажется неверным. Возможно то же решение, что и в случае с делением – выдать сообщение об ошибке. Однако это противоречит естественному порядку, принятому в математике, – мы выполняем операции сложения чисел, не задумываясь о соответствии типов. В данном случае разумным представляется преобразовать целое число к вещественному и выполнить сложение над вещественными аргументами. Именно так и делается на практике: компилятор осуществляет автоматическое преобразование или, как еще говорят, приведение типов и выбирает соответствующую команду обработки.

В общем случае преобразование выполняется к “старшим” типам, то есть к вещественным и с большим диапазоном значений. Таким образом, в выражении могут смешиваться целые и вещественные типы всех разновидностей, но программист в этом случае должен быть очень внимателен и понимать, в каком порядке будут осуществлять преобразования и собственно выполнение операций, иначе результаты расчетов могут его сильно удивить.

Отметим один существенный факт: в операторе присваивания в правой части может стоять целое выражение, а в левой вещественная переменная, но не наоборот. Для приведения вещественного аргумента к целому типу можно воспользоваться функциями явного преобразования: Round(x) и Trunc(x).Обе функции преобразуют вещественное число к типу Longint, причем первая округляет число, а вторая просто отбрасывает дробную часть.

Для нечисловых типов данных аппарат автоматического преобразования типов отсутствует, что, конечно, вполне понятно. В выражениях нельзя смешивать числа, логические и символьные значения, т.е., например, запись x := 2 + ’2’ недопустима.

С другой стороны, существует возможность явного преобразования символов и строк в числа и наоборот. Для символов это функция `Ord()`, которая возвращает десятичный порядковый номер (код ASCII) символа, и функция `Chr()`, которая по десятичному коду выдает соответствующий символ.

Текстовые строки, которые имеют вид, совпадающий с синтаксическим определением числовой константы, преобразуются с помощью процедур `Val` и `Str`, включенных в библиотеку System. Процедуры, также как функции, являются заранее разработанными библиотечными программами и отличаются от функций способом обращения к ним. Оператор вызова процедуры в языке Object Pascal записывается очень просто. Это имя процедуры, за которым в круглых скобках следует список параметров. В дополнение к рассмотренным подпрограммам, ставшими стандартными со времен Borland Pascal, язык Object Pascal содержит ряд новых процедур и функций: FloatToStr, StrToFloat, DateToStr,… Их описание может быть без труда найдено в справочной системе, здесь мы его дублировать не будем.

Некоторые стандартные математические функции

Обычно языки программирования содержат лишь необходимый минимум выразительных средств. В дополнение к ним создатели сред разработки добавляют библиотеки функций, содержащие эффективные реализации некоторого множества часто используемых операций. В их число входят, например, подпрограммы вычисления элементарных математических функций, подпрограммы обработки строк и другие.

В данном пункте мы приведем некоторые стандартные математические подпрограммы, входящие в состав библиотеки System. Особенность этой библиотеки состоит в том, что она всегда автоматически подключается редактором связей к собираемой программе пользователя, т.е. использование подпрограмм из нее не требует подключения в секции uses.

Функция	Описание
Abs(x)	Взятие модуля
Sqr(x)	Возведение в квадрат
Sqrt(x)	Извлечение квадратного корня

Exp(x)	e^x
Ln(x)	Логарифм натуральный от x
Sin(x)	Синус
Cos(x)	Косинус
ArcTan(x)	Арктангенс
Int(x)	Целая часть
Frac(x)	Дробная часть

Заметим, что в число функций не входит, например, функция тангенс и ряд других элементарных функций. Их значения вычисляются с помощью формул над приведенными в таблице функциями. В частности, показательная функция (a^x) может быть реализована как Exp(Ln(a)*x).

Система программирования Borland Delphi содержит модуль Math, реализующий многие другие математические функции (арифметические, тригонометрические, финансовые, статистические). Эти и некоторые другие стандартные подпрограммы мы будем рассматривать по мере необходимости.

Простейшие средства ввода и вывода информации. Стандартный ввод-вывод в консольном приложении

Программирование ввода и вывода информации, а в общем смысле организация взаимодействия с пользователем, – одна из важнейших задач для разработчика. Уже в первом примере книги мы видели, что число операторов, обеспечивающих обмен информацией, превышало число операторов расчетных. Это, конечно, не всегда так, но программа, предназначенная для коммерческого использования, как правило, содержит весьма емкий блок, реализующий интерфейс с пользователем. В настоящее время подавляющее большинство таких программ, используют интерфейс графический (GUI – graphic user interface), основанный на понятии окна, из-за чего сами программы обычно называются оконными приложениями. Построение подобного интерфейса представляет собой тему для отдельной книги, таких книг немало (откройте, например, любую книгу по системе визуального программирования Borland Delphi). Однако существует класс задач, в которых возможности графического интерфейса и оконных приложений не нужны. Так, например, к этому классу относятся большинство расчетных программ, системные утилиты. В таких случаях в качестве устройства вывода используется текстовая консоль. В данном разделе мы познакомимся с тем, как программируется простейший ввод/вывод в текстовом режиме в консольных приложениях. При этом мы узнаем, как запрашивать от пользователя данные и выводить на экран результаты. Желающие организовывать полноценный интерфейс в текстовом режиме с использованием различных цветов, очисткой экрана, позиционированием курсора и т.д. могут обратиться к справочной системе MSDN[1].

Подпрограмма ввода с клавиатуры имеет вид:

```
Read[(<список переменных>)];
```

или

[1] MSDN – Microsoft Developer Network – библиотека справочной информации по продуктам Microsoft.

```
Readln[(<список переменных>)];
```

При этом список переменных может содержать переменные любых стандартных типов языка Pascal.

Данная подпрограмма является весьма сложной и обеспечивает:

- отображение нажимаемых в процессе ввода клавиш-символов на экране дисплея;
- синтаксический контроль вводимой информации;
- преобразование данных из символьного (внешнего) представления во внутренний код, соответствующий типу записанной в списке ввода переменной.

Схема работы Read следующая:

- программа переходит в режим ожидания ввода с клавиатуры;
- пользователь осуществляет ввод информации, отделяя значения переменных пробелами или нажатием клавиши Enter;
- признаком завершения ввода является нажатие клавиши Enter при одновременном исчерпании списка ввода (лишние значения будут проигнорированы);
- начинается обработка введенной информации: преобразование из символьного представления к типу данных соответствующей переменной в списке ввода;
- если введенные данные некорректны, например, производится попытка ввести в целую переменную символы, недопустимые в представлении целых чисел, работа всей программы аварийно завершается и выдается сообщение об ошибке времени исполнения (например, “Invalid numeric input” – “Неверный численный ввод”).

Необходимо обратить внимание на один существенный момент в работе подпрограммы Read – последний символ Enter, которым заканчивается ввод информации, Read *не считывает*. Для этого предназначена вторая форма подпрограммы Readln.

В соответствии с представленным выше форматом подпрограммы Read и Readln могут не иметь списка ввода. При этом Read фактически не выполнит никаких действий, а Readln будет ожидать нажатия клавиши Enter (если она не была нажата ранее и не считана).

Подпрограмма вывода на дисплей имеет вид:

```
Write[(<список вывода>)];
```

или

```
Writeln[(<список вывода>)];
```

В список вывода входят *выражения*, разделенные запятыми. Значения выражений преобразуются к строковому виду и выводятся на экран. Имеется возможность форматирования выводимых значений такая же, как и в процедуре Str. Например, предложение

```
Write(Sin(Pi/4):5:3);
```

выдаст результат в виде 0.707, в то время как предложение

```
Write(Sin(Pi/4));
```

выдаст на экран число в стандартной экспоненциальной форме 7.0710678119E-01.

Если программист ошибся в выборе формата и, например, указал общее число позиций, меньшее чем число значащих цифр, то программа сама исправит положение и напечатает число в стольких позициях, сколько необходимо для размещения числа.

Подпрограмма `Writeln` помимо вывода значений из списка вывода обеспечивает перевод курсора в начало следующей строки.

Заметим, что значения всех рассмотренных нами встроенных типов могут быть введены и выведены указанными подпрограммами. Исключение составляет ввод булевских значений. `Read` не предназначен для чтения констант `true` и `false`. Сообщение об ошибке будет выдано еще на этапе компиляции. В то же время `Write` выводит эти константы в строковом виде.

Рассмотрим примеры.

```
{ ========================================================================= }
{ Пример 4.1 }
{ Вывод на дисплей десятичного кода ASCII требуемого символа }
Program DisplayCode;

{$APPTYPE CONSOLE}

var
  Symbol: Char;
begin
  Write('Символ: ');
  Readln(Symbol);
  Writeln('Код: ', Ord(Symbol));
  Readln;
end.
{ ========================================================================= }
```

Эта программа запрашивает ввод символа, читает его и печатает код. На экране диалог выглядит следующим образом:

```
Символ: k
Код: 107
```

```
{ ========================================================================= }
{ Пример 4.2 }
{ Значения элементарных функций }
Program Functions;

{$APPTYPE CONSOLE}

var
  x: Real;
begin
  Write('Аргумент? ');
  Readln(x);
  Writeln('Sin(',x:5:3,') = ', Sin(x):5:3);
  Writeln('Cos(',x:5:3,') = ', Cos(x):5:3);
  Writeln('Exp(',x:5:3,') = ', Exp(x):5:3);
  Readln;
end.
{ ========================================================================= }
```

Результат работы программы:

```
Аргумент? 3.45
Sin(3.450) = -0.304
```

```
Cos(3.450) = -0.953
Exp(3.450) = 31.500
```

Выводы

Нам представляется, что данная глава очень важна не только с практической, но и с теоретической точки зрения. Тем из Вас, кто выберет программирование в качестве своей профессии, предстоит изучить многие языки программирования, познакомиться с новыми средствами и технологиями, и, быть может, даже увековечить свое имя в зале славы их разработчиков. Совершенно ясно, что изучение нового материала не должно занимать месяцы, у Вас просто не будет так много времени в нашем динамично меняющемся мире. Содержание главы представляет собой тот базис, те фундаментальные понятия и принципы, на основе которых, как правило, и строятся языки программирования.

5. Структурное программирование и операторы языка Object Pascal

Раз дощечка, два дощечка – будет лесенка.
Раз словечко, два словечко – будет песенка.
Слова из песни

В предыдущей главе мы рассмотрели базовые составляющие языка программирования высокого уровня Object Pascal: алфавит, ключевые слова, правила формирования идентификаторов, выражений и операторов, структуру программы. Также обсудили такие важнейшие элементы любой программы, как переменная и оператор присваивания. Без правильного их понимания ни одну сколько-нибудь серьезную программу написать невозможно (да и считаться программистом на наш взгляд тоже). Наконец, мы изучили простейшие средства обмена информацией между программой и ее пользователем – операторы ввода и вывода. Используя полученные знания, мы совершили пару маленьких шагов на пути к сияющим вершинам программистского мастерства – написали несколько элементарных программ.

Однако чтобы продвинуться дальше, нам потребуется нечто большее, чем простое изучение элементов языка. Как овладев, пусть даже в совершенстве, английским языком, Вы не станете автоматически Уильямом Шекспиром, так и изучив досконально язык программирования, пусть даже такой мощный как Object Pascal, Вы не подниметесь выше уровня написания программ “для себя”. Процесс создания сложных, больших программных комплексов включает в себя и радость творчества и рутину кодирования и мучительные поиски ошибок, однако все это будет бесполезной тратой сил и времени, если сам процесс не будет должным образом организован. Будучи Левшой можно “на коленке” создать невероятно красивую, оптимизированную, оригинальную, полезную (подставьте любой эпитет в превосходной степени) программу. Но! Исходный код программных комплексов уровня операционной системы содержит десятки миллионов строк. Никому не под силу не только разработать такую систему в одиночку, но и удержать в голове все необходимые решения, которые будут приняты в процессе создания. Более того, написать код подобного объема, не придерживаясь определенных правил и стандартов, не используя типовые приемы, невозможно. Нужны организационные и технологические решения, которые позволили бы справиться с этой фантастической сложностью. Одним из таких решений является концепция структурного программирования, с которой мы познакомимся в настоящей главе. Но прежде обсудим, что же скрывается за самим понятием *технология программирования*.

Программирование как технологический процесс. Понятие технологии программирования

Еще в первой главе мы перечислили ряд вопросов, без получения ответов на которые невозможна разработка сколько-нибудь серьезной программной системы. Вот еще некоторые.

Из чего состоит сам процесс создания сложного программного комплекса? Как организована работа коллектива программистов, процессы анализа, проектирования, программирования, отладки, модификации. И почему мы употребляем здесь слово процесс? Только ли для того, чтобы подчеркнуть сложность или по какой-то другой причине?

Кажущаяся легкость написания учебных “игрушечных” программ порождает ложное впечатление о наличии простых ответов на перечисленные вопросы. Однако на поверку выясняется, что простотой здесь и не пахнет. Для того чтобы это продемонстрировать, рассмотрим пример из любой давно существующей отрасли, например, судостроения. Давным-давно индейцы Северной Америки не испытывали никаких проблем в этой области. Так, для того, чтобы построить пирогу, не требовалось никаких специальных методов и уж конечно никакой формальной теории. Казалось бы, в чем проблема? Топор, дерево, немного усилий, и пирога способна принять на борт Ваших соплеменников. А вот удастся ли на этой пироге достичь Британии? Что с ней будет в случае возникновения шторма? А что, если в пирогу врежется акула? А если ее подожгут? И это только часть вопросов, за каждым из которых своя проблема, требующая решения.

Поместите теперь мысленно в эту пирогу себя. Вот Вы беретесь за весла… Какой такой двигатель? Навигационные приборы? Телевизор? Вы не можете путешествовать без компьютера? Вы чувствуете недостаток сотовой связи? Всего этого не сделать топором. Тут понадобятся некоторые знания и инструменты. Иногда даже понадобится много рабочий силы и оборудования, быть может, целые заводы. Добавьте сюда проектирование, разработку, спуск на воду и ходовые испытания корабля, которые в настоящее время представляют собой очень сложный, но продуманный, отлаженный и детально расписанный процесс, *технологический* процесс. Для решения большинства производственных задач существуют свои технологии. Давно ушли в прошлое времена, когда изготовление чего-либо выполнялось одним человеком по наитию. Сегодня уже невозможно представить ситуацию, когда рабочие приходят утром на завод и голосованием решают, кто вырубает топором корпус, кто ставит мачту, кто приделывает парус и тем более, как именно это делать. Получившееся корыто (простите, корабль!) вряд ли вообще поплывет. Технологические процессы, в которых все четко определено, заняли ведущее место на производстве.

Итак, что же такое технология? *Технология*[1] *– совокупность производственных процессов в определенной отрасли производства, а также научное описание способов производства.*

Вернемся теперь к программированию. В 60-х–70-х годах XX века актуальным считался вопрос о сущности программирования, о том, что это такое: наука, техника или искусство? Одна из самых известных книг по программированию, написанная в то время и выдержавшая множество переизданий (настоятельно рекомендуем нашим читателям внести ее в перечень ближайших пополнений своей домашней библиотеки), так и называется: “Искусство программирования” [9, 10]. Уже тогда людям было ясно, что ввиду роста сложности решаемых при помощи компьютера

[1] С.И. Ожегов. Словарь русского языка. - М.: Советская энциклопедия, 1975

задач неимоверно возрастает стоимость разработки программ. Причем, если стоимость аппаратуры растет умеренными темпами, а иногда и вовсе падает, то со стоимостью разработки программ ничего поделать не удается. Именно тогда вопросы о том, что такое программирование и как оптимизировать процесс разработки, вышли на первый план.

Для того чтобы ответить на эти вопросы потребовалось определить, куда же уходят средства, за счет чего возрастает стоимость разработки программных систем? Рассмотрим кратко следующую схему:

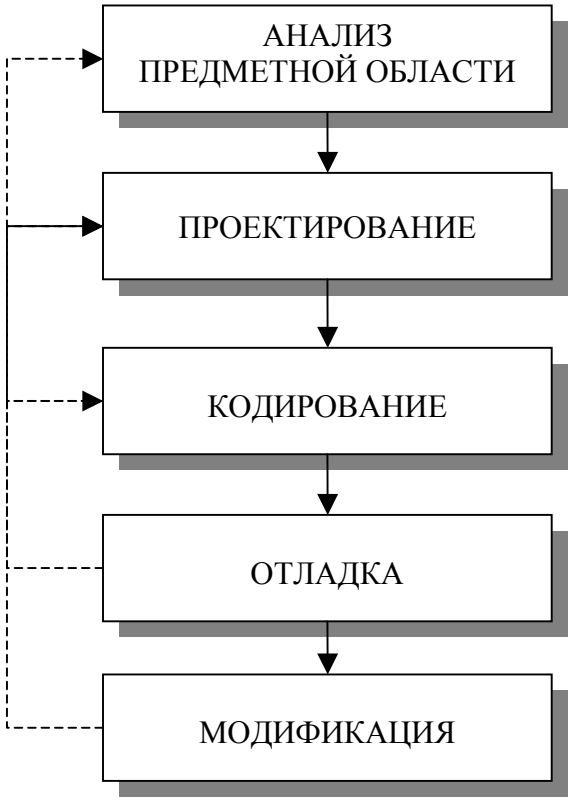


Рис. 5.1. Схема процесса создания программной системы

Данная схема представляет собой последовательность стандартных этапов, которые необходимо выполнить при создании любой программной системы. Именно на этих этапах и возникают существенные финансовые затраты. Для их оптимизации необходимо было понять, что программирование есть обычный *технологический процесс*, по характеру возникающих проблем мало чем отличающийся от, скажем, строительства дома или корабля.

Для сокращения затрат необходимо было конкретизировать схему, упорядочить действия, выполняемые на каждом этапе, разработать методы решения возникающих на разных этапах проблем. В довершении ко всему, схема подразумевает возвраты назад (циклы), в тех случаях, когда обнаруживается ошибка предыдущего этапа.

В результате кропотливой работы большого количества специалистов на каждом этапе и подэтапе возникли и продолжают появляться и совершенствоваться специальные *технологии*, позволяющие решать задачи в заданные сроки с заданным качеством.

Итак, *технология программирования* – совокупность методов, приемов и средств для сокращения стоимости и повышения качества разработки программных систем.

В любой серьезной компании, занимающейся разработкой программного обеспечения, на каждом этапе рассмотренной схемы применяется большое количество разных технологий. Сразу

отметим, что та часть организационных моментов, которая относится к обеспечению взаимодействия множества людей в процессе работы над программным комплексом, а также к построению самого процесса разработки программного обеспечения от постановки задачи до получения результата в виде готовой программной системы, выходит за рамки данной книги. Мы уделим этому минимально необходимое внимание, а серьезно изучать будем ту часть технологий программирования, которая непосредственно связана с процессом превращения имеющегося набора алгоритмов в программную реализацию. Положения рассмотренных в книге технологий – структурного, модульного и объектно-ориентированного программирования – стали к настоящему моменту общепринятыми, устоялись, перешли в разряд базовых. Надеемся, что скоро и Вы согласитесь с их фундаментальным значением.

Концепция структурного программирования

Возникновение концепции структурного программирования [13, 28, 41] связывается с именем известного голландского ученого Э. Дейкстры[1] – в 60-х годах прошлого века он сформулировал основные ее положения.

Принцип, на котором зиждется *технология структурного программирования* – фундаментальная научная и техническая идея о выделении множества базисных элементов, с помощью которых можно выразить (из которых можно собрать) любой объект из некоторого широкого набора. Так же как из ограниченного числа деталей детского конструктора можно при наличии некоторой фантазии построить весьма большое количество “изделий”, так же как суперпозицией функций из базисного множества можно выразить любую функцию из некоторого пространства (например, все булевы функции могут быть построены на основе полной системы из конъюнкции и отрицания), так и программы, как было доказано, можно создавать, используя лишь небольшое число базисных алгоритмических конструкций.

Итак, основной принцип технологии структурного программирования гласит: Для любой *простой* программы можно построить функционально эквивалентную ей *структурную* программу, т.е. программу, сформированную на основе фиксированного базисного множества, включающего *структуру последовательного действия*, *структуру выбора* одного из двух действий и *структуру цикла*, то есть многократного повторения некоторого действия с проверкой условия остановки повторения.

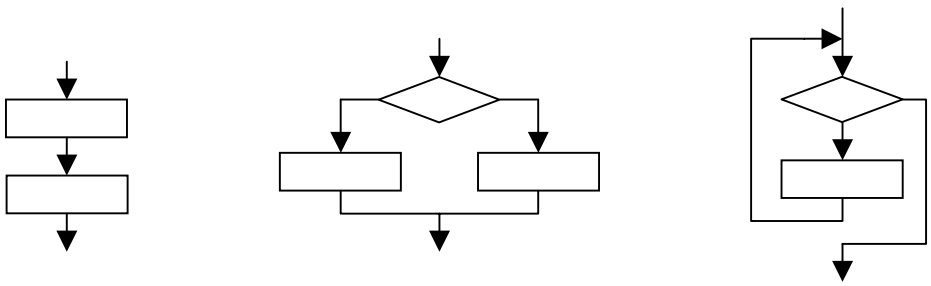


Рис. 5.2. Блок-схемы базисных алгоритмических конструкций

[1] Эдсгер Вайб Дейкстра (Edsger Wybe Dijkstra) (1930 – 2002) – голландский ученый, доктор компьютерных наук, профессор. Один из авторов концепции структурного программирования. Активно участвовал в разработке языка программирования Algol, автор первого компилятора Algol 60. Лауреат премии Тьюринга. Разработал многие ставшие классическими алгоритмы. Идеи Э. В. Дейкстры оказали огромное влияние на развитие компьютерной индустрии.

На рисунке представлено изображение указанных алгоритмических конструкций в виде блок-схем. Здесь прямоугольник обозначает обобщенное действие, ромб – проверку условия, стрелки – переход от одного действия к другому.

Под *простой* программой в данном случае понимается программа, имеющая ровно один вход и один выход по управлению, такая, что через все ее функциональные блоки проходит путь от входа до выхода.

Изложенный принцип представляет собой *теорему о структурировании*. Ее точная формулировка и доказательство не входят в нашу задачу, желающие могут обратиться к монографии [13].

Базисные алгоритмические конструкции обладают важным свойством – они в точности удовлетворяют определению *простой* программы, то есть имеют один вход и один выход, что обеспечивает возможность осуществлять их суперпозицию. Любая из трех структур может быть подставлена в остальные или в саму себя. Например, на рисунке 5.3 произведена подстановка цикла в выбор.

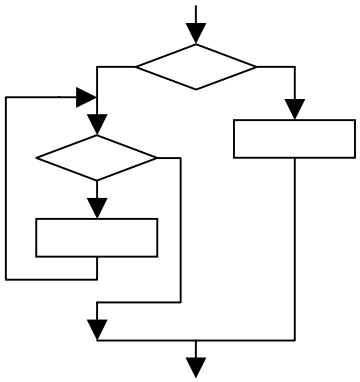


Рис. 5.3. Суперпозиция алгоритмических конструкций

Таким образом, центральный технологический принцип структурного программирования состоит в том, что формулировку алгоритма и его запись в виде программы рекомендуется выполнять на основе базиса из трех алгоритмических конструкций, применяя при необходимости их суперпозицию. Результатом последовательного применения этого принципа будет более ясная структура программы (в особенности, если использовать выделение структурных уровней с помощью отступов), что, несомненно, облегчит поиск в ней ошибок и упростит ее модификацию.

Ориентация языка программирования на некоторую технологию означает включение в него соответствующих выразительных средств. Язык Pascal создавался как язык структурного программирования, поэтому в него включены операторы, реализующие рассмотренные структурные примитивы. Более того, для большего удобства эти примитивы имеют несколько вариантов реализации.

Прежде чем перейти к изучению соответствующих операторов языка Pascal, сделаем два важных замечания общего плана.

Во-первых, концепция структурного программирования носит универсальный характер и не связана с конкретным языком. Например, в первом широко известном языке программирования высокого уровня – языке Fortran (точнее в его ранних версиях) отсутствуют операторы структурного программирования. Но это совсем не значит, что данная технология не может быть применена для составления программ на языке Fortran. Просто требуется выбрать способ записи базисных структур на основе имеющихся в языке операторов и придерживаться этого способа при разработке программ.

Во-вторых, необходимо иметь в виду: использование технологии структурного программирования (да и любой другой) при разработке программ не самоцель, а средство, средство, применяемое в конечном счете для уменьшения затрат на их создание. Таким образом, на практике поступать следует в точном соответствии с общеизвестной истиной, гласящей, что из любого правила бывают исключения, – если точное следование канонам технологии ухудшает, каким угодно образом, текст программы, каноны нужно нарушить. В структурном программировании таким каноном, прямое следование которому способно в некоторых ситуациях принести больше вреда, чем пользы является принцип: “один вход, один выход”. В дальнейшем мы рассмотрим ситуации, в которых отступление от этого принципа является вполне оправданным.

Программирование последовательности действий

Реализация *структуры последовательного действия* чрезвычайно проста. В языке Pascal, как и в большинстве других языков программирования, принято естественное правило выполнения операторов в порядке их физического следования в тексте программы. Все приведенные в предыдущей главе фрагменты программ как раз и являются примерами использования этой алгоритмической конструкции. Как Вы уже могли по этим примерам заметить, в языке Pascal операторы отделяются друг от друга точкой с запятой.

И все-таки, несмотря на простоту первой из базисных структур один важный момент, необходимость и полезность которого в полной мере мы проясним дальше, нужно обсудить именно здесь. Как было указано выше, каждая из трех базисных алгоритмических конструкций может быть подставлена в качестве элемента в любую другую. То есть одна структура последовательного действия может быть элементом другой структуры последовательного действия. Вполне очевидно, что подобное разбиение последовательности действий на элементы является исключительно логическим, однако Pascal содержит средства, позволяющие такое разбиение осуществить в тексте программы явно. Помимо прочего оно является указанием компилятору при осуществлении синтаксического анализа текста программы рассматривать данный блок как один оператор, что часто бывает важно для правильной интерпретации программы.

Итак, для того чтобы оформить структуру последовательного действия в отдельную синтаксическую конструкцию, необходимо использовать так называемые *операторные скобки*: ключевые слова **begin** и **end**. При этом сама такая конструкция называется *составным оператором*.

```
Write('Введите координаты прямоугольника');
begin
  Read(x1, y1);
  Readln(x2, y2)
end;
{дальнейший код, например рисование}
```

В приведенном примере операторы ввода синтаксически выделены в отдельный составной оператор. Заметим, что после последнего оператора группы, то есть перед ключевым словом **end**, точку с запятой можно не ставить (однако из практических соображений пользоваться этим исключением из правил мы не рекомендуем).

Программирование выбора

Структура выбора является без сомнения самой важной из базисных алгоритмических конструкций. Лишь чрезвычайно простые программы могут быть написаны без ее использования. *Структура последовательного действия* естественна, как отображение того факта, что любой алгоритм есть последовательность некоторых шагов. А *структура цикла*, при наличии в языке программирования оператора безусловного перехода может быть реализована, как будет показано далее, на основе структуры выбора. В данном разделе мы рассмотрим практические ситуации, в которых возникает необходимость в использовании этой структуры, изучим имеющиеся в языке Pascal *операторы выбора*, реализующие структуру выбора в ее исходном виде или в виде некоторых расширений, повышающих удобство программирования.

Выбор из двух вариантов

Рассмотрим одну из типичных задач, возникающих при реализации взаимодействия программы с пользователем, – обеспечение контроля вводимых данных. Допустим, что пользователь в ответ на запрос программы должен ввести целое число, обозначающее день месяца. Число это естественно должно находится в интервале от 1 до 31. Допустим также, что программа написана правильно и, помимо предложения ввести число, выводит на экран подсказку о правильном диапазоне. Очевидно, что даже в этом случае пользователь при вводе может ошибиться и набрать неверное число или вовсе нажать клавиши с символами, отличными от цифр. Таким образом, данные, получаемые от пользователя необходимо проконтролировать и при наличии ошибки как минимум выдать надлежащее сообщение, а еще лучше предложить ввести число заново.

Решение этой задачи в общем виде требует использования цикла, поэтому пока мы рассмотрим минимальный и не совсем верный с практической точки зрения вариант, в котором возможные ошибки ввода только проверяются. Словесное описание данного алгоритма можно дать в следующей форме (такую запись иногда называют *псевдокодом*):

```
вводим данное
если введенное данное не является целым числом, то
  выдаем сообщение об ошибке,
иначе,
  если введенное число – целое, но не входит в интервал от 1 до 31, то
    выдаем сообщение об ошибке,
  иначе
    выполняем необходимые действия
завершаем работу
```

Приведенная последовательность действий содержит характерную алгоритмическую конструкцию вида “если… то… иначе…”. Очевидно, что эта конструкция в точности соответствует *структуре выбора*. Именно в таком виде она и реализована в языке Pascal (естественно на английском языке): “if… then… else…”.

Формальная запись *оператора выбора* **if** выглядит следующим образом:

```
if <условное выражение> then
  <оператор1>
else
  <оператор2>;
```

При этом условное выражение, по которому собственно осуществляется выбор должно иметь *булевский* тип. Если результат вычисления выражения – **true** (истина), то выполняется оператор1, в противном случае оператор2.

При использовании условного оператора **if** нужно иметь в виду два важных *синтаксических* правила. Во-первых, перед ключевым словом **else**, то есть после оператора, записанного в секции **then**, нельзя ставить точку с запятой. Компилятор в таком случае выдаст ошибку. Во-вторых, согласно приведенному формату записи условного оператора каждое из двух действий в нем *должно* состоять *ровно* из одного оператора. Что же делать, если по схеме алгоритма действий должно быть много? Решение было указано в предыдущем разделе – необходимо использовать *составной оператор*. В результате получится, например, следующее:

```
if <условное выражение> then
begin
  <оператор1>;
  <оператор2>;
end
else begin
  <оператор3>;
  <оператор4>;
  <оператор5>;
end;
```

Вернемся к поставленной в начале раздела задаче контроля ввода числа. В языке Pascal предусмотрены специальные средства для выполнения автоматического контроля операций ввода и вывода. В частности контролируется соответствие типа вводимой информации, и при необходимости можно установить контроль за попаданием введенных чисел в некоторый диапазон. Будут средства контроля включены в код исполняемого модуля или нет, зависит от *директив* компилятора. Так проверка операций ввода и вывода задается директивами {$I}, {$IOCHECKS}, а проверка на диапазон – директивами {$R} и {$RANGECHECKS}. По умолчанию проверка ввода-вывода включена, проверка на диапазон выключена.

Использование стандартных средств, однако, приводит к тому, что при возникновении ошибок, во-первых, выдаются сообщения вида “Runtime error 106 at 00403A30”, которые мало о чем скажут пользователю программы (а скорее просто напугают его), а во-вторых, автоматически обрывается выполнение программы, что и вовсе является недопустимым. Неверный ввод не является критическим для работоспособности программы и не должен приводить к ее завершению. Грамотно написанная программа должна сама “перехватывать” ошибочный ввод и выдавать сообщения, из которых можно понять, что именно произошло и, что в результате делать дальше.

Таким образом, в примере, приведенном ниже, мы не используем директиву {$R} и отказываемся от действия директивы {$I}. Взамен, для того чтобы узнать, чем закончилась последняя операция ввода-вывода, мы используем функцию IOResult. Если последняя операция ввода-вывода завершилась успешно, функция возвращает нуль, иначе значение отличное от нуля.

```
{ ========================================================================== }
{ Пример 5.1 }
{ Контроль вводимой информации }
Program Control;

{$APPTYPE CONSOLE}

var
```

```
  Day: Integer;
begin
  {$I-} { отключили автоматическую проверку ввода-вывода }
  ReadLn(Day);
  if (IOResult <> 0) then { ошибка ввода }
    WriteLn('Ошибка 1: Недопустимые символы в целом числе')
  else
    if (Day < 1) or (Day > 31) then
      WriteLn('Ошибка 2: День месяца вне диапазона 1..31')
    else begin
     { Дальнейший текст программы }
    end;
end.
{ ========================================================================== }
```

На практике часто встречается ситуация, когда содержательные действия необходимы *только* при выполнении некоторого условия, в противном же случае ничего делать не требуется. В этой ситуации в языке Pascal можно использовать так называемый *пустой оператор*, то есть:

```
  if <условное выражение> then
    <оператор1>
  else
    ; { это пустой оператор, "ничего не делать" }
```

Однако такая конструкция, оправданная синтаксически, является явно избыточной с точки зрения здравого смысла. Поэтому оператор **if** имеет сокращенную форму (иногда ее называют *неполной альтернативой*), в которой секция **else** опускается вместе со своим ключевым словом:

```
  if <условное выражение> then
    <оператор>;
```

Заметим, что теперь после оператора в секции **then** точка с запятой необходима.

В качестве примера использования сокращенной формы оператора **if** рассмотрим программу поиска корня уравнения $ax + b = 0$. Известно, что данное уравнение имеет ноль корней, если $a = 0$ и $b \neq 0$, бесконечно много корней, если $a = 0$ и $b = 0$, один корень, если $a \neq 0$.

```
{ ========================================================================== }
{ Пример 5.2 }
{ Решение линейного уравнения }
Program LinearEquation;

{$APPTYPE CONSOLE}

var
  a, b: Real;
begin
  WriteLn('Введите коэффициенты уравнения ax+b=0: ');
  ReadLn(a, b);
  if (a = 0) and (b = 0) then
    WriteLn('Корней бесконечно много');
  if (a = 0) and (b <> 0) then
    WriteLn('Корней нет');
  if (a <> 0) then
    WriteLn('x = ', -b/a);
end.
```

```
{ ========================================================================== }
```

Указанные три варианта можно анализировать в разном порядке, в том числе и с использованием условного оператора в виде полной альтернативы. Соответствующий вариант программы мы предлагаем написать Читателю самостоятельно.

Вернемся еще раз к примеру 5.1. В представленном коде была произведена суперпозиция двух структур выбора. Чтобы обеспечить строгое следование правилу структурного программирования “один вход – один выход”, весь основной текст программы нам пришлось поместить в составной оператор, что и показывает комментарий. Однако с чисто практической точки зрения это не слишком удобно. Отчасти потому, что при соблюдении правила отступов, текст программы постоянно сдвигается вправо, что ухудшает ее читабельность. Но главное, сильная синтаксическая связь основного текста с интерфейсной частью (часть программы, в которой осуществляется взаимодействие с пользователем) неоправданна, так как они часто модифицируются независимо. Поэтому здесь вполне допустимо отойти от “структурного экстремизма” и допустить “обрывы” по управлению. Поскольку в приведенной реализации выполнение программы прекращается при ошибочном вводе числа, такой “обрыв” управления мы можем реализовать специальной процедурой останова `Halt`. Кроме того, условные операторы в этом случае разумно использовать в сокращенной форме. Получится вот что:

```
{ ========================================================================== }
{ Пример 5.3 }
{ Контроль вводимой информации – вариант 2 }
Program Control2;
{$APPTYPE CONSOLE}

var
  Day: Integer;
begin
  {$I-} { отключили автоматическую проверку ввода-вывода }
  ReadLn(Day);
  if (IOResult <> 0) then { ошибка ввода }
  begin
    WriteLn('Ошибка 1: Недопустимые символы в целом числе');
    Halt;
  end;
  if (Day < 1) or (Day > 31) then
  begin
    WriteLn('Ошибка 2: День месяца вне диапазона 1..31');
    Halt;
  end;
  { Дальнейший текст программы }
end.
{ ========================================================================== }
```

В приведенных в этом разделе примерах мы уже несколько раз использовали операторы выбора, условные выражения в которых содержали логические операции **and** и **or**. Как известно для получения результата в таких выражениях не обязательно подсчитывать значения обоих операндов. Например, если вычисленный первым операнд операции **or** имеет значение **true** , то вне зависимости от значения второго операнда результат всей операции также равен **true**. Если принять такой подход неполного вычисления булевского выражения, то это не только сократит время работы программы, но и обеспечит ее более компактную и ясную запись.

Рассмотрим пример. Допустим, некоторая программа использует в качестве входной информации целое число, которое должно быть делителем (нацело) другого целого числа (пусть, например, речь идет о числе равноправных потребителей некоторого ресурса). Фрагмент программы, обеспечивающий контроль вводимого числа (количества потребителей), выглядит следующим образом:

```
{ ========================================================================== }
{ Пример 5.4 }
{ Неполное вычисление логического выражения }
Progam Calculation;

{$APPTYPE CONSOLE}

const
  Resourses = 28;
var
  Consumers: Integer;
begin
  Writeln('Количество потребителей ресурса: ');
  {$I-}
  Readln(Consumers);
  if (IOResult <> 0) or (Resourses mod Consumers <> 0) or
    (Consumers < 1) then
  begin
    WriteLn('Ошибка 1: Недопустимые число потребителей');
    Halt;
  end;
  { Продолжение программы }
end.
{ ========================================================================== }
```

Использованный в коде оператор выбора одновременно проверяет синтаксическую корректность введенного числа и ограничения на его значение. Если бы булевское выражение вычислялось полностью, то могла бы возникнуть следующая неприятная ситуация. При вводе синтаксически неправильного числа оператор `ReadLn` состояние переменной `Consumers` не изменит, то есть ее значение будет определяться состоянием отведенного ей в момент запуска программы блока оперативной памяти. Вполне возможно, что это значение будет равно нулю. Следовательно, при попытке вычислить второй операнд условного выражения в операторе выбора весьма вероятна выдача стандартного системного сообщения о делении на ноль, что, конечно, недопустимо.

В языке Pascal по умолчанию действует директива компилятора `{$B-}`, устанавливающая неполное вычисление условных выражений. Таким образом, приведенная программа будет работать корректно. Если по каким либо причинам требуется полное вычисление всех операндов условных выражений, необходимо использовать директиву `{$B+}`.

В качестве окончания данного раздела приведем одну практическую рекомендацию по способу записи оператора выбора. Очевидно, что любой условный оператор можно записать в двух вариантах, отличающихся порядком секций **then** и **else**. К примеру, если нам необходимо проверить, не равна ли нулю переменная a, то это можно сделать так:

```
  if a = 0 then
     <оператор1>
  else
    <оператор2>;
```

А можно так:

```
if a <> 0 then
  <оператор2>
else
  <оператор1>;
```

На первый взгляд кажется, что между этими вариантами нет никакой разницы. Тогда немного изменим ситуацию. Пусть, если a=0, необходимо выполнить много действий, а в противоположном случае одно. Тогда первый вариант примет следующий вид:

```
if a = 0 then
begin
  <оператор1>
  ...
  <операторN>
end
else
  <операторN+1>;
```

А второй вариант будет выглядеть так:

```
if a <> 0 then
  <операторN+1>
else begin
  <оператор1>
  ...
  <операторN>
end
```

Как Вы думаете, какой вариант лучше? Конечно же, второй! В этом случае ключевые слова секций условного оператора располагаются рядом друг с другом, тогда как в первом варианте, чтобы найти секцию **else** потребуется приложить некоторые усилия. Поверьте, их можно потратить с большей пользой, чем разыскивать “убежавшую” секцию.

Выбор из нескольких вариантов

Составим программу, реализующую функциональность простейшего калькулятора с операциями +, -, *, /. Программа должна запрашивать два числа, операцию, которую необходимо выполнить и вычислять результат. При выполнении операции деления необходимо проверить делитель на равенство нулю.

```
{ ========================================================================== }
{ Пример 5.5 }
{ Простейший калькулятор }
Program Calculator;

{$APPTYPE CONSOLE}

var
  x, y: Real;
  Res: Real;
  Op: Char;
begin
  WriteLn('Введите операцию (+, -, *, /): ');
```

```
  ReadLn(Op);
  WriteLn('Введите аргументы: ');
  ReadLn(x, y);
  if (Op = '+') then
    Res := x + y
  else
    if (Op = '-') then
      Res := x - y
    else
      if (Op = '*') then
        Res := x * y
      else
        if (Op = '/') then
        begin
          if (y <> 0) then
            Res := x / y
          else begin
            WriteLn('Деление на ноль');
            Halt;
          end;
        end
        else begin
          WriteLn('Неизвестная операция');
          Halt;
        end;
  WriteLn(x, ' ', Op, ' ', y, ' = ', Res);
end.
{ ======================================================================== }
```

Мы использовали рассмотренный в предыдущем разделе оператор выбора в виде полной альтернативы. Чтобы не дублировать вывод результата, в программе пришлось предусмотреть две дополнительные точки выхода с использованием процедуры `Halt`. Достаточно очевидно, что предложенный вариант обладает рядом недостатков. Если потребуется увеличить количество поддерживаемых калькулятором операций, текст программы будет уходить вправо “за горизонт”. Постоянный повтор фактически одной и той же строки вида

```
  if (Op = 'символ операции') then
```

тоже не доставляет удовольствия. И, наконец, количество строк данной программы, в которых выполняются реальные действия, меньше, чем число строк, содержащих чисто служебную информацию (вроде операторных скобок). Чем вызваны все эти неприятности? Мы столкнулись с явным противоречием: алгоритм в данной программе – это, по сути, выбор из нескольких однотипно описываемых альтернатив, а для его записи у нас в распоряжении лишь оператор выбора из двух вариантов. Если попытаться изобразить блок-схему алгоритма, получится примерно следующее:

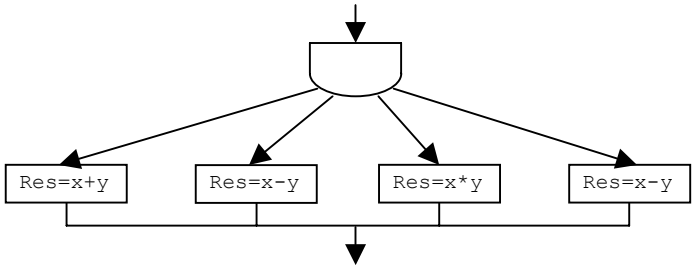


Рис. 5.4. Блок-схема калькулятора

Подобные алгоритмы встречаются весьма часто. В частности по такой же схеме устроен алгоритм обработки команд меню – задача, которую приходится в том или ином виде решать в любой более-менее сложной программе. К счастью язык Pascal содержит прямую реализацию алгоритма выбора из нескольких вариантов – *оператор множественного выбора*.

Формальная запись *оператора множественного выбора* выглядит следующим образом:

```
case <выражение порядкового типа> of
  <список выбора 1> : <оператор1>;
  <список выбора 2> : <оператор2>;
  ...
  <список выбора N> : <операторN>;
  [else <операторN+1>;]
end;
```

Здесь “список выбора” это:

- константа того порядкового типа, который соответствует результату выражения после ключевого слова case;
- список констант через запятую;
- тип диапазон, образованный из констант соответствующего порядкового типа.

При этом любая константа не может содержаться более чем в одном списке выбора.

Например:

```
var
  a, b: Integer;
...
case a of
  1     : b := a + 1;
  3, 4  : b := a - 1;
  5..10 : b := a * a;
  else    b := a;
end;
```

К порядковым типам в языке Pascal относятся целые типы, символьный и булевский типы, перечислимый тип, тип диапазон[1].

Оператор множественного выбора выполняется следующим образом. Вычисляется значение выражения. Если это значение содержится в одном из списков выбора, то выполняется соответствующий оператор. Если значение не принадлежит ни одному из списков выбора, тогда, если имеется секция **else**, то выполняется оператор этой секции, в противном случае оператор **case** не производит никакого действия.

Используя этот оператор, пример 5.5 можно переписать следующим образом:

```
{ ========================================================================== }
{ Пример 5.6 }
{ Простейший калькулятор - вариант 2 }
Program Calculator2;

{$APPTYPE CONSOLE}

var
```

[1] подробнее перечислимый тип и тип диапазон будут рассмотрены в следующей главе

```
  x, y: Real;
  Res: Real;
  Op: Char;
begin
  WriteLn('Введите операцию (+, -, *, /): ');
  ReadLn(Op);
  WriteLn('Введите аргументы: ');
  ReadLn(x, y);
  case Op of
    '+' : Res := x + y;
    '-' : Res := x - y;
    '*' : Res := x - y;
    '/' : if (y <> 0) then
            Res := x / y
          else begin
            WriteLn('Деление на ноль');
            Halt;
          end;
    else  begin
            WriteLn('Неизвестная операция');
            Halt;
          end;
  end;
  WriteLn(x, ' ', Op, ' ', y, ' = ', Res);
end.
{ ========================================================================== }
```

Как видим, текст программы стал заметно короче, понятнее, избавился от лишних элементов. Да и выполнить расширение функциональности калькулятора теперь будет не в пример проще.

Заметим также, что оператор множественного выбора соответствует схеме “один вход – один выход”, а значит, может использоваться в качестве элемента в любой из базисных алгоритмических конструкций.

Программирование цикла

Цикл с предусловием

Рассмотрим задачу вычисления факториала целого неотрицательного числа. Согласно определению *факториал* натурального числа N есть произведение всех чисел от 1 до N, то есть

$$N! = 1 \cdot 2 \cdot \ldots \cdot N$$

$$0! = 1$$

Факториал нуля по определению полагается равным единице.

Составим алгоритм для решения этой задачи. Первое очевидное действие – задание требуемого числа N (с проверкой его корректности в общем случае). Кажется, что вторым шагом должна идти проверка ситуации N = 0, поскольку формула вычисления факториала для нуля не совпадает с формулой для остальных чисел. Вернемся к этому позже. Если N > 0, то для получения

результата нужно несколько раз (а именно N) повторить одно и то же действие – умножение. Очевидно, нам потребуется алгоритмическая конструкция *цикл*. Однако, хоть действие, которое будет выполняться в цикле и одно и то же, то, над чем это действие будет осуществляться, каждый раз будет меняться. На первой итерации цикла нужно умножить 1 на 1, на второй 1 (результат предыдущей итерации) на 2, на третьей 2 (результат предыдущей итерации) на 3, на четвертой 6 на 4, на пятой 24 на 5 и так далее.

Из сказанного можно сделать важный вывод: *алгоритмическая формулировка цикла требует не только выявления повторяемого действия, но и соответствующего однообразного представления обрабатываемых в цикле данных*. Последнее, собственно, и представляет основную "интеллектуальную" проблему программирования цикла.

Итак, в нашей задаче вычисления факториала необходимо предложить однообразное представление сомножителей для каждой итерации. Известно, что формулу расчета факториала можно записать так: N! = (N – 1)! * N. Представим себе, что нам осталось выполнить последнюю (с номером N) итерацию цикла, тогда действие на этой итерации будет состоять в следующем: результат предыдущей итерации умножить на номер текущей итерации. Нетрудно заметить, что формулировка действия никак не связана с тем, что эта итерация была последней, и может быть с успехом применена на всех остальных. Для ее реализации нам потребуется переменная i – счетчик числа итераций и переменная F – результат предыдущей итерации. Таким образом, действия, повторяемые в цикле (так называемое *тело цикла*), будут иметь вид:

```
F = F * i
i = i + 1
```

Последний важный вопрос формулировки цикла – установка начальных значений переменных и условия выхода. В нашем случае логично присвоить начальное значение переменной F = 1 (минимальное значение факториала). Чтобы задать начальное значение счетчика i, необходимо определиться с условием останова цикла. Особенность рассматриваемой задачи состоит в том, что, если N = 0, то умножение производить не надо, так как правильное значение факториала обеспечено начальной установкой переменной F. Таким образом, если мы установим начальное значение счетчика i = 1, то условие выхода $i \le N$ обеспечит невыполнение цикла при N = 0 и правильный подсчет числа умножений при $N \ge 1$.

Итак, все необходимые для построения алгоритма решения приняты. В результате мы можем представить следующий псевдокод:

```
вводим N
i = 1
F = 1
пока i ≤ N выполнять
начало
  F = F * i
  i = i + 1
конец
вывод значения F
```

Использованная нами конструкция в точности соответствует последней из базисных алгоритмических структур – *структуре цикла*. Реализация этой структуры в языке Pascal выполнена следующим образом:

```
while <условное выражение> do
  <оператор>;
```

При этом, как и в *операторе выбора* условное выражение должно иметь *булевский* тип. Пока результат вычисления выражения – **true** (истина), выполняется оператор. При необходимости выполнять в теле цикла несколько действий, нужно использовать *составной оператор*. Обращаем внимание на то, что внутри цикла должны содержаться операторы, изменяющие значение условного выражения так, чтобы оно, в конце концов, стало равно **false**.

Приведем теперь программу вычисления факториала.

```
{ ========================================================================== }
{ Пример 5.7 }
{ Вычисление факториала }
Program Factorial;

{$APPTYPE CONSOLE}

var
  N, i: Word;
  F: Longword;
begin
  {$R+}
  Write('Аргумент: ');
  Readln(N);
  i := 1;
  F := 1;
  while i <= N do
  begin
    F := F * i;
    i := i + 1;
  end;
  WriteLn(N, '! = ', F);
  ReadLn;
end.
{ ========================================================================== }
```

Чтобы не загромождать пример явными проверками корректности, при объявлении переменной N мы использовала тип Word (неотрицательные числа) и воспользовались директивой {$R+}, которая обеспечит вставку в исполняемый модуль команд проверки на принадлежность введенного значения N диапазону типа Word. Для переменной F тип Word является не слишком подходящим, поскольку факториал – очень быстро растущая функция (диапазон типа Word от 0 до 65 535). Впрочем, и использованный нами тип Longword с диапазоном от 0 до 4 294 967 295 также позволяет корректно вычислять факториал лишь для небольших значений N. Попробуйте, например, последовательно вычислить с использованием данной программы факториал для чисел 11, 12 и 13.

Вообще говоря, представленный код является не совсем корректным, поскольку не предусматривает проверки на максимально возможное значение N, при котором программа еще способна выдать правильный результат. Однако выполнить такую проверку не так-то просто. Как именно это можно сделать мы обсудим чуть позднее.

Цикл с постусловием

Вернемся еще раз к задаче о контроле получаемой от пользователя информации. Как мы уже отмечали выше, ввод данных является потенциально опасным местом с точки зрения выполнения программы – пользователь может случайно (или намеренно, если пользователь – тестер программы) задать неверные данные, на которых программа не сможет продолжать работу. Однако, с точки зрения пользователя, тот факт, что он ошибся при вводе, не может служить основанием для аварийного завершения программы. Налицо явное противоречие. Самый правильный подход в данной ситуации – не дать пользователю ошибиться вообще. Например, при запросе дня месяца программа может показывать на экране календарь и предлагать пользователю *выбрать* нужный день. Понятно, что в этом случае выбрать 32-е число пользователю не удастся при всем желании. К сожалению, такой подход не всегда возможен. Если список альтернатив заранее неизвестен, и выбор предоставить нельзя, тогда то, что ввел пользователь, необходимо проверять. Проверять на соответствие типов (требовалось ввести число, а ввели букву) и на допустимость (попадание в диапазон или в некоторый допустимый набор вариантов). Программирование таких проверок мы рассмотрели в предыдущем разделе. Там же было отмечено, что кроме проверки корректности необходимо предоставить пользователю возможность повторить ввод в случае ошибки. Теперь мы можем реализовать такой повтор. Перепишем код примера 5.3 так, чтобы обеспечить правильный ввод данных.

```
{ ================================================================== }
{ Пример 5.8 }
{ Контроль вводимой информации – вариант 3 }
Program Control3;

{$APPTYPE CONSOLE}

var
  Day: Integer;
  IORes: Integer;
begin
  {$I-} { отключили автоматическую проверку ввода-вывода }
  Write('Введите день месяца: ');
  ReadLn(Day);
  IORes := IOResult;
  while (IORes <> 0) or (Day < 1) or (Day > 31) do
  begin
    if (IORes <> 0) then
      WriteLn('Ошибка 1: Недопустимые символы в целом числе')
    else
      WriteLn('Ошибка 2: День месяца вне диапазона 1..31');
    Write('Введите день месяца: ');
    ReadLn(Day);
    IORes := IOResult;
  end;
  { Дальнейший текст программы }
end.
{ ================================================================== }
```

Проанализируем представленный вариант программы.

Во-первых, исчезли оба досрочных выхода с помощью процедуры `Halt`. Теперь программа добивается от пользователя получения верных данных, вместо завершения работы в случае ошибки.

Во-вторых, после первоначального ввода значения переменной `Day`, запускается цикл **`while`**, проверяющий возможные ошибки. Внутри цикла, если ошибка имела место, она классифицируется, выдается соответствующее сообщение, после чего ввод повторяется.

В третьих, в коде появилась новая переменная `IORes`, в которую сохраняется значение, возвращаемое функцией `IOResult`. Вызвано это тем, что функция `IOResult` не только возвращает состояние последней операции ввода-вывода, но и обнуляет внутренний флаг ошибки. Таким образом, повторный ее вызов, необходимый в условном операторе, сообщающем пользователю вид ошибки, не привел бы к нужному эффекту.

И, наконец, представленный вариант обзавелся повтором из трех одинаковых строк:

```
Write('Введите день месяца: ');
ReadLn(Day);
IORes := IOResult;
```

Эти строки выполняются до цикла – первоначальный ввод, и внутри цикла – повторный ввод. Неприятность эта – прямое следствие конструкции цикла **`while`**. Цикл **`while`** является *циклом с предусловием* – сначала проверка, потом действие. А условное выражение, на основании значения которого принимается решение об остановке, естественно не может содержать неизвестных (неопределенных) переменных.

Вместе с тем, в данном случае намного логичнее было бы использовать цикл, организованный по противоположному принципу – сначала действие, затем проверка и при необходимости повтор, то есть *цикл с постусловием*.

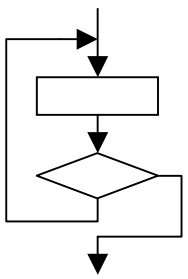


Рис. 5.5. Блок-схема цикла с постусловием

Ситуации, в которых требуется подобная схема организации цикла, встречаются не так уж редко, а строк кода, которые необходимо будет продублировать, если использовать для ее программирования цикл **`while`**, может оказаться много больше, чем три. Таким образом, наличие в языке программирования реализации *цикла с постусловием* является весьма полезным. В языке Pascal такая реализация есть и выглядит она следующим образом:

```
repeat
  <оператор 1>;
  <оператор 2>;
  ...
  <оператор N>;
until <условное выражение>;
```

Дословный перевод с английского этой конструкции: “повторять … пока не …”. Таким образом, в отличие от *цикла с предусловием* **`while`**, цикл **`repeat`** выполняется, пока результат

вычисления условного выражения – **false** (ложь). Кроме того, в теле цикла **repeat** может располагаться несколько операторов – в данном случае в качестве *операторных скобок* выступают ключевые слова **repeat** и **until**.

С использованием цикла с постусловием пример 5.8 может быть переписан следующим образом:

```
{ ========================================================================== }
{ Пример 5.9 }
{ Контроль вводимой информации - вариант 4 }
Program Control4;

{$APPTYPE CONSOLE}

var
  Day: Integer;
  IORes: Integer;
begin
  {$I-} { отключили автоматическую проверку ввода-вывода }
  repeat
    Write('Введите день месяца: ');
    ReadLn(Day);
    IORes := IOResult;
    if (IORes <> 0) then
      WriteLn('Ошибка 1: Недопустимые символы в целом числе')
    else
      if (Day < 1) or (Day > 31) then
        WriteLn('Ошибка 2: День месяца вне диапазона 1..31');
  until (IORes = 0) and (Day >= 1) and (Day <= 31);
  { Дальнейший текст программы }
end.
{ ========================================================================== }
```

Итак, после некоторых усилий мы получили, наконец, код, который в точности соответствует требуемой функциональности в задаче о контроле ввода информации: ввели значение; проверили на корректность; если были ошибки, сообщили о них и вернулись к вводу; если не было, вышли из цикла и продолжили работу.

Цикл с известным числом повторений

Описанные в двух предыдущих разделах операторы цикла **while** и **repeat** являются универсальными и позволяют запрограммировать любой алгоритм, связанный с выполнением повторяющихся действий. Однако среди всех таких алгоритмов существует очень большой класс, в котором циклическая обработка выполняется заранее известное количество раз и чаще всего над элементами некоторого упорядоченного множества.

Все алгоритмы этого класса обладают общим устройством. В них требуется счетчик числа итераций и задание для этого счетчика начального и конечного значений. При этом конечное значение определяет условие останова цикла, а сам счетчик на каждой итерации меняется на одно и то же число, чаще всего равное единице. Одним из примеров такого алгоритма является рассмотренный выше алгоритм вычисления факториала.

Вследствие перечисленных обстоятельств язык Pascal (как и многие другие языки программирования) содержит специальный оператор цикла для программирования алгоритмов подобного вида – оператор *цикла с известным числом* повторений **for**.

Формальная запись оператора **for** выглядит следующим образом:

```
for <счетчик цикла> := <начальное значение> to <конечное значение> do
  <оператор>;
```

При этом `счетчик цикла` – переменная *порядкового* типа, `начальное значение` и `конечное значение` – арифметические выражения, результат каждого из них должен быть того же типа, что и счетчик или должен быть совместим с ним по присваиванию. В конце каждой итерации счетчик цикла автоматически увеличивается на единицу. Изменение счетчика в теле цикла не допускается[1]. Количество итераций, которое будет выполнено, вычисляется до входа в цикл, таким образом, изменения в теле цикла переменных, входящих в `начальное значение` и `конечное значение`, не приводят к изменению числа итераций. Более того, число итераций, которое выполняет цикл **for** всегда[2] равно `конечное значение – начальное значение + 1`.

Кроме указанной формы цикл **for** имеет второй вариант:

```
for <счетчик цикла> := <начальное значение> downto <конечное значение> do
  <оператор>;
```

В этом варианте в конце каждой итерации счетчик цикла автоматически *уменьшается* на единицу.

Если к моменту входа в цикл `начальное значение` оказалось больше (во второй форме цикла меньше), чем `конечное значение`, то цикл не выполняется ни разу.

Попытаемся теперь переписать пример 5.7 с использованием цикла **for**.

```
{ ========================================================================== }
{ Пример 5.10 }
{ Вычисление факториала – вариант 2 }
Program Factorial2;

{$APPTYPE CONSOLE}

var
  N, i: Word;
  F: Longword;
begin
  {$R+}
  Write('Аргумент: ');
  Readln(N);
  F := 1;
  for i := 1 to N do
    F := F * i;
  WriteLn(N, '! = ', F);
  ReadLn;
end.
{ ========================================================================== }
```

[1] сказанное верно для Object Pascal, более ранние версии языка позволяли менять значение счетчика цикла в теле цикла.

[2] поскольку в более ранних версиях языка Pascal смена значения счетчика цикла внутри тела цикла разрешалась, то число итераций цикла for можно было изменить. В Object Pascal это стало невозможным.

Как видим, код стал существенно короче – вся работа со счетчиком итераций, которую в примере 5.7 мы программировали явным образом, теперь выполняется автоматически.

Приведем еще один пример не вполне очевидного использования цикла **for**. Пусть мы хотим распечатать коды символов букв английского алфавита от ‘A’ до ‘Z’. Фрагмент кода для этой задачи может выглядеть следующим образом:

```
var
  ch: Char;
...
for ch := 'A' to 'Z' do
  WriteLn('For simbol ', ch, ' code is ', Ord(ch));
```

Здесь используется тот факт, что тип `Char` относится к порядковым, а значит вполне может служить типом счетчика цикла **for**.

Выход из тела цикла

Еще раз вернемся к задаче о вычислении факториала. Представленные в примерах 5.7 и 5.10 программы допускают лишь однократное выполнение. Однако на практике пользователю может потребоваться вычислить факториал не для одного, а для нескольких чисел. Конечно, можно в самом начале программы спросить, сколько факториалов нужно будет рассчитать и весь имеющийся расчетный код заключить в цикл с известным числом повторений **for**. Однако это решение не вполне корректно. Общепринятым способом использования программ является вариант, в котором программа работает до тех пор, пока пользователь не решит из нее выйти. Таким образом, обычно тело программы (кроме начальных и финальных действий, которые выполняются однократно) заключается в тело цикла **repeat**, в конце которого задается вопрос вида: “Продолжить (Да/Нет) ?”. В данном случае мы можем сэкономить и запрос числа N совместить с вопросом об окончании работы. Поскольку N = –1 является недопустимым значением, вопрос на ввод информации может выглядеть так:

```
Write('Аргумент (-1 - для завершения): ');
```

Такое решение приводит к тому, что в теле цикла **repeat** нам потребуется проверка. По соображениям, указанным в разделе “Программирование выбора”, оптимальный вариант такой проверки:

```
if N = -1 then
  ...;
```

Осталось только решить, что должно стоять на месте трех точек? Если после цикла **repeat** никаких действий больше делать не надо, то можно использовать такой вариант:

```
if N = -1 then
  Halt;
```

А если надо? Как быть в этом случае? Хотелось бы вместо `Halt` подставить что-то, что позволило бы выйти из цикла досрочно. То есть реализовать следующую схему

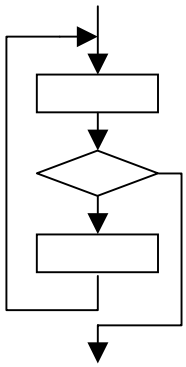


Рис. 5.6. Блок-схема цикла с выходом из тела цикла

К сожалению, в языке Pascal такой оператор отсутствует. Тем не менее, указанную схему можно запрограммировать.

Во-первых, для этого можно использовать *оператор безусловного перехода* **goto**.

Формат этого оператора:

```
goto <метка>;
```

где метка – целое число в диапазоне от 0 до 9999 или *идентификатор*. Метки должны быть объявлены в собственной секции объявлений **label**, где они просто перечисляются через запятую.

```
label
  1, 345, EndLoop;
```

Метка может быть поставлена в начале любой строки тела программы, содержащей оператор (в том числе *пустой оператор*). От остальной строки метка отделяется двоеточием.

Оператор **goto** передает управление на строку, отмеченную меткой.

Используя этот оператор, мы можем представить следующий код:

```
{ ========================================================================== }
{ Пример 5.11 }
{ Вычисление факториала – вариант 3 }
Program Factorial3;

{$APPTYPE CONSOLE}

label 1;
var
  i: Word;
  N: Integer;
  F: Longword;
begin
  {$R+}
  repeat
    Write('Аргумент (-1 - для завершения): ');
    Readln(N);
    if N = -1 then
      goto 1;
```

```
    F := 1;
    for i := 1 to N do
      F := F * i;
    WriteLn(N, '! = ', F);
  until false;
l: ;
end.
{ ========================================================================== }
```

Оператор безусловного перехода существует в языках программирования очень давно. Без него невозможно программирование на языке низкого уровня Assembler, также весьма активно он используется в большинстве версий языка Basic. Однако повсеместное применение этого оператора в программах делает их код чрезвычайно запутанным, сложным для прочтения и поиска ошибок и, хотя во многих языках высокого уровня этот оператор существует, использовать его не рекомендуется. Помимо прочего, оператор безусловного перехода противоречит положениям технологии структурного программирования, поскольку из любой алгоритмической конструкции позволяет выйти, что называется “через окно”. Таким образом, прежде чем использовать оператор **goto**, убедитесь, что другого способа реализации нужного Вам алгоритма нет.

В силу указанных обстоятельств, а также поскольку конструкция цикла, в которой решение о продолжении или выходе требуется принимать в его середине, весьма распространена, в реализацию языка Pascal от фирмы Borland, начиная с версии 7.0, ввели, а в языке Object Pascal сохранили специальные *операторы управления циклом*.

Первый из них – оператор **break** – представляет второй способ реализовать конструкцию, изображенную в виде блок-схемы на рисунке 5.6. Его вызов обрывает выполнение цикла и передает управление на оператор, стоящий *сразу за* циклом. В этом принципиальное отличие от оператора безусловного перехода **goto**, с помощью которого управление может быть передано в любую точку программы, в том числе стоящую выше по тексту.

Второй оператор управления циклом – **continue** – предназначен для досрочного окончания не всего цикла, а только текущей итерации. При его вызове выполнение итерации прекращается, управление передается на начало цикла и начинается следующая итерация. В цикле **for** при этом, как и положено, происходит изменение счетчика цикла.

Использование оператора **break** вместо **goto** в примере 5.11, позволит избавиться от объявления метки и пустого оператора в конце программы, то есть сократит текст на три строки.

Расширенный пример

В конце главы рассмотрим полное решение задачи, с которой мы начали раздел “Программирование цикла”.

Все представленные в предыдущих примерах программы вычисления факториала обладают одним существенным недостатком, который никак не связан с расчетным алгоритмом, а вытекает лишь из выбранного способа хранения результата (тип переменной F) и того факта, что любой числовой тип данных представляет не все числовое множество (целое или вещественное), а лишь ограниченную его часть.

В данном случае проблема состоит в том, что какой бы тип мы не выбрали для переменной F, рано или поздно мы “упремся” в верхнюю границу значений этого типа, после чего программа перестанет выдавать верные результаты. Решение, которое первым приходит в голову, – в самом начале программы подсчитать максимально допустимое N, соответствующее выбранному типу переменной F, сообщить об этом пользователю и проверять ввод переменной N. Решение правильное со всех точек зрения, осталось лишь понять, как воплотить его в жизнь.

Возникшую задачу можно представить математически. Имеется монотонно возрастающая функция F(N). Дано число FMax. Требуется найти максимальное N, при котором $F(N) \le FMax$. Получилась задача на решение неравенства от одной переменной. Однако в силу сложности формулы F(N) найти решение аналитически будет весьма непросто.

Возможный численный вариант состоит в следующем. Будем последовательно увеличивать N, начиная с 1, подсчитывать значение факториала и так до тех пор, пока F(N) не превысит первый раз значения FMax. Все правильно? Математически абсолютно. Вот только программу по такому принципу составить нельзя. Проблема как раз в том и состоит, что за FMax в типе данных ничего больше нет. Кажется, мы зашли в тупик. Или нет? Напрягаем интеллект и… Эврика! Если нельзя “перейти” через границу FMax, надо остановиться в тот момент, когда следующий шаг сделать будет невозможно. В такой постановке проблема решается: будем на каждой итерации делить FMax на текущее значение F нацело и сравнивать частное со следующим N. Как только частное станет меньше чем N, пора остановиться.

Наконец, для того чтобы программа была максимально корректной, добавим в нее контроль за вводом числа N. В результате получится следующий код.

```
{ ========================================================================== }
{ Пример 5.12 }
{ Вычисление факториала - вариант 4 }
Program Factorial4;

{$APPTYPE CONSOLE}

var
  N: Integer;
  NMax, i: Word;
  F, FMax: Longword;
  IORes: Integer;
begin
  {$I-}
  FMax := High(Longword);
  F := 1;
  i := 1;
  while (FMax div F >= i) do
  begin
    F := F * i;
    Inc(i);
  end;
  NMax := i - 1;
  repeat
    Write('Аргумент (-1 - для завершения): ');
    Readln(N);
    if N = -1 then
      break;
    IORes := IOResult;
    if (IORes <> 0) then
```

```
    begin
      WriteLn('Ошибка 1: Недопустимые символы в целом числе')
      continue;
    end
    else
      if (N < -1) or (N > NMax) then
      begin
        WriteLn('Ошибка 2: Аргумент вне диапазона 0..', NMax);
        continue;
      end;
    F := 1;
    for i := 1 to N do
      F := F * i;
    WriteLn(N, '! = ', F);
  until false;
  ReadLn;
end.
{ ======================================================================== }
```

Выводы

В данной главе мы перекинули мостик от написания “игрушечных” программ к проектированию и созданию полноценных программных продуктов, в общих чертах познакомившись с понятием технологии программирования. Нами был получен ответ на важный вопрос о том, что такое разработка программ. С каждым днем становится все больше сторонников того мнения, что разработка программ есть технологический процесс со своими методами, приемами, средствами – технологиями, обеспечивающими повышение производительности труда программистов и способствующими достижению результата. Хорошо известен один из так называемых законов Мерфи: “Любая программа обходится дороже и требует больших затрат времени, чем предполагалось”. Задача нас как специалистов в данной области побороть эту закономерность в максимально возможной степени, и структурное программирование, с основным положениями которого мы познакомились в данной главе – первый шаг в этом направлении.

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.

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 число 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].

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.

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

[1] подробно о подпрограммах мы поговорим в главе 7

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, также как и в стандартном программном обеспечении, нельзя заранее предусмотреть всевозможные 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 что нами задано лишь множество значений. Вторая существенная 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 над 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.

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.

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.

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.

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 раннее, 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 цеху. 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.

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 также 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.

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 также 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.

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 окончание 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 программу, 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.

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 элементов 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 хоть 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.

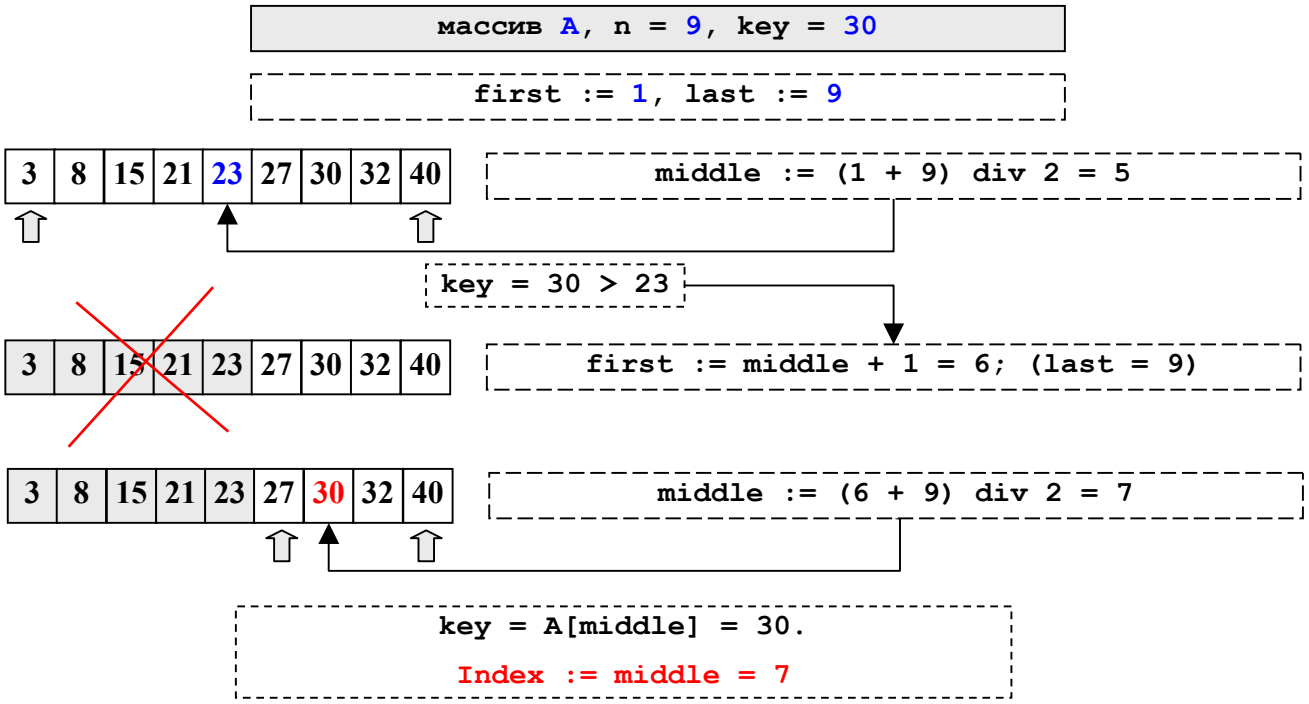


Рис. 6.1. Бинарный поиск в упорядоченном массиве

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, расположенные 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”.

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 года 1700, 1800, 1900. И соответственно не будет год 2100. Таким образом, приведенная программа будет корректно работать лишь до 2099 года.

Поля записи, также 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;
```

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 правильность 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: (Jornal: 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;
  Jornal := '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.

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, характеризующие 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;
```

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.

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.

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 присвоение `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;

[1] более подробно речь об этом пойдет в главе 7

- `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.

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.

7. Модульное программирование

Внутри каждой большой задачи сидит маленькая, пытающаяся пробиться наружу.
Закон больших задач Хоара.

Представьте себе такую ситуацию: Вы руководитель отдела в программистской фирме. К Вам пришел заказчик со следующим предложением: “Мне нужна программа для нахождения равновесной цены на рынке”.

Вы: “Отлично. Вы пришли в нужное место. У нас лучшие специалисты по нахождению цен на рынке, и именно равновесных”.

Что? Что-то не так? У Вас возникло чувство дежавю? Или, может, авторы что-то перепутали? Уверяем Вас, что нет. Просто на этой ситуации, с которой мы, действительно, начали самую первую главу книги, мы хотим обсудить еще одну из весьма важных концепций, входящих в обязательный багаж знаний любого квалифицированного программиста.

Итак, разовьем ситуацию дальше. Представьте, что с учетом приобретенного опыта Вы провели анализ предметной области, построили информационную модель задачи, разработали алгоритм решения. Пора приступать к программированию. И тут… Вы вспоминаете, что Вы *руководитель отдела*, то есть у Вас в подчинении *несколько* человек. И каждый горит желанием поработать. Ваша прямая обязанность разделить задачу между всеми, так чтобы, в конечном счете, и подчиненные не “били баклуши”, теряя квалификацию, и работа была сделана по возможности быстро и качественно. Как это сделать?

Конечно, Вы можете возразить, что данная задача достаточно проста, а значит ее вполне можно “отдать на откуп” одному человеку. Все верно, однако в реальной ситуации задача будет посложнее описанной, и вопрос, нами заданный, неизбежно возникнет. Итак…

В принципе, достаточно несложно прийти к мысли, что в указанной нами ситуации имеющуюся задачу надо попытаться разбить на подзадачи, выполнение каждой из которых поручить отдельному специалисту. Этот принцип давно и с успехом работает в других областях человеческой деятельности. И все бы хорошо. Вот только то, что мы с Вами успели изучить к настоящему моменту, не дает ответа на вопрос: А можно ли точно так же разбить и программу на части? Представьте себе, как Ваши программисты договариваются: Коллеги, участок программы с 1250-ой по 1740-ю строку пишу я, не трогайте его, будьте так добры.

Подводя итог, хотим отметить, коллективная деятельность является основополагающей формой выполнения работ в человеческом обществе. К тому же возможность разбиения работы на составляющие облегчает достижение результата, даже если коллектив состоит всего из одного исполнителя. И разработка программного обеспечения не могла стать исключением. А значит, сообщество программистов неизбежно должно было выработать средства поддержки такой формы деятельности. Обсуждению этих средств с общих позиций и конкретного их воплощения в языке Object Pascal и посвящена настоящая глава.

Концепция модульного программирования

Технология модульного программирования – оформившаяся в начале 70-х годов XX века идея разработки больших программных систем [45]. Это фундаментальная концепция, являющаяся основой всех современных подходов к проектированию и реализации. В то же время суть ее проста и отражает широко известные научные и технические методы, заключающиеся в поиске и реализации некоторого базового набора элементов, комбинации которых дают решение всех задач из определенного круга.

Если концепция структурного программирования, рассмотренная нами в главе 5, предлагает некоторый универсальный *алгоритмический* базис, то модульное программирование состоит в разработке под конкретную задачу или круг задач (предметную область) *собственного* базиса в виде набора *модулей*, позволяющего наиболее эффективно по целому ряду критериев построить программный комплекс. Модули, входящие в базис, это целые программы (в отличие от примитивов структурного программирования), решающие некоторые подзадачи основных задач.

Средства поддержки технологии модульного программирования в целом следует разделить на два уровня. Первый, которому мы уделим основное внимание, это *аппарат подпрограмм*, имеющийся в языках программирования и, в том числе, причем в весьма развитом виде, в языке Pascal. Эти языковые средства направлены на оформление алгоритма в виде отдельного модуля и определение его *интерфейса* с другими частями программного комплекса. Второй уровень – это надъязыковые средства в виде различных инструментальных систем разработки пакетов прикладных программ. Целью этих систем является создание универсальных средств сборки программы из модулей по заданию, составленному пользователем-непрограммистом на так называемом профессиональном языке, то есть языке, отражающем понятия предметной области.

Необходимость модульного разбиения программной системы

Применение технологии модульного программирования начинается уже с первых этапов разработки – четкой формулировки задачи и построения математических моделей. Целью так называемого *модульного анализа* предметной области является выделение подзадач, алгоритмы решения которых будут оформлены в виде модулей. Рассмотрим мотивы, которыми руководствуются при выделении той или иной подзадачи-модуля.

Прежде всего, пытаются избежать дублирования кода, основанного на применении похожих или даже совпадающих методов или их фрагментов, необходимых для решения разных задач предметной области. Такие универсальные или, как еще говорят, инвариантные методы оформляются в виде самостоятельных модулей и составляют модульный базис.

Однако выделение некоторого метода в отдельный модуль часто производится даже, если он используется при решении всего лишь одной задачи. Необходимость такого разбиения может быть вызвана большим объемом программы. Как и в других областях человеческой деятельности, со сложной программистской задачей значительно легче справиться по частям (при некотором уровне сложности другого способа просто не существует), программируя и отлаживая части отдельно. Нетрудно понять, что держать в голове логику работы программы размером в сотню другую тысяч строк, заниматься ее совершенствованием и отладкой, просто невозможно.

Развивая эту мысль, укажем, что еще одним побуждающим мотивом использования модульного программирования является разбиение задачи на части с целью ее коллективного решения. В этом случае каждому исполнителю необходимо выделить собственный участок работы, что наиболее естественно сделать, если разрабатываемые ими фрагменты общей программной системы будут оформлены в виде отдельных модулей.

Следующим важным моментом является “объем” больших программных комплексов. В настоящее время многие программные системы настолько велики по размерам, что их размещение в оперативной памяти как единого целого либо чрезвычайно неэффективно, либо в принципе невозможно. Следовательно, такую программу приходится разбивать на части – модули, которые загружаются в память по мере возникновения в них необходимости.

Еще один фактор связан с уже упоминавшейся нами необходимостью модификации, которой подвергается любая промышленная программа в течение своей “жизни”. Правильно выполненный модульный анализ предусматривает выделение частей программы, которые в перспективе могут быть изменены, с тем, чтобы это не привело к необходимости переделки всей системы в целом. Характерным примером такой модифицируемой части является пользовательский интерфейс. Допустим, при первоначальной разработке в качестве интерфейса программы был выбран консольный вариант. В дальнейшем по мере увеличения функциональности возникла потребность перейти на графический интерфейс пользователя. Очевидно, разработчик существенно облегчит себе жизнь, если при проектировании уже первого варианта системы предусмотрит возможность такой ситуации и выделит блок “визуализации” в отдельный модуль, изменение которого не будет затрагивать остальные части программы.

Наконец, модульная структура программы может облегчить отладку. Правда, в данном случае применение модульного подхода – палка о двух концах. С одной стороны, ясно, что отдельный модуль легче отлаживать, чем программу в целом. Кроме того, модульное разбиение помогает быстрее локализовать ошибку при тестировании программы. Однако, с другой стороны, модульность порождает и новые проблемы при отладке. Во-первых, возникает вопрос, как отлаживать отдельный модуль, если он во время работы взаимодействует с другими частями программы, которые еще не отлажены или просто не реализованы? Во-вторых, программа, собранная из правильно работающих частей, может в целом работать неправильно, поскольку, например, имеются рассогласования во взаимосвязи модулей при обмене данными. Разумеется, существуют подходы к решению этих проблем. Первая из них решается с помощью “имитаторов” или “заглушек”, то есть созданием фиктивного окружения отлаживаемого модуля, реализующего передачу ему стандартной отладочной информации. Вторая проблема может быть решена лишь путем тщательного проектирования и проверки интерфейса модулей. В целом, надо понимать, что отладка многомодульной программы – это специфичная проблема, требующая применения особой техники.

Средства поддержки модульной технологии в языках программирования

Надеемся, вышесказанное убедило Читателя в безусловной необходимости технологии модульного программирования. На самом деле можно сказать, что в настоящее время все программы, исключая быть может простейшие чисто школьные задачи, разрабатываются с использованием данной технологии. Рассмотрим теперь в общем виде, какие возможности обычно предоставляют языки программирования для поддержки модульной технологии.

Подпрограммы

Практически все распространенные языки программирования предлагают средства оформления модулей в виде *подпрограмм*, разделяя их на два вида: *процедуры* и *функции*. Между собой они отличаются синтаксическим оформлением, а также способом вызова и передачи результирующего значения. Отметим, что содержательно вызов подпрограммы любого вида состоит в одном и том же: работа вызвавшего модуля приостанавливается, управление передается на первый оператор вызванного модуля, модуль выполняет действия до момента своего завершения, управление возвращается вызвавшему модулю.

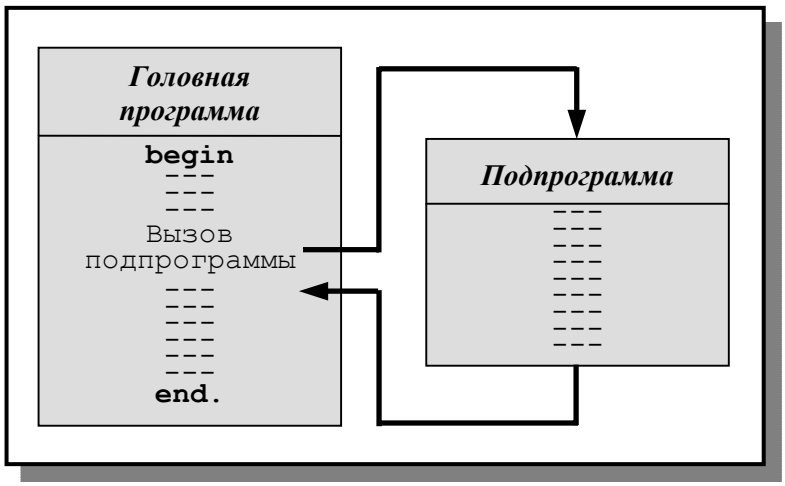


Рис. 7.1. Вызов подпрограммы

Подпрограммы-функции используются в случаях, когда необходимо вернуть некоторый, чаще всего числовой результат (если не учитывать так называемого побочного эффекта, о котором мы поговорим ниже). Обычно функции участвуют в выражениях в качестве операндов.

Подпрограммы-процедуры вызываются с помощью отдельного оператора и возвращают результат произвольного типа в заранее оговоренные переменные (или вообще не выдают результирующих значений, производя лишь какие-нибудь действия).

Использование подпрограмм предполагает решение вопроса о способе обмена информацией между ними. Существует два подхода к организации такого обмена: с помощью списка параметров и с помощью общих разделяемых разными модулями областей памяти.

Список параметров – набор переменных, задаваемых при описании подпрограммы и используемых в ее теле в качестве средства получения исходных данных (входные параметры) и передачи результатов работы (выходные параметры). При написании подпрограммы параметрам задаются некоторые содержательные имена. Поскольку эти имена лишь формально обозначают переменные для обмена информацией, то и называются они *формальными параметрами*. При вызове подпрограммы в списке параметров указываются имена переменных, в которых должны находиться входные для модуля данные или в которые надо поместить выходные результаты. Эти переменные называются *фактическими параметрами*.

Передача параметров в подпрограмму и из нее обычно осуществляется двумя принципиально отличными способами: *по значению* и *по ссылке* (адресу). В первом случае в подпрограмме для параметра отводится собственная память, куда копируется значение фактического параметра из вызвавшего модуля. Во втором подпрограмма получает адрес фактического параметра вызвавшего модуля и работает с ним напрямую.

Альтернативным способом обмена информацией между модулями является выделение некоторых общедоступных глобальных переменных. В общем случае этот вариант ускоряет работу подпрограмм, но может привести к существенным трудностям в логике программы в целом. Кроме того, общедоступность глобальных переменных ведет к возможности их несанкционированного изменения, а отловить подобные ошибки – дело весьма непростое.

Сборка

Программа, состоящая из модулей, должна быть собрана в единое целое. Технически сборка может осуществляться несколькими способами.

Первый вариант – сборка на уровне исходных текстов. Она выполняется программистом самостоятельно с помощью редактора текстов. При этом используется соответствующая технология оформления модулей в виде *внутренних подпрограмм*. В результате компиляции собранного текста получается единый объектный модуль.

Второй вариант предполагает отдельную трансляцию каждого модуля. Исполняемый модуль из полученных объектных модулей строится специальной программой – редактором связей. Организация такого способа сборки требует наличия в языке программирования аппарата *внешних подпрограмм*.

Наконец, возможен вариант, когда окончательное объединение модулей в оперативной памяти вообще не происходит, а они загружаются с диска по мере необходимости. Как мы уже упоминали, этот вариант характерен для больших программных систем.

Подпрограммы в языке Object Pascal

Описание и вызов процедур и функций

Правила обращения к функциям и процедурам нам уже фактически известны. Совсем обойтись без подпрограмм в языке Pascal практически невозможно, и мы уже использовали довольно большое их количество, просто не заостряя на этом внимания.

Итак, формально *функция* вызывается в выражениях указанием ее имени со следующим за ним в скобках списком параметров. Вызов *процедуры* оформляется записью в виде отдельного оператора, состоящего из имени процедуры со списком параметров. Заметим, что и у процедур, и у функций список параметров может отсутствовать.

Синтаксические правила оформления процедур и функций, а также правила обращения к ним одинаковы для внутренних и внешних подпрограмм языка Pascal.

Текст подпрограммы состоит из заголовка и следующего за ним блока. Напомним, что блок – это совокупность объявлений и исполняемых операторов, причем весь набор операторов заключается в скобки **begin** и **end**.

Заголовки функции и процедуры имеют вид:

```
function <имя функции>(<список формальных параметров>): <тип возвращаемого
  результата>;
procedure <имя процедуры>(<список формальных параметров>);
```

Здесь **function** и **procedure** – зарезервированные слова. Имена функций и процедур – идентификаторы.

Рассмотрим примеры описания функции и процедуры:

```
{ ========================================================================== }
{ Пример 7.1 }
{ Подпрограммы }

{ Показательная функция }
function Deg(a, x: Real): Real;
begin
  Deg := Exp(x * Ln(a)); { Оператор присваивания, указывающий }
                         { результирующее значение            }
  //  Result := Exp(x * Ln(a));
  { Это второй способ вернуть результат }
end; {Deg}

{ Частное и остаток от деления целых чисел }
procedure DivMod(x, y: Integer; var d, m: Integer);
begin
  d := x div y;
  m := x mod y;
end; {DivMod}
{ ========================================================================== }
```

Список параметров представляет собой перечисление формальных параметров через точку с запятой. Каждый параметр задается идентификатором со следующим за ним через двоеточие описателем типа. Если несколько параметров принадлежат одному типу, все их можно перечислить перед описателем типа через точку с запятой.

Способ передачи параметра (по значению или по ссылке) задается с помощью ключевого слова **var**. Если **var** указано перед параметром, то параметр передается по ссылке, в противном случае – по значению. Параметры, передаваемые по ссылке, в терминологии языка Pascal называются *параметрами-переменными*. Параметры, передаваемые по значению, называются *параметрами-значениями*.

Параметры-значения в языке Pascal передаются только в одну сторону – из вызывающей подпрограммы в вызываемую. Другими словами, параметр-значение может быть только входным и не может использоваться для возвращения вычисленных подпрограммой значений. Таким образом, входные параметры, оформленные как параметры-значения, защищены от непредусмотренного изменения в процессе работы подпрограммы, что, конечно, является плюсом. Достигается это за счет того, что реально подпрограмма оперирует с копией переданного параметра. Создание копии (выделение памяти и копирование значения) осуществляется автоматически. Попутно это позволяет внутри подпрограммы использовать параметры-значения как обычные переменные, при этом изменение их значений никак не скажется на фактических параметрах.

Однако в таком подходе есть и свои минусы. Если в подпрограмму требуется передать структуру, требующую для размещения сотню килобайт, создание копии может стать весьма накладным, как по памяти, так и по времени, которое будет потрачено на собственно копирование значений. Корректное решение этой проблемы мы обсудим чуть ниже.

При передаче в подпрограмму параметра-переменной копии не создается, вместо этого передается адрес фактического параметра, что, естественно, экономит время, память и дает возможность изменять значение параметра внутри подпрограммы.

Соответствие между формальными и фактическими параметрами при вызове устанавливается по порядку их следования в заголовке подпрограммы и в операторе вызова (или в указателе функции). Разумеется, количество фактических и формальных параметров должно совпадать.

Важное требование, о котором следует постоянно помнить, состоит в обязательной идентичности типов соответствующего фактического и формального параметра. Таким образом, анонимное объявление типа формального параметра не допускается, то есть нельзя создать подпрограмму вида

```
procedure Sort(Arr: array[1..10] of Integer);
```

Для функций помимо параметров нужно указать еще тип возвращаемого значения. Чтобы передать результат функции вызывающей подпрограмме (программе) используется оператор присваивания, в левой части которого стоит имя функции или зарезервированный идентификатор `Result`.

Виды параметров

Вернемся к проблеме передачи в подпрограмму параметра большого размера. Мы выяснили, что передача его по значению приводит к существенным накладным расходам, а передача по ссылке может привести к порче данных внутри подпрограммы, и, что самое неприятное, компилятор не сможет помочь нам обнаружить этот факт. Что же делать? На помощь приходит возможность языка Object Pascal объявлять параметры как константы:

```
type
  Numbers = array[1..10000] of Integer;
  Complex = record
    Re:Real;
    Im: Real;
  end;

function Min(const a, b: Integer): Integer;     { передача по значению }
procedure Print(const a: Numbers);              { передача по ссылке   }
function Mult(const c1, c2: Complex): Complex; { передача по ссылке   }
```

Обсудим, что происходит с *константными параметрами* при вызове подпрограммы.

Прежде всего, отметим, что внутри подпрограммы значение параметра-константы изменено быть не может вне зависимости от его типа. Компилятор “зорко” следит за этим.

Что касается способа передачи, то для простых типов данных, таких, как в функции `Min`, параметры-константы передаются по значению, то есть для них создается копия. Для сложных типов данных (массивов, записей) передача параметров-констант осуществляется по ссылке. Таким образом, модификатор **`const`** на пару с компилятором решают проблему эффективной и безопасной передачи в подпрограммы параметров большого размера.

Рассмотрим еще одну задачу. Пусть нам требуется функция, заполняющая массив типа `Numbers` начальными значениями. Пишем функцию:

```
procedure Fill(var a: Numbers; Value: Integer);
var
```

```
    i: Integer;
  begin
    for i := 1 to 10000 do
      a[i] := Value;
  end;
```

А теперь представим себе, что в большинстве случаев инициализирующее значение будет равно нулю и лишь иногда другому значению. Хотелось бы иметь возможность сократить вызов процедуры `Fill`, не указывая второй параметр, если его значение должно быть равно нулю.

Такая возможность обеспечивается *параметрами по умолчанию*. Чтобы задать значение параметру, которое будет использоваться по умолчанию при вызове, при описании этого параметра ставится знак "=" и указывается конкретное значение:

```
procedure Fill(var a: Numbers; Value: Integer = 0);
```

В результате следующие два вызова процедуры `Fill` будут эквивалентными:

```
Fill(Arr, 0);
Fill(Arr);
```

Иметь значения по умолчанию могут только параметры-значения и параметры-константы. У подпрограммы может быть несколько аргументов со значениями по умолчанию. При этом действует следующее ограничение: все такие аргументы должны находиться в конце списка параметров.

Последний из специальных видов параметров, который мы обсудим – *открытые массивы*.

Допустим, мы хотим написать подпрограмму сортировки массива целых чисел. При ее объявлении мы должны будем указать некоторый заранее созданный тип-массив. В соответствие с требованием идентичности формальных и фактических параметров при вызове этой подпрограммы мы можем использовать лишь созданный нами тип. Очень неудобно! К тому же, при описании типа мы должны указать конкретный размер массива, что тоже сужает возможности по использованию данной подпрограммы. В то же время, очевидно, что алгоритм сортировки не зависит ни от типа массива, ни от числа элементов в нем. Открытые массивы языка Pascal позволяют разрешить эту ситуацию.

Синтаксически объявление параметра как открытого массива производится конструкцией вида:

```
array of <тип элементов>
```

Таким образом, процедуру сортировки мы можем объявить как:

```
procedure Sort(var a: array of Integer);
```

Нумерация элементов открытого массива начинается с нуля. Номер последнего элемента можно узнать с помощью функции

```
High(<имя открытого массива>)
```

При вызове подпрограммы с параметром открытым массивом в качестве фактического параметра может быть подставлен любой массив с соответствующим типом элементов, а также простая переменная того же типа.

Открытый массив может являться параметром-значением, параметром-переменной и параметром-константой.

В окончании раздела приведем пример реализации процедуры сортировки.

```
{ ================================================================== }
{ Пример 7.2 }
```

```
{ Процедура сортировки с параметром – открытым массивом }
procedure Sort(var a: array of Integer);
var
  i, j: Integer;
  temp: Integer;
begin
  for i := 1 to High(a) do
    for j := High(a) downto i do
    if a[j] < a[j - 1] then
    begin
      temp := a[j];
      a[j] := a[j - 1];
      a[j - 1] := temp;
    end;
end;
{ ========================================================================== }
```

Внутренние подпрограммы. Область действия имен

Внутренние процедуры и функции в языке Pascal описываются в разделе объявлений блока. Причем здесь речь идет как о блоке основной программы (начинающейся со слова **Program**), так и о блоке любой процедуры или функции. Другими словами, подпрограммы могут быть описаны внутри других подпрограмм.

Рассмотрим пример описания внутренних подпрограмм.

```
{ ========================================================================== }
{ Пример 7.3 }
{ Внутренние подпрограммы }
Program P1;

{$APPTYPE CONSOLE}

var
  a, b: Real;
  x: Boolean;
  i: Integer;

  procedure P2;
  var
    a: Integer;
    x: Boolean;

    procedure P4;
    type
      Complex = record
        Re, Im: Real;
      end;
    var
      a: Real;
      f: Complex;
```

```
    begin
      { Тело процедуры P4 }
      // a - 4, b - 1, x - 2, i - 1, f - 4
    end; { P4 }

  begin
    { Тело процедуры P2 }
      // a - 2, b - 1, x - 2, i - 1
  end; { P1 }

  procedure P3;
  var
    b: Integer;
    a: Real;
  begin
    { Тело процедуры P3 }
    // a - 3, b - 3, x - 1, i - 1
  end; { P3 }

begin
  { Тело программы P1 }
  // a - 1, b - 1, x - 1, i - 1
end.
{ ========================================================================== }
```

Главная программа `P1` содержит в своей декларативной части описание двух процедур `P2` и `P3`. Процедура `P2` в свою очередь содержит описание процедуры `P4`.

Написание текста программы и процедур может выполняться независимо, то есть в разные моменты времени, разными людьми. Но если принято решение, что все подпрограммы являются внутренними, то перед компиляцией всей созданной программной системы должна быть выполнена сборка текста (с помощью редактора текста) в единый исходный модуль в соответствии с синтаксисом языка и выбранными правилами вложенности подпрограмм.

Поскольку в рассматриваемом случае все модули подаются на вход транслятора одновременно, то он "видит" все используемые имена объектов программы. Поэтому центральным вопросом, связанным с использованием внутренних подпрограмм, является вопрос разделения имен между подпрограммами. Например: могут ли две подпрограммы использовать один и тот же идентификатор для обозначения разных объектов? Или, напротив: доступен ли некоторый объект (переменная, подпрограмма и т.п.), описанный в одной подпрограмме, для использования в другой подпрограмме?

Для однозначного разрешения этих и других подобных вопросов вводится система правил, определяющая *области действия* (доступности, "видимости") имен. Правила состоят в следующем. Объект, объявленный в некотором модуле, является *локальным* для этого модуля. Он "не виден" во всех других модулях (как "соседних", так и "объемлющих" данный), за исключением тех, которые описаны как внутренние подпрограммы данного модуля. Для этих внутренних подпрограмм объект является *глобальным*, то есть доступным для использования в том смысле, как он описан.

Однако если имя глобального объекта используется для описания другого объекта во внутренней подпрограмме, то вновь вступает в действие первая часть правила. Это значит, что имя вновь локализуется и бывший глобальный объект становится недоступен по "переобъявленному" имени внутри этой внутренней подпрограммы.

Проиллюстрируем правила на приведенном выше примере 7.3. Главная программа `P1` может использовать в своей исполняемой части обращения к процедурам `P2` и `P3`, но ни `P1`, ни процедура `P3` не могут вызвать процедуру `P4`, так как она локализована в процедуре `P2`. Процедура `P3` может обратиться к процедуре `P2`, но не наоборот, так как `P2` объявлена ранее `P3`.

Далее: поскольку переменные `a`, `b`, `x`, `i` описаны в главной программе, они являются глобальными для всех внутренних модулей. С другой стороны, во внутренних подпрограммах имеются переобъявления имен, что локализует их. В приведенной схеме около имен стоит номер модуля, в смысле объявления которого следует понимать переменную. Например, одно и тоже имя `a`, объявлено во всех модулях. Это означает, что имеется четыре разных переменных с именем `a`, причем компилятор однозначно определяет, где о какой переменной идет речь, на основании вышеизложенных правил. Дополнительно заметим, что объявленный в `P4` тип `Complex`, а также переменная `f` недоступны для использования нигде, кроме процедуры `P4`.

На основе вышеизложенного может сложиться впечатление, что в результирующем загрузочном модуле под все локальные и глобальные переменные будут отведены свои области памяти, и, например, в памяти, занимаемой программой `P1`, будет выделено 8 байт под вещественное `a` главной программы, 4 байта под целое `a` процедуры `P2`, и еще два раза по 8 байт под вещественные `a`, принадлежащие процедурам `P3` и `P4`. Это не так. Язык Pascal обладает механизмом динамического размещения переменных. Суть его состоит в том, что переменная существует (то есть реально размещается в оперативной памяти) только в то время, когда является активным блок, где она объявлена. Блок считается активным после начала его выполнения и до завершения его работы. Другими словами, память под переменные отводится только после входа в подпрограмму, где они описаны, и отбирается у них после завершения работы подпрограммы. Заметим, что переменные, объявленные в главной программе, существуют все время работы программы (для них отводится специальная область исполняемого модуля).

Понимание данного метода управления памятью очень важно, так как в противном случае могут возникнуть необоснованные надежды на сохранение записанных в локальные переменные значений после завершения работы соответствующей подпрограммы. Например, требуется разработать модуль, который, в частности, должен подсчитывать число обращений к нему. Если программист объявит счетчик вызовов, как локальную переменную, и будет надеяться, что при каждом новом входе в подпрограмму в счетчике сохранится прежнее значение, к которому достаточно лишь прибавить 1, то в общем случае это приведет к ошибке. Мы пишем “в общем случае”, поскольку случайно может оказаться, что система управления памятью каждый раз отводит под счетчик одно и то же место в памяти, и оно в промежутках между работой модуля не отдается под другие переменные.

Главным достоинством динамического распределения памяти под переменные является снижение затрат памяти. Однако имеется и существенный недостаток – дополнительное время на работу системных процедур управления памятью, а также дополнительные затраты памяти для размещения этих процедур.

Использование глобальных переменных позволяет организовать обмен информацией между модулями помимо аппарата параметров. С одной стороны, эта возможность облегчает запись текста программы, так как отпадает необходимость постоянно переписывать длинные списки фактических параметров сложных модулей. С другой стороны, этот способ “размывает” межмодульный интерфейс, делает его неявным, что может породить ошибки.

В общем случае можно сформулировать рекомендацию, состоящую в следующем: используйте глобальные переменные только тогда, когда без них нельзя обойтись или когда их использование имеет серьезные основания.

Побочный эффект

В языке Pascal при работе с подпрограммами возможно возникновение так называемого *побочного эффекта*.

Первое из его проявлений связано с тем, что, функция может возвращать значения не только стандартным для нее способом, но и через параметры, помеченные ключевым словом **var**. Например, процедура одновременного вычисления частного и остатка от деления целых чисел, приведенная выше, может быть переписана в виде функции:

```
{ ================================================================== }
{ Пример 7.4 }
{ Функция с побочным эффектом }
function DivMod(x, y: Integer; var m: Integer): Integer;
begin
  DivMod := x div y;
  m := x mod y;
end;
{ ================================================================== }
```

Функция `DivMod` в качестве “основного” результата возвращает частное, а “побочно” еще и остаток от деления.

Другой возможный побочный эффект связан с ошибками при работе с глобальными и локальными переменными и возникает при пропуске описания локальной переменной, имя которой совпадает с именем однотипной глобальной переменной. Компилятор в этой ситуации не в состоянии обнаружить ошибку. Рассмотрим пример программы сортировки двух чисел, которая использует процедуру перестановки значений двух переменных:

```
{ ================================================================== }
{ Пример 7.5 }
{ Еще раз побочный эффект }
program Sort;

{$APPTYPE CONSOLE}

var
  x, y :Real;

{ Перестановка значений двух переменных }
procedure Swap(var a, b: Real);
begin
  y := a;
  a := b;
  b := y;
end;

begin
  Writeln('Упорядочение двух чисел по возрастанию');
  Writeln('Введите два вещественных числа');
  Readln(x, y);

  if x > y then
```

```
    Swap(x, y);

  Writeln(x, y);
end.
{ ========================================================================== }
```

В этом примере грамотно написанная процедура перестановки значений двух переменных `Swap` содержит рабочую переменную `y`, имя которой "случайно" совпадает с именем глобальной переменной, обозначающей одно из сортируемых чисел. Как мы знаем, в этом нет ничего страшного, если рабочая переменная локализуется в подпрограмме соответствующим объявлением. Однако в нашем примере это не сделано. Разумеется, компилятор не выдаст сообщение об ошибке, так как имя `y` описано до его использования. При работе программы в случае, когда `x > y`, значение введенной переменной `y` будет заменено значением `x`, то есть произойдет *побочный эффект* при работе процедуры `Swap`.

Передача ссылки на модуль (процедурный тип данных)

Допустим, требуется составить подпрограмму печати значений произвольной функции для некоторого набора аргументов. Например, эта подпрограмма должна уметь печатать "Таблицу Брадиса" для выбранной элементарной функции: `Sin(x)`, `Cos(x)` и других. Подумаем над реализацией этой подпрограммы. Для начала вопрос: должна ли подпрограмма печати сама вычислять значения функций? Очевидно, что нет. Задачи расчета значений функции и печати этих значений, вообще говоря, совершенно независимы, так что разумно их разделить. Тогда возникает вопрос номер два: как передать значения функции в подпрограмму печати?

Возможно решение – предварительно вычислить все требуемые значения, записать их в массив и передать его как фактический параметр. Надеемся, что Читателю очевиден существенный недостаток этого подхода. Для размещения массива требуется память, в то время как на самом деле целиком он нам не нужен (по крайней мере, об этом в условии задачи ничего не было сказано) – для печати достаточно иметь только текущее значение функции. Итак, от идеи рассчитать все значения заранее отказываемся – цикл печати будет вызывать расчетную подпрограмму с конкретным значением аргумента. Остается последний вопрос: как сообщить этой расчетной подпрограмме, для какой функции производить вычисления?

Ответ на этот вопрос подводит нас к важному понятию межмодульного интерфейса: к ситуации, когда в модуль нужно передать не значение некоторого параметра, а правило (алгоритм) его вычисления. Другими словами, нам требуется одному модулю сообщить имя другого, которым следует воспользоваться для получения требуемых данных.

Для решения этой проблемы язык Pascal включает возможность конструирования специального типа данных – *процедурного типа*. Значениями этого типа являются процедуры и функции, описанные в некотором блоке. Процедурный тип так же, как и массивы или записи, – это некоторый класс типов. Конкретный процедурный тип характеризуется видом заголовка подпрограммы, включающего описание параметров. Общий вид объявления процедурного типа представлен ниже:

```
type
  <имя типа> = <заголовок функции или процедуры без имени>;
```

Например, следующее объявление процедурного типа

```
type
  FuncType = function(x :Real): Real;
```

задает класс всех вещественных функций с одним вещественным аргументом. “Константой” этого типа может быть функция с любым именем (а также любым другим именем формального параметра), но с такой же структурой заголовка и типами аргумента и возвращаемого значения. Например, функция

```
function Pol3(z: Real): Real;
begin
  Pol3 := Sqr(z) * z - 3.5 * Sqr(z) + 6.7 * z - 70.4;
end;
```

принадлежит типу `FuncType`.

Синтаксически параметр процедурного типа в списке формальных параметров подпрограммы описывается так:

```
<идентификатор>: <имя процедурного типа>;
```

Например:

```
procedure PrintTable(Func: FuncType);
```

В теле подпрограммы идентификатор рассматривается как имя процедуры или функции, список формальных параметров которой указан в объявлении процедурного типа.

Существует важное правило оформления подпрограмм, передаваемых в качестве параметров, касающееся стандартных подпрограмм, содержащихся в библиотеках системы Pascal таких, например, как `Sin(x)`, `Cos(x)`. Их имена нельзя непосредственно передавать в качестве параметров. Если возникает такая необходимость, то следует оформить собственную подпрограмму, в которую как в капсулу заключить обращение к стандартной подпрограмме. Например, для `Sin(x)` можно записать подпрограмму-оболочку вида:

```
function SinX(x: Real): Real;
begin
  SinX := Sin(x);
end;
```

и в качестве фактического параметра использовать имя `SinX`.

Ниже приведен пример программы, содержащей процедуру печати таблицы значений произвольной функции (принадлежащей к ранее рассмотренному типу), задаваемой как параметр. Значения функции вычисляются в N + 1 равноотстоящих точках отрезка [a, b].

```
{ ========================================================================== }
{ Пример 7.6 }
{ Табулирование функций }
{ Передача имени функции в качестве параметра }
program Table;

{$APPTYPE CONSOLE}

type
  FuncType = function(x: Real): Real;

function Pol3(z: Real): Real;
begin
```

```
  Pol3 := Sqr(z) * z - 3.5 * Sqr(z) + 6.7 * z - 70.4;
end;

function SinX(x: Real): Real;
begin
  SinX := Sin(x);
end;

{ Процедура печати таблицы значений функции }
procedure PrintTable(a, b: Real; N: Word; Func: FuncType);
var
  i: Word;
  Step, x, y: Real;

begin
  Step := (b - a) / N;
  x := a;
  Writeln('  X     Y   ');
  for i := 1 to N do
  begin
    y := Func(x);
    Writeln(x:5:2, ' ', y:9:2);
    x := x + Step;
  end;
  y := Func(b);
  Writeln(b:5:2, ' ', y:9:2)
end;

begin
  PrintTable(2.4, 10.6, 10, Pol3);
  Readln;
  PrintTable(2.4, 10.6, 10, SinX);
  Readln;
end.
{ ========================================================================== }
```

В этом примере для печати таблицы необходимо в теле программы записать оператор вызова процедуры `PrintTable`, в котором требуется явно указать имя подпрограммы, реализующей функцию, значения которой нас интересуют. При этом все такие функции должны быть заранее написаны в виде подпрограмм и включены в общую программу.

Приведенная программа, конечно, является чисто тестовой. В общем случае программа печати таблиц значений функций должна предоставлять средства ввода функций в привычном всем формульном виде. Достижение этого идеала – задача непростая. Язык Pascal, как, впрочем, и все другие широко распространенные языки программирования, не имеет встроенных средств представления и обработки формул, то есть средств аналитических преобразований формул. Для этого существуют специализированные пакеты программ (Maple, Mathcad, Mathematica). Конечно, средствами языка Pascal можно реализовать подобную систему, но это отдельная сложная задача, требующая специальных знаний. Рассмотрение соответствующей методологии выходит за рамки нашей книги.

Однако кое-что для частичного решения рассматриваемой проблемы мы все-таки можем сделать. Мы не можем справиться с вводом формулы функции, то есть все они должны быть запасены в виде подпрограмм. Но мы можем предоставить пользователю возможность выбирать требуемую функцию, вводя либо ее имя, либо номер.

Язык Pascal позволяет объявлять переменные процедурного типа. Мы можем сделать объявление вида:

```
var
  f: FuncType;
```

Далее этой переменной можно присваивать имена функций, принадлежащих типу `FuncType`, то есть допустимы операторы вида:

```
f := SinX;
f := Pol3;
```

Имя такой переменной может быть использовано при вызове соответствующей подпрограммы. Если, например, переменной `f` было присвоено одно из вышеприведенных значений, то можно написать оператор вида:

```
x := f(4.5);
```

Значения процедурного типа могут использоваться и при конструировании сложных типов. Например, можно объявить массив с элементами процедурного типа:

```
type
  ArrayOfFunc = array[1..2] of FuncType;
```

Мы можем запасти соответствующую типизированную константу:

```
const
  F: ArrayOfFunc = (SinX, Pol3);
```

Обращение к функции теперь может выглядеть как

```
x := F[1](4.5);
```

Усовершенствуем теперь программу из примера 7.5, позволив пользователю вводить номер функции, таблицу значений которой требуется напечатать. Номер функции будет выбираться из меню.

```
{ ========================================================================== }
{ Пример 7.7 }
{ Табулирование функций – вариант 2 }
{ Передача имени функции в качестве параметра }
program Table;

{$APPTYPE CONSOLE}

type
  FuncType = function(x: Real): Real;

var
  Key: Char;
  K: Byte;

function Pol3(z: Real): Real;
begin
  Pol3 := Sqr(z) * z - 3.5 * Sqr(z) + 6.7 * z - 70.4;
end;

function SinX(x: Real): Real;
begin
  SinX := Sin(x);
```

```
end;

const
   F: array[1..2] of FuncType = (SinX, Pol3);

{ Процедура печати таблицы значений функции }
procedure PrintTable(a, b: Real; N: Word; Func: FuncType);
var
  i: Word;
  Step, x, y: Real;

begin
  Step := (b - a) / N;
  x := a;
  Writeln('  X     Y   ');
  for i := 1 to N do
  begin
    y := Func(x);
    Writeln(x:5:2, ' ', y:9:2);
    x := x + Step;
  end;
  y := Func(b);
  Writeln(b:5:2, ' ', y:9:2)
end;

begin
  repeat
    Writeln('   Печать значений функции');
    Writeln('Функция                                        Клавиша');
    Writeln('Sin(x)                                            1');
    Writeln('Sqr(x) * x - 3.5 * Sqr(x) + 6.7 * x - 70.4        2');
    Writeln('Конец работы                                 Любая другая');
    Writeln;

    Readln(Key);
    if (Key < '1') or (Key > '2') then
      break;

    case Key of
    '1': K := 1;
    '2': K := 2;
    end;
    PrintTable(2.4, 10.6, 10, F[K]);
  until False;
end.
{ ========================================================================== }
```

Бестиповые параметры

Считая это как бы само собой разумеющимся, мы часто на протяжении книги пользовались очень удобным свойством некоторых операций и стандартных подпрограмм, состоящим в том, что один и тот же знак операции или одно и то же имя подпрограммы можно записывать в

совокупности с операндами (аргументами) разных типов. Например, вещественная и целая операция сложения, а также операция объединения строк имеют одно и то же обозначение “+”; функция возведения в квадрат `Sqr(X)` может применяться как к целому, так и вещественному аргументу, причем выдает результат того же, что и аргумент, типа и т.д. Такое свойство операции называется *полиморфизмом*.

Оставляя в стороне вопрос о реализации полиморфных операций над предопределенными типами данных (это проблема разработчиков компилятора), задумаемся над тем, как можно самому реализовывать полиморфные подпрограммы. Выгоды такого подхода к созданию подпрограмм очевидны: если требуется реализовать некий алгоритм общего характера, было бы очень удобно иметь одну универсальную подпрограмму вне зависимости от типа параметров.

Однако на пути реализации полиморфных подпрограмм стоят серьезные трудности. Пусть мы хотим реализовать подпрограмму сложения последовательности чисел. Очевидно, что исполняемый код модуля, складывающего вещественные числа, должен содержать команду вещественного сложения, а целые числа – команду сложения целочисленного. Значит, мы должны иметь два экземпляра кода модуля (или, по крайней мере, его части). Можно, конечно, предложить способ автоматического распознавания, какой вариант кода модуля нужен в данный момент. Правда, в этом случае возникает дополнительная проблема: когда выяснится тип складываемой последовательности – во время компиляции или во время исполнения программы? В первом случае связывание нужного кода может выполнить сборщик (редактор связей), во втором нужны дополнительные средства динамического связывания.

Язык Object Pascal предоставляет две возможности для реализации полиморфных подпрограмм. Одну из них мы разберем сейчас, о второй поговорим в разделе “Перегрузка подпрограмм”.

Итак, рассмотрим задачу сравнения двух переменных. Если мы полагаем, что переменные считаются равными при условии их побитового совпадения (что справедливо во многих случаях), то процедура сравнения может содержать одни и те же машинные команды для любого типа аргументов. Другими словами, вне зависимости от типа переменных-аргументов мы можем рассматривать их как последовательности переменных типа `Byte` и производить их побайтовое сравнение.

Обобщая вышесказанное, можно сказать, что, если удается предложить алгоритм, состоящий из одних и тех же машинных команд для разных типов аргументов, тогда тяжесть в реализации полиморфизма переносится в область синтаксического оформления того факта, что формальные параметры подпрограммы не связаны с определенным типом, а так же того, что фактические параметры интерпретируются внутри подпрограммы не так, как они объявлены в вызывающем модуле.

Язык Pascal имеет соответствующие средства.

Смена типа

Вначале разберемся в том, как можно указать, что двоичный код некоторой переменной будет интерпретироваться другим способом, нежели это предписано объявлением переменной. В языке имеется возможность *смены типа* переменной путем указания перед ней имени нового целевого типа (сама переменная при этом заключается в круглые скобки).

Пусть, например, переменная `c` объявлена как символ, то есть `Char`. Мы можем интерпретировать двоичный код символа, записанного в эту переменную, как целое число типа `Byte`, записав смену типа `Byte(c)`.

Важно понять, что смена типа не означает приведение типа. При смене типа не производится перекодировка значения. Другими словами, если мы записываем смену типа `A := Real(S)`, где `S` принадлежит типу `String[7]`, а `A` типу `Real`, то это не означает, что строка `S`, равная например `'-3.1416'`, будет преобразована к виду с плавающей точкой, и в `A` будет записано вещественное число `-3.1416`. Смена типа позволяет лишь интерпретировать двоичный код, представляющий строку `'-3.1416'`, как вещественное число. Заметим, что операция смены типа требует совпадения длин переменных исходного и целевого типа, поэтому в нашем примере строка имеет длину 7, что означает общую длину переменной 8 (за счет дополнительного байта, задающего текущую длину строковой переменной).

В качестве примера использования смены типа можно привести прием, позволяющий прочитать длину строковой переменной без использования функции `Length`. Длина строки, находящейся в данный момент в строковой переменной, содержится в самом первом байте переменной, который имеет номер ноль[1]. Доступ к этому байту можно получить обычным способом выделения символа строки, то есть с помощью селектора `S[0]`, где `S` – имя некоторой строковой переменной. Однако надо понимать, что количество символов строки закодировано в этом байте как целое число типа `Byte`, но имеет "родительский" тип `String`. Таким образом, если мы запишем оператор `L := S[0]`, где `L` – целая переменная типа `Byte`, то компилятор выдаст ошибку вида "`Type mismatch`" – "несовпадение типов". Заметим, что мы не можем воспользоваться и процедурой перевода строки в число `Val`, так как в этом байте находится число, а не символ. Выход состоит в смене типа. Оператор `L := Byte(S[0])` обеспечивает решение задачи.

Реализация полиморфной подпрограммы

Рассмотрим теперь, как возможность смены типа переменной используется при реализации полиморфных подпрограмм. Стандартные подпрограммы обработки строк типа `Pos`, `Val`, `Copy` и другие[2] в качестве параметров допускают строковые переменные любой длины. Эта интуитивно естественная возможность не согласуется с требованиями к соответствию формальных и фактических параметров, которые должны быть строго однотипными. Таким образом, если формальный параметр описан как `String`, а подставленный фактический параметр – как `String[20]`, то компилятор выдаст сообщение об ошибке (несовпадение типов). Мы рассмотрим, как можно обойти это препятствие, используя возможность смены типа и задания так называемых *бестиповых параметров* подпрограммы.

Бестиповые параметры могут быть *только* параметрами-переменными. Кстати, ответьте на вопрос, почему? Их запись в заголовке процедуры выглядит очень просто: после зарезервированного слова **`var`** следует имя формального параметра без указания его типа.

Важно то, что в теле подпрограммы бестиповые параметры не могут использоваться сами по себе. Каждое их вхождение в выражение или другое применение должно сопровождаться сменой типа, то есть указанием, как в данной конкретной ситуации интерпретировать двоичный код значения параметра.

Пусть требуется реализовать целую функцию, подсчитывающую количество вхождений некоторого задаваемого как параметр подпрограммы символа в строку, также являющуюся

---

[1] В Delphi это правило справедливо только для типа ShortString

[2] более детально о функциях работы со строковыми типами мы поговорим в главе 9

параметром подпрограммы. Потребуем, чтобы эта функция была полиморфной в том смысле, что ее фактическим параметром может быть строка любого типа, то есть объявленная с любой длиной. Ниже приведена программа, решающая эту задачу.

```
{ ========================================================================== }
{ Пример 7.8 }
{ Использование бестиповых параметров и смены типа }
{ для реализации полиморфной процедуры              }
Program UntypedPar;

{$APPTYPE CONSOLE}

var
  S: String[20];
  S1: String[62];

{ Подсчет количества вхождений символа в строку }
function NumOfChar(var S; C: Char): Byte;
var
  Count, i, Len: Byte;
begin
  { Длина строки - фактического параметра }
  Len := Byte(ShortString(S)[0]);
  Count := 0;
  i := 1;
  while i <= Len do
  begin
    if ShortString(S)[i] = C then
      Count := Count + 1;
    Inc(i);
  end;
  NumOfChar := Count;
end;

begin
  S := 'Полиморфизм';
  Writeln(NumOfChar(S, 'о'));
  S1 := 'Borland Pascal';
  Writeln(NumOfChar(S1, 'a'));
  Readln;
end.
{ ========================================================================== }
```

Сделаем замечания по примеру. Бестиповым параметром функции `NumOfChar` является параметр `S` (исходная строка). В теле подпрограммы ему приписывается тип `ShortString(S)`. Синтаксис языка допускает после операции смены типа запись селектора элементов массивов или полей записей. Поэтому запись `ShortString(S)[i]`, используемая в функции для выделения символа из строки, является правильной. Ну и, наконец, обратим внимание, что мы воспользовались вышеизложенным приемом получения длины строки из ее первого байта: `Len := Byte(ShortString(S)[0])`.

Перегрузка подпрограмм

Вернемся к задаче о сложении последовательности чисел. Очевидно, что алгоритм решения этой задачи никак не связан с типом, с помощью которого мы будем представлять элементы последовательности. Однако реализовать лишь одну подпрограмму, с помощью которой можно найти сумму элементов последовательности, невозможно, хотя бы потому, что, как мы указывали ранее, команды, осуществляющие сложение целых и вещественных чисел в процессоре различны. Как же быть?

В данном случае существует, можно сказать, Соломоново решение. Мы все-таки вынуждены реализовать две подпрограммы сложения, однако язык Object Pascal позволяет нам назвать эти подпрограммы одним и тем же именем и тем самым создать иллюзию выполнению операции одной и той же командой. Эта возможность и называется *перегрузка подпрограмм*. Распознавание, какую из подпрограмм вызвать, мужественно берет на себя компилятор, используя для этого информацию о типах фактических параметров.

Синтаксически перегрузка оформляется с помощью ключевого слова **overload**.

```
  <описание подпрограммы>; overload;
```

Следующий пример демонстрирует создание и использование перегруженных подпрограмм.

```
{ ========================================================================== }
{ Пример 7.9 }
{ Перегрузка - сложение последовательности чисел }
Program Summa;

{$APPTYPE CONSOLE}

const
  NMax = 1000;
type
  IntArray = array[1..NMax] of Integer;
  RealArray = array[1..NMax] of Real;
var
  IntNum: IntArray;
  RealNum: RealArray;
  i: Integer;
  Sum: Real;

function SumOfSequence(Arr: array of Integer): Integer; overload;
var
  i: Integer;
begin
  Result := 0;
  for i := 0 to High(Arr) do
    Result := Result + Arr[i];
end;

function SumOfSequence(Arr: array of Real): Real; overload;
var
  i: Integer;
begin
  Result := 0;
```

```
  for i := 0 to High(Arr) do
    Result := Result + Arr[i];
end;

begin
  { Формирование последовательностей }
  Randomize;
  for i := 1 to NMax do
  begin
    IntNum[i] := random(1000);
    RealNum[i] := random(1000);
  end;

  { Расчет и вывод результатов }
  Sum := SumOfSequence(IntNum);
  Writeln('Сумма последовательности целых чисел: ', Sum:6:1);
  Sum := SumOfSequence(RealNum);
  Writeln('Сумма последовательности вещественных чисел: ', Sum:6:1);
end.
{ ========================================================================== }
```

Отметим, что мы использовали открытые массивы в качестве параметра функций `SumOfSequence`, чтобы сделать их максимально общими.

Ключевое слово **`overload`**, как видно из представленного кода, должно указываться для каждой подпрограммы, использующей один и тот же идентификатор в качестве имени.

Используя механизм перегрузки необходимо иметь в виду, что компилятор распознает подпрограммы только по списку типов параметров и никак не учитывает наличие и тип возвращаемого результата подпрограммы. Таким образом, нельзя перегрузить процедуру и функцию с одним и тем же списком типов параметров или две функции, отличающиеся лишь возвращаемым типом.

Следующие подпрограммы компилироваться не будут.

```
  function Cap(S: ShortString): ShortString; overload;
  begin
    ...
  end;

  procedure Cap(var Str: ShortString); overload;
  begin
    ...
  end;
```

Напротив, подпрограммы

```
  function Func(X: Real; Y: Integer): Real; overload;
  begin
    ...
  end;

  function Func(X: Integer; Y: Real): Real; overload;
  begin
    ...
  end;
```

будут успешно оттранслированы, поскольку списки типов их параметров различны – как видим, порядок типов в списке также имеет значение.

Подводя итог, отметим, что механизм перегрузки является весьма удобным средством, особенно активно использующимся в объектно-ориентированном программировании, речь о котором пойдет в главе 10.

Рекурсивные подпрограммы

Вспомним задачу о вычислении значения факториала из главы 5. Решение, которое мы тогда предложили, было основано на формуле N! = (N – 1)! * N. Если для факториала ввести обозначение $F_n = N!$, то формулу можно будет переписать так $F_n = F_{n-1} * N$. Подобного вида формулы, где значение на очередном шаге определяется через значение на предыдущем, известны нам еще со школы. Вспомним, например, арифметическую и геометрическую прогрессии. Такая запись зависимостей называется в математике *рекурсивной*. В общем случае можно говорить не только о рекурсивных формулах расчетов, но и о рекурсивных алгоритмах, когда результат на очередном шаге алгоритма получается на основе результата предыдущего шага, причем выполнение каждого шага происходит по одной и той же схеме. Не правда ли, сказанное весьма похоже на определение цикла? Так и есть. Одним из способов программирования рекурсивных алгоритмов является использование циклов. Это не всегда так просто, как в примере с факториалом, но всегда возможно. Использование подпрограмм позволяет запрограммировать рекурсивные алгоритмы еще одним способом.

В общем смысле идея выглядит довольно просто. Отдельный шаг рекурсивного алгоритма оформляется в виде подпрограммы. В тот момент, когда необходимо получить результат предыдущего шага, подпрограмма вызывает сама себя с соответствующими значениями параметров. Единственное, о чем необходимо позаботиться, – в нужный момент прервать эту цепочку вызовов. Подпрограмма, устроенная по изложенной схеме и называется *рекурсивной*.

Прежде чем обсуждать технические детали организации таких подпрограмм в языке Pascal, посмотрим пример:

```
{ ========================================================================== }
{ Пример 7.10 }
{ Рекурсия – вычисление факториала }
Program FactorialNew;

{$APPTYPE CONSOLE}

var
  N: Word;

function Factorial(N: Word): Longword;
begin
  if N = 0 then
  begin
    Result := 1;
    Exit;
  end
  else
    Result := N * Factorial(N - 1);
```

```
end;

begin
  Write('Аргумент: ');
  Readln(N);
  WriteLn(N, '! = ', Factorial(N));
  ReadLn;
end.
{ ========================================================================== }
```

Как видим, рекурсивная функция вычисления факториала не отличается большой сложностью. Строка

```
  Result := N * Factorial(N - 1);
```

реализует формулу расчета, а условие

```
  if N = 0 then
```

не дает рекурсии стать бесконечной и устанавливает начальное значение. Несмотря на простоту, данная функция дает точное представление об общей схеме любой рекурсивной подпрограммы.

А теперь обещанные технические подробности. Вызов любой подпрограммы сопровождается целым рядом обслуживающих действий, код для которых автоматически генерирует компилятор. Во-первых, в месте вызова подпрограммы компилятор вставляет команду передачи управления на адрес, с которого начинается код подпрограммы. Во-вторых, необходимо где-то запомнить *адрес возврата* – адрес инструкции, которая в коде вызывающего модуля расположена следующей за вызываемым. В-третьих, параметры подпрограммы, переданные по значению, нужно, как мы помним копировать, то есть выделять под них место. Область памяти, в которой выделяется это “место” (в ней же размещаются и локальные переменные подпрограммы), называется *стеком*. Адреса возврата запоминаются в нем же.

Учитывая сказанное, нетрудно догадаться, что для вычисления факториала использовать рекурсивную подпрограмму не нужно. Накладные расходы слишком велики в сравнении с реализацией с помощью цикла. На самом деле такая ситуация имеет место практически в любом случае. Единственным преимуществом рекурсии является более простой и естественно выглядящий код. Однако в некоторых случаях это преимущество может иметь решающее значение.

В качестве примера рассмотрим один из известных алгоритмов сортировки – *сортировку слиянием*.

Алгоритм основан на операции слияния двух отсортированных массивов в третий. Используются три индекса, по одному на каждый массив. Индекс в результирующем массиве на каждом шаге увеличивается на единицу. В исходных массивах элементы с текущими индексами сравниваются. Меньший элемент при сортировке по возрастанию (или больший при сортировке по убыванию) копируется в результирующий массив. Индекс в массиве, из которого выполнялось копирование, перемещается на следующий элемент.

В алгоритме сортировки слияние используется следующим образом. Массив условно делится пополам. Предполагая, что каждая половина уже упорядочена, выполняется их слияние. То же самое повторятся для каждой из половин, затем для четвертей и так далее, пока размер каждой из частей не станет равен единице. Исходя из описания алгоритма, очевидно, напрашивается рекурсивная реализация.

```
{ ========================================================================== }
{ Пример 7.11 }
```

```
{ Рекурсия - сортировка слиянием }
{ Основная подпрограмма }
procedure QSort(iStart, iEnd: Integer);
var
  iMiddle: Integer;
begin
  if (iStart = iEnd) then
    Exit;
  iMiddle := (iStart + iEnd) div 2;
  QSort(iStart, iMiddle);
  QSort(iMiddle + 1, iEnd);
  Combine(iStart, iMiddle, iMiddle + 1, iEnd);
end;
{ ========================================================================== }
```

Сортировка слиянием относится к числу “быстрых” сортировок, поэтому в названии процедуры использована буква Q – первая буква английского quick.

Для упрощения в реализации процедуры предполагается, что сортируемый массив является глобальной переменной.

```
{ ========================================================================== }
{ Пример 7.11 }
{ Рекурсия - сортировка слиянием }
{ Объявления }

{$APPTYPE CONSOLE}

const
  NMax = 5000;
type
  TArray = array [1..Nmax] of Integer;
var
  B, C: TArray;
{ ========================================================================== }
```

Осталось лишь написать процедуру слияния.

```
{ ========================================================================== }
{ Пример 7.11 }
{ Рекурсия - сортировка слиянием }
{ Слияние частей }
procedure Combine(iStart1, iEnd1, iStart2, iEnd2: Integer);
var
  i, j, k: Integer;
begin
  i := iStart1;
  j := iStart2;
  k := iStart1;
  while ((i <= iEnd1) and (j <= iEnd2)) do
  begin
    if B[i] <= B[j] then
    begin
      C[k] := B[i];
      Inc(i); Inc(k);
    end
```

```
    else begin
      C[k] := B[j];
      Inc(j); Inc(k);
    end
  end;
  if (i > iEnd1) then
    while (j <= iEnd2) do
    begin
      C[k] := B[j];
      Inc(j); Inc(k);
    end;
  if (j > iEnd2) then
    while (i <= iEnd1) do
    begin
      C[k] := B[i];
      Inc(i); Inc(k);
    end;
  for i := iStart1 to iEnd2 do
    B[i] := C[i];
end;
{ ========================================================================== }
```

Надеемся, что в особых комментариях представленный код не нуждается. Отметим лишь, что по исчерпании элементов одного из массивов, элементы оставшегося просто копируются в результирующий. Также заметим, что для сокращения места мы несколько отступили от принципа: “На каждой строке по одному оператору”.

В конце раздела приведем пример возможного использования написанной сортировки.

```
{ ========================================================================== }
{ Пример 7.11 }
{ Рекурсия - сортировка слиянием }
{ Тело программы }
begin
  Randomize;
  for i := 1 to Nmax do
    B[i] := random(1000);
  for i := 1 to Nmax do
    write(B[i], ' ');
  Readln;
  QSort(1, NMax);
  for i := 1 to Nmax do
    write(B[i], ' ');
  Readln;
end.
{ ========================================================================== }
```

Внешние подпрограммы

В наиболее полном объеме технологию модульного программирования поддерживает аппарат *внешних* подпрограмм. Только раздельная компиляция модулей обеспечивает возможность

организовать достаточно “распараллеленную” коллективную разработку проекта, скрытие деталей реализации и в целом реализовать в полной мере все преимущества модульного подхода.

Язык Pascal обладает замечательным высокотехнологичным аппаратом реализации принципа раздельной компиляции. Отдельно компилируемой частью программы на языке Pascal может быть не только отдельный функциональный модуль, а целая коллекция подпрограмм и, что очень существенно, данных или их описаний. Этот набор подпрограмм и данных оформляется в виде особой программной единицы – **Unit**. Это английское слово имеет много значений и довольно трудно переводится в данном контексте. Мы будем придерживаться уже использованного термина – *библиотека*, наиболее полно отражающего содержание данной программной конструкции.

Оформление библиотеки

Описание библиотеки состоит из четырех основных частей: *интерфейсной* секции, секции *реализации*, секции *инициализации* и секции *финализации*.

Интерфейсная часть библиотеки (помечаемая в тексте зарезервированным словом **interface**), по сути, представляет собой каталог того, что в ней находится: здесь перечисляются заголовки подпрограмм и записываются объявления типов, переменных, констант, которыми можно пользоваться в программе, присоединившей к себе библиотеку.

Секция реализации (**implementation**) содержит сами тексты подпрограмм. Однако она может содержать не только те подпрограммы, которые были вынесены в каталог – интерфейс библиотеки, но и вспомогательные, “невидимые” пользователю подпрограммы, необходимые для реализации основных модулей. Кроме того, в этой секции могут быть сделаны всевозможные объявления данных. Эти данные также не будут “видны” в программе, присоединившей библиотеку, но могут использоваться при реализации подпрограмм библиотеки.

Третья часть библиотеки – секция инициализации (**initialization**) – не является обязательной. Ее записывают, если перед началом работы всей программы необходимо присвоить начальные значения некоторым переменным, объявленным в библиотеке. Выполнение всего программного комплекса начинается не с передачи управления на первый оператор главной программы, как это можно было ожидать, а с выполнения секций инициализации всех присоединенных к этой программе библиотек.

Четвертая часть библиотеки – секция финализации (**finalization**) – также не обязательна. Более того, она может присутствовать, только если в библиотеке использовалась секция инициализации.

В целом синтаксис записи библиотеки выглядит следующим образом:

```
Unit <Имя библиотеки>;
interface
[uses ;]
...
implementation
[uses ;]
...
[initialization]
...
[finalization]
...
```

```
end.
```

В заголовке библиотеки указывается ее имя. В принципе это произвольный идентификатор, однако при его выборе необходимо учитывать следующие соображения. Как уже отмечалось, требование на присоединение библиотеки выражается с помощью записи зарезервированного слова **uses**, в котором записывается имя вызываемой библиотеки. Сборщик (редактор связей) интерпретирует это имя как имя файла, в котором расположена библиотека. Это имя имеет стандартное расширение .dcu (Delphi compiled unit) и образуется после трансляции дублированием имени исходного текстового файла библиотеки и заменой расширения .pas на .dcu. Имя же исходного файла задается средствами редактора текстов. Таким образом, желательно, чтобы имя библиотеки, задаваемое в заголовке, совпадало с именем текстового файла (разумеется, с его первой частью), в котором размещается библиотека. В противном случае необходимо дополнительно указывать, где находится библиотека.

После зарезервированного слова **interface**, начинающего интерфейсную часть библиотеки, могут быть указаны вспомогательные библиотеки, которые необходимо присоединить к данной. Это делается, если, например, в интерфейсной части определяются данные, при задании которых требуются понятия, определенные в других библиотеках.

Далее в интерфейсной части следуют описания констант, типов и переменных, которые будут доступны программе, присоединившей эту библиотеку.

И, наконец, в интерфейсной части приводятся заголовки подпрограмм. Еще раз подчеркнем, что сами алгоритмы, по которым работают подпрограммы, здесь не записываются.

Следующая часть библиотеки, начинающаяся с зарезервированного слова **implementation**, синтаксически похожа на обычный блок. В ней допускаются все объявления данных и записываются тексты подпрограмм, собственно и составляющих библиотеку, а также вспомогательные подпрограммы. Как видно из описания, секция реализации также может включать запрос на присоединение библиотек, которые необходимы для реализации подпрограмм.

Реализация подпрограмм, заголовки которых вынесены в интерфейсную секцию, имеет одну особенность. Допускается не записывать списки параметров в их заголовках. Они как бы копируются из интерфейсной секции. Однако пользоваться этой возможностью мы не рекомендуем, дополнительный контроль в данном случае лишним не будет: в случае несоответствия описаний заголовков компилятор сможет сообщить об ошибке.

Секция инициализации начинается с зарезервированного слова **initialization**. В этой секции можно разместить операторы, которые должны быть выполнены в программе, подключившей данный модуль. Например, это может быть открытие файлов, необходимых для работы подпрограмм библиотеки.

Секция финализации начинается с зарезервированного слова **finalization**. В ней можно поместить операторы, которые будут выполнены при завершении работы программы. Например, это может быть закрытие используемых в библиотеке файлов.

Завершается текст библиотеки словом **end**, после которого ставиться точка.

Квалифицированные идентификаторы

Принцип действия имен имеет свои особенности при использовании библиотек. Все идентификаторы, объявленные в интерфейсной секции библиотеки, становятся глобальными для

присоединившей ее программы, то есть они видны в главной программе и во всех внутренних подпрограммах. Вместе с тем, эти идентификаторы могут быть переобъявлены. В этом случае библиотечный идентификатор становится недоступным, однако к нему все же можно обратиться, использовав так называемый *квалифицированный идентификатор*. Квалифицированное имя состоит из двух частей, разделенных точкой. Первая часть – имя библиотеки, в которой объявлен идентификатор, вторая – собственно идентификатор.

Пример – библиотека матричных операций

Рассмотрим пример создания и использования библиотеки. Ниже приведен текст библиотеки, содержащей некоторые операции обработки матриц: ввод, вывод матрицы, сложение матриц и поиск максимального элемента в матрице. Пользователь библиотеки получает в свое распоряжение новый тип данных `Matrix`. Кроме того, он может пользоваться переменными `M` и `N`, задающими текущую размерность матрицы. Эти переменные получают начальные значения при инициализации библиотеки. В случае необходимости размерность может быть переопределена либо с использованием процедуры ввода `InDim`, либо, например, просто операторами присваивания.

```
{ ========================================================================== }
{ Пример 7.12 }
{ Библиотека матричных операций }
Unit Matr;
interface

const
  NRow = 10; { Максимально допустимое число строк }
  NCol = 10; { и столбцов                          }

type
  RowInd = 1..NRow; { Тип индекса строк }
  ColInd = 1..NCol; { Тип индекса столбцов }
  Matrix = array[RowInd, ColInd] of Real; { Тип матриц }

var
 M: RowInd; { Текущие значения числа строк }
 N: ColInd; { и столбцов                   }

procedure InDim;
procedure InMatrix(var A: Matrix);
procedure OutMatrix(A: Matrix);
function MaxElement(A: Matrix) :Real;
procedure PlusMatrix(A, B: Matrix; var C: Matrix);

implementation

var
 i: RowInd;
 j: ColInd;

{ Ввод размерностей матрицы (числа строк и столбцов) }
procedure InDim;
begin
```

```
  {$R+ автоматическая проверка диапазона допустимых значений }
  Readln(M, N);
  {$R-}
end;

{ Ввод матрицы }
procedure InMatrix(var A: Matrix);
begin
  for i := 1 to M do
    for j := 1 to N do
      Read(A[i, j]);
  Readln;
end;

{ Вывод матрицы }
procedure OutMatrix(A: Matrix);
begin
  for i := 1 to M do
  begin
    for j := 1 to N do
      Write(A[i, j]:5:2,' ');
    Writeln;
  end;
end;

{ Нахождение максимального элемента матрицы }
function MaxElement(A: Matrix): Real;
var
  Max: Real;
begin
  Max := A[1,1];
  for i := 1 to M do
    for j := 1 to N do
      if A[i, j] > Max then
        Max := A[i, j];
  MaxElement := Max;
end;

{ Сложение матриц }
procedure PlusMatrix(A, B: Matrix; var C: Matrix);
begin
  for i := 1 to M do
    for j := 1 to N do
      C[i, j] := A[i, j] + B[i, j];
end;

initialization
  { Инициализация: задание размерностей матрицы "по умолчанию" }
  M := 2;
  N := 3;
end.
{ ================================================================== }
```

В следующей программе, использующей вышеприведенную библиотеку, следует обратить внимание на квалифицированное имя `Matr.M`. Необходимость в нем возникает, поскольку в

программе определена переменная M для обозначения максимального элемента матрицы, имя которой совпадает с именем переменной, обозначающей число строк матрицы.

```
{ ========================================================================== }
{ Пример 7.12 }
{ Использование библиотеки Mtr }
program MatrixProc;
uses Matr;
var
  X, Y, Z: Matrix;
  M: Real;  { Максимальный элемент матрицы }

begin
  Write('Размерности матрицы по умолчанию: ');
  Writeln(Matr.M, ' на ', N);

  Writeln('Введите матрицу ...');
  InMatrix(X);

  M := MaxElement(X);
  Writeln('Максимальный элемент матрицы: ', M:5:2);

  Writeln('Введите матрицу ...');
  InMatrix(Y);

  PlusMatrix(X, Y, Z);

  Writeln('Сумма первой и второй матриц:');
  OutMatrix(Z);

  Readln;
end.
{ ========================================================================== }
```

Общие принципы сборки многомодульной программы

Итак, мы провели анализ предметной области, выделили набор процедур и функций для обработки данных, рассортировали подпрограммы по библиотекам и, наконец, реализовали их. Как теперь получить из всего этого богатства работающую программу? Говоря другим словами, как ее “собрать”? Обсудим.

Прежде всего, если оба возможных предложения **uses** библиотеки пусты, то она может быть оттранслирована как самостоятельная единица. В результате получится объектный файл с расширением .dcu в Delphi или tpu в версии Borland Pascal 7.0.

Далее, если библиотека подключает другие модули, для ее трансляции предварительно должны быть собраны они. Точнее это требование является жестким в Borland Pascal 7.0, а в Delphi компилятору достаточно наличия файла с исходным кодом подключаемой библиотеки. Конечно, если Вы предоставите ему файл объектный, он тоже не обидится.

В реальной практике прикладного программиста необходимость в трансляции библиотек по отдельности возникает не так уж часто. Обычно они транслируются непосредственно вместе с основной программой, для которой собственно и были написаны. Поскольку, как мы уже отмечали, модульный принцип разработки программ используется повсеместно, все интегрированные среды предоставляют специальные механизмы для управления всем тем, из чего, в конечном счете, складывается готовая программа. Чаще всего основным элементом этого механизма является понятие *проекта*. В общем смысле проект – это совокупность файлов, необходимых для сборки исполняемого модуля (а также динамической или статической библиотеки), плюс набор файлов, в которых сохраняется информация о настройках интегрированной среды для данного проекта, порядке сборки, указаний компилятору и редактору связей. Обычно в наборе дополнительных файлов выделяется один главный, который именуется *файлом проекта*. Сам проект, по сути, можно считать некоторым логическим контейнером, содержимое которого составляет исходный материал для получения готовой программы.

Наличие проекта позволяет в большинстве случаев выполнять сборку программы нажатием пары кнопок. Более подробную информацию о содержимом проекта в Delphi и порядке его настройки можно найти в справочной системе.

Принципы модульной технологии в полном объеме используют и разработчики сред программирования. Как мы обсуждали еще в главе 2, любая интегрированная среда является весьма сложной системой, включающей в себя большое количество различных инструментов. Однако кроме такого функционального разбиения модульный подход используется и при реализации средств языка программирования. Существенная часть возможностей языка Object Pascal оформлена в виде подпрограмм, разбитых на некоторое количество библиотек, обычно именуемых *системными*, в том смысле, что их создателями являются разработчики среды программирования.

Использование системных библиотек, каждая из которых содержит довольно большое количество подпрограмм, предполагает решение одного важного вопроса: все ли подпрограммы библиотеки включаются в результирующий исполняемый модуль или только те, вызовы которых содержатся в программе явно?

Сборщик, входящий в систему Pascal, к счастью, обладает возможностью осуществлять “умное связывание”, то есть после привязки всей библиотеки в целом в исполняемый модуль не включается код неиспользуемых подпрограмм. С другой стороны, все информационные объекты (константы и переменные), объявленные в интерфейсной секции, остаются в исполняемом модуле вне зависимости от того, использовались они или нет.

Последний момент, который нам осталось здесь обсудить касается ситуации, когда библиотека `Lib1` нуждается в использовании библиотеки `Lib2`, а та в свою очередь “желает” использовать `Lib1`.

Как мы помним, библиотека содержит два предложения **`uses`**, одно в интерфейсной секции и одно в секции реализации. Правила, касающиеся циклического подключения библиотек в этих секциях разные.

В интерфейсной секции циклы в использовании библиотеками друг друга не допускаются, то есть следующий код вызовет ошибку компиляции.

```
Unit Lib1;
interface
uses Lib2; { Компиляция Lib1 не возможна, Lib2 содержит ссылку на Lib1 }
...
end.
```

```
Unit Lib2;
interface
uses Lib1; { Компиляция Lib2 не возможна, Lib1 содержит ссылку на Lib2 }
...
end.
```

Вызвано это требование тем фактом, что компилятор должен оттранслировать интерфейсную часть библиотеки прежде, чем другая библиотека сможет воспользоваться имеющимися в ней объявлениями. Например, в интерфейсной части могут быть объявлены глобальные переменные, под которые должна быть выделена память, – использовать эти переменные до момента получения ими действительного адреса, конечно же, невозможно.

Правило "отсутствия цикла" в последовательности подключения библиотеками друг друга распространяется на любую длину этого цикла. То есть ошибочной будет и схема: `Lib1` использует `Lib2`, `Lib2` использует `Lib3`, `Lib3` использует `Lib1`. Общий принцип подключения библиотек в интерфейсной секции может быть сформулирован следующим образом: *последовательность подключений библиотеками друг друга должна быть такова, чтобы компилятор смог найти порядок трансляции, при котором интерфейсная секция любого модуля транслируется прежде, чем используется*. Фактически, если построить граф подключений, где библиотеки являются узлами, а использование – ребрами, то этот граф должен представлять собой дерево.

В секции реализации напротив циклическое подключение вполне допустимо, что придает бо́льшую гибкость разработке модульной структуры программы.

Концепция нисходящего проектирования программы

Проектирование программной системы, как и любой другой вид деятельности по созданию нового объекта, является творческим процессом, который невозможно полностью формализовать и тем самым выдать проектировщику точную инструкцию. Вместе с тем, существуют общие фундаментальные подходы к организации этого творческого процесса, предлагающие как бы некоторую "дисциплину мышления", которая оказывается полезной в самых разных областях приложений, в том числе и в программировании.

Один из таких подходов – формирование базисного набора конструктивных элементов – уже рассматривался нами, причем в разных его применениях. На этом подходе основаны методологии модульного и структурного программирования. Своеобразной идейной надстройкой над ними является метод решения задачи "сверху-вниз", широко применяемый в настоящее время при проектировании программных систем.

Напомним, что центральная идея модульного программирования состоит в предложении разрабатывать программу по частям, оформляя их в виде операций более высокого (чем имеющийся базовый) уровня – модулей, с помощью которых можно выстроить алгоритмы решения для некоторого набора задач предметной области. В свою очередь, структурное программирование дает рекомендацию, как стандартным образом формулировать эти алгоритмы, предлагая три основных алгоритмических примитива: последовательность действий, выбор из нескольких вариантов действий и повторение действий (цикл). "Действиями" являются,

например, обращения к модулям. В общем же случае действие – один из упомянутых алгоритмических примитивов, который подставляется в другой примитив или сам в себя.

До настоящего момента мы оставляли в стороне вопрос о порядке подстановки. Более точно его можно сформулировать так: в какой последовательности производить сборку программы из действий? Здесь существует два крайних подхода.

Первый состоит в так называемом проектировании “снизу-вверх”. Представьте себе, что перед вами рассыпанный на полу детский конструктор, и вы пытаетесь собрать из его элементов одну из игрушек, изображенных в приложенной к конструктору инструкции. Беря детали, вы начинаете комбинировать их, постепенно наращивая заложенный поначалу фундамент. В некоторый момент может выясниться, что ранее было принято неверное решение, и продолжение сборки не приведет к получению желаемого результата. В этой ситуации необходимо вернуться на несколько шагов назад, демонтируя детали, и попытаться выбрать верный путь. Такой метод построения и называется разработкой “снизу-вверх”.

В приложении к проектированию программной системы его можно сформулировать следующим образом. Опираясь на базовый язык программирования или на уже разработанный модульный базис, проектировщик комбинирует его элементы одним из трех, определенных техникой структурного программирования способов, в попытке поэтапно построить алгоритм работы системы и выделить конструктивные элементы (модули) промежуточных уровней иерархии.

Другой подход, называемый проектированием “сверху-вниз”, мы проиллюстрируем на примере разработки чертежа некоторой конструкции. Вначале рисуется общий внешний облик объекта. Далее начинают прорисовывать его элементы, все более и более детально. Проектировщик стремится на нижнем уровне детализации получить как можно более “крупные” типовые подконструкции, которые соответствуют стандарту, а значит, могут быть закуплены на каком-нибудь предприятии. При некотором выбранном конструкторском решении может оказаться, что для построения объекта требуется уникальная деталь. Тогда следует принять решение либо о самостоятельном производстве детали, либо вернуться “наверх” по иерархии детализации и попробовать выбрать другой способ уточнения способа создания подконструкции.

Переходя к проблеме проектирования системы, мы скажем, что метод “нисходящего проектирования” (так часто называют метод проектирования “сверху-вниз”) представляет собой последовательность решений по декомпозиции задачи с применением на каждом этапе одной из базовых алгоритмических структур.

Обратим внимание на важный аспект проектирования, который до сих пор оставался вне нашего рассмотрения. Проектирование программы состоит не только в разработке ее процедурной части, но и в формировании информационной модели задачи, то есть в проектировании типов данных. В процессе нисходящей разработки осуществляется уточнение структуры данных с использованием заложенных в язык стандартных приемов конструирования новых типов. Таким образом, на каждом этапе уточняется вид конкретного обрабатываемого значения, например, оно определяется как запись, как массив и т.п.

Метод нисходящего проектирования, по-видимому, более адекватно отражает характер мыслительного процесса проектировщика, идущего в своих размышлениях от постановки общей задачи к деталям ее решения. Вместе с тем, разумеется, невозможно строго выдержать один из рассмотренных подходов, и в реальности происходит их смешение, когда человек, разрабатывающий программную систему, многократно возвращается вверх и вниз по иерархии решений, то производя декомпозицию задачи на части, то пытаясь скомбинировать решение подзадачи из элементарных заготовок.

Говоря о нисходящем проектировании, мы как бы отделяли этот этап во времени от остальных этапов создания программы, то есть записи ее на алгоритмическом языке и отладки. На самом же

деле, распространенным подходом является совмещение процессов проектирования, реализации и отладки, так что правильнее говорить о нисходящей (возможно коллективной) разработке программной системы.

Современные системы программирования направлены в том числе и на автоматизацию труда проектировщика программного комплекса. В целом, такие системы обеспечивают возможность одновременного выполнения на компьютере всех “программистских” (то есть, речь, конечно, не идет о разработке математических методов) этапов разработки системы. Другими словами, разработчику, например, предоставляется возможность (конечно, при соблюдении им правил технологии) начать отлаживать программу еще до того, как она не только не записана полностью на языке программирования, но даже и не спроектирована в целом.

Сразу, конечно, возникает вопрос: как можно отлаживать программу, если не все ее части реализованы? Основной прием, позволяющий решить эту проблему, состоит в следующем. Вместо модулей, которые еще не написаны, но к которым производится обращение из отлаживаемой части программы, подставляются так называемые *заглушки* или *имитаторы*, то есть “пустые” модули, имеющие соответствующий интерфейс, но не решающие в полном объеме свою подзадачу, а лишь каким-то образом имитирующие ее решение, например, возвращающие стандартные значения, заданные либо константами, либо с помощью датчика случайных чисел. При этом имитатор может выводить на экран дисплея какое-то сообщение, подтверждающее, что он успешно отработал.

Общие приемы нисходящей разработки программы и возможности интегрированной среды языка Pascal по поддержке этой методологии мы продемонстрируем на примере, рассмотрению которого посвящен следующий раздел.

Пример разработки программы “сверху–вниз”

Допустим, требуется составить программу, которая по заданной площади и периметру прямоугольного треугольника определяет его стороны. Такая постановка задачи может быть формализацией вопроса типа: если садоводу разрешено выбрать участок определенной площади, и оговорено, что в силу особенностей ландшафта он должен иметь форму прямоугольного треугольника, то может ли садовод обойтись имеющимся у него материалом для огораживания участка?

Математическая модель задачи записывается в виде приведенной ниже системы уравнений, где a, b – катеты треугольника, c – его гипотенуза, а S и P – площадь и периметр соответственно:

$$c^2 = a^2 + b^2$$
$$P = a + b + c$$
$$s = \frac{a \cdot b}{2}$$

Цель решения системы – найти a, b и c. После преобразований мы получим квадратное уравнение вида:

$$(2P) \cdot x^2 - (P^2 + 4S) \cdot x + 4P \cdot S = 0,$$

вещественное решение которого дает значения катетов a и b.

Если дискриминант уравнения меньше нуля, построить треугольник при данных значениях площади и периметра невозможно. Принимая во внимание постановку содержательной задачи,

можно потребовать, чтобы катеты были не меньше некоторой заранее оговоренной величины, например, одного метра.

Перейдем теперь к проектированию, реализации и отладке программы. Однако прежде сделаем замечание методического характера. Рассматриваемый пример, безусловно, представляет собой очень простую задачу, для решения которой в реальной практике, конечно, не потребовалось бы использовать весь арсенал методов нисходящей разработки. Точно так же на примере конструирования прогулочной лодки трудно продемонстрировать, как создаются авианосцы. В то же время разбор сколь-нибудь серьезного по своим масштабам примера представляется практически невозможным – за неизбежным обилием деталей потеряются принципиальные идеи. Таким образом, нам придется довольствоваться небольшой программой и не обращать внимания на явную избыточность иерархии проектных решений и, соответственно, многомодульность результирующей программы.

Итак, приступим.

На первом шаге проектирования алгоритм решения задачи формулируется как простая последовательность действий: ввести данные об объекте обработки – участке, выполнить расчет и вывести результат, который является частью информационной модели участка. Будем считать, что *техническое задание* на программу требует реализовать возможность повторных расчетов в течение одного сеанса работы с программой. Тогда только что рассмотренный алгоритм должен быть телом цикла, повторяющего расчеты до выдачи команды завершения работы. Эти проектные решения уже могут быть зафиксированы в виде программы на языке Pascal:

```
{ ========================================================================= }
{ Пример 7.13 }
{ Основная программа }
Program Sizes;

{$APPTYPE CONSOLE}

uses DataLib1, Lib1;
var
  Lot: LotType; { информационная модель участка }
begin
  repeat          { повторять                          }
    Input(Lot);   { ввод модели участка                }
    Solve(Lot);   { решение                            }
    Output(Lot);  { вывод результата                   }
  until Finish;   { до получения команды завершения }
end.
{ ========================================================================= }
```

В упомянутых в этой программе библиотеках `DataLib1` и `Lib1` разместим имитаторы подпрограмм и определений типов данных. В первой библиотеке будут строиться информационные модели. На этом шаге мы лишь сделаем предположение, что модель участка наиболее адекватно выражается типом данных *запись*:

```
{ ========================================================================= }
{ Пример 7.13 }
{ Библиотека информационных моделей – шаг 1 }
Unit DataLib1;
interface
```

```
type
  LotType = record
  end;

implementation

end.
{ ========================================================================== }
```

В библиотеке Lib1 реализованы имитаторы всех модулей, упомянутых в главной программе, причем функция выхода имитируется с помощью датчика случайных чисел:

```
{ ========================================================================== }
{ Пример 7.13 }
{ Библиотека подпрограмм - шаг 1 }
Unit Lib1;
interface
uses DataLib1;

procedure Input(var Lot: LotType);
function Finish: Boolean;
procedure Solve(var Lot: LotType);
procedure Output(Lot: LotType);

implementation

function Finish: Boolean;
begin
  Finish := (Random < 0.5);
end;

procedure Input(var Lot: LotType);
begin
  WriteLn('Данные об участке введены...');
end;

procedure Solve(var Lot: LotType);
begin
  WriteLn('Расчет выполнен... ');
end;

procedure Output(Lot: LotType);
begin
  WriteLn('Результаты выведены...');
end;

begin
  Randomize;
end.
{ ========================================================================== }
```

Хотя по сути решения задачи сделано довольно мало, мы, тем не менее, уже реализовали заметную часть программы и даже можем отлаживать не только ее синтаксис, но и семантику, по крайней мере в части правильности реализации цикла.

Следующие шаги разработки состоят в уточнении понятий, определенных в библиотеках. Для того чтобы отладка стала управляемой, нам, прежде всего, следует реализовать функцию `Finish`.

Ее алгоритм может состоять в следующем: запрашиваем, следует ли продолжать работу, до тех пор, пока пользователь не даст синтаксически правильный ответ:

```
{ ========================================================================= }
{ Пример 7.13 }
{ Функция завершения работы программы }
function Finish: Boolean;
var
  Key: Char;
begin
  Write('Еще? (Y-Да/N-Нет)');
  repeat
    Readln(Key);
  until (UpCase(Key) = 'Y') or (UpCase(Key) = 'N');
  Finish := UpCase(Key) = 'N';
end;
{ ========================================================================= }
```

Уточним алгоритмы других процедур. Для записи процедуры ввода модели участка следует, прежде всего, уточнить способ представления участка в программе. Возьмем в качестве модели набор величин, характеризующих размеры участка: длины сторон, площадь и периметр. Кроме того, имеется еще одна важная характеристика, указывающая на корректность заданных величин. Эту характеристику мы отразим с помощью булевского поля `Exists` (существует?). Таким образом, уточненная библиотека `DataLib1` имеет вид:

```
{ ========================================================================= }
{ Пример 7.13 }
{ Библиотека информационных моделей - шаг 2 }
Unit DataLib1;
interface
type
  LotType = record
    S,           { площадь    }
    P,           { периметр   }
    a, b,        { катеты     }
    c    :Real; { гипотенуза }
    Exists :Boolean; { признак корректности соотношения }
                     { геометрических характеристик     }
  end;

implementation

end.
{ ========================================================================= }
```

Процедура ввода модели участка должна запрашивать значения площади и периметра до тех пор, пока не будут введены синтаксически правильные значения (то есть числа, а не другие наборы символов). Кроме того, можно частично проверить семантику значений: они должны быть, по крайней мере, положительными.

Процедура расчета распадается на два последовательных действия – решение квадратного уравнения и вычисление размеров сторон.

Вывод состоит в печати значений сторон, если треугольник существует при исходных соотношениях площади и периметра, а также, если стороны удовлетворяют заданным пороговым значениям (выполнение всех этих условий отражается в поле `Exists`).

После этих уточнений библиотека `Lib1` принимает следующий вид:

```
{ ========================================================================== }
{ Пример 7.13 }
{ Библиотека подпрограмм – финальный вариант }
Unit Lib1;
interface
uses DataLib1;

procedure Input(var Lot: LotType);
function Finish: Boolean;
procedure Solve(var Lot: LotType);
procedure Output(Lot :LotType);

implementation
uses Lib2;

{ Запрос на завершение работы }
function Finish: Boolean;
var
  Key: Char;
begin
  Write('Еще? (Y-Да/N-Нет)');
  repeat
    Readln(Key);
  until (UpCase(Key) = 'Y') or (UpCase(Key) = 'N');
  Finish := UpCase(Key) = 'N';
end;

{ Ввод площади и периметра }
procedure Input(var Lot: LotType);
  { Проверка корректности данных }
  function IsDataRight :Boolean;
  var
    IOR: Word;
  begin
    IOR := IOResult;
    with Lot do
    begin
      if IOR <> 0 then
        Writeln('Ошибка 1: Неправильный формат вещественного числа')
      else
        if (P <= 0) and (S > 0) then
          Writeln('Ошибка 2: Недопустимые характеристики интервала');
      IsDataRight := (IOR = 0) and (P > 0) and (S > 0);
    end;
  end;

begin
  Writeln('Введите площадь и периметр участка');
  repeat
    {$I-}
```

```
    with Lot do
      Readln(S, P);
    {$I+}
  until IsDataRight;
end;

{ Решение }
procedure Solve(var Lot: LotType);
var
  Root: RootsType;  { корни уравнения }
begin
  Equation(Lot, Root); { решение квадратного уравнения  }
  Sides(Lot, Root);    { вычисление сторон треугольника }
end;

{ Вывод }
procedure Output(Lot: LotType);
begin
  with Lot do
    if not Exists then
      Writeln('При данных соотношениях площади и периметра',
        ' треугольник не существует')
    else
      Writeln('Катеты ', a:6:2, ' ',b:6:2, ' Гипотенуза ', c:6:2);
end;

end.
{ ========================================================================= }
```

Для того чтобы программу можно было отлаживать, необходимо реализовать имитаторы определения типа `RootsType`, а также процедур `Equation` и `Sides`. Определение типа мы добавим в библиотеку `DataLib1`, а имитаторы процедур поместим в новую библиотеку `Lib2`:

```
{ ========================================================================= }
{ Пример 7.13 }
{ Библиотека информационных моделей - шаг 3 }
Unit DataLib1;
interface
type
  LotType = record
    S,          { площадь    }
    P,          { периметр   }
    a, b,       { катеты     }
    c    :Real; { гипотенуза }
    Exists :Boolean; { признак корректности соотношения }
                     { геометрических характеристик     }
  end;

  RootsType = record
  end;

implementation

end.
{ ========================================================================= }
```

```
{ ========================================================================== }
{ Пример 7.13 }
{ Библиотека вспомогательных подпрограмм – шаг 1 }
Unit Lib2;
interface
uses DataLib1;

procedure Equation(Lot: LotType; var Root :RootsType);
procedure Sides(var Lot: LotType; Root :RootsType);

implementation

{ Имитатор решения уравнения }
procedure Equation(Lot: LotType; var Root: RootsType);
begin
end;

{ Имитатор вычисления сторон }
procedure Sides(var Lot: LotType; Root: RootsType);
begin
  with Lot do
  begin
    Exists := (Random < 0.5);
    a := 3;
    b := 4;
    c := 5;
  end;
end;

begin
  Randomize;
end.
{ ========================================================================== }
```

Написанная часть программы практически полностью отражает структуру программного комплекса. Определен интерфейс всех подпрограмм. Эту часть можно отлаживать и модифицировать, например, совершенствовать процедуры ввода и вывода. Но при этом мы фактически еще не приступали к реализации математического метода решения! Проделанная нами работа фактически демонстрирует содержание труда проектировщика программной системы.

Следующий шаг – уточнение процедур из библиотеки `Lib2`. Для этого следует разобраться со способом представления значений типа `RootsType`. Квадратное уравнение имеет два корня, которые мы обозначим как `X1` и `X2`. Нас интересуют только вещественные значения корней, поэтому мы введем логический признак `Complex`, истинное значение которого будет означать, что в нашем случае решение уравнения отсутствует. Кроме того, как мы договаривались ранее, следует определить пороговые значения для сторон треугольника. Мы введем их в виде поименованных констант. Окончательный вариант библиотеки `DataLib1` выглядит следующим образом:

```
{ ========================================================================== }
{ Пример 7.13 }
```

```
{ Библиотека информационных моделей - финальный вариант }
Unit DataLib1;
interface
const
  Min_a = 1;
  Min_b = 1;
  Min_c = 1;
type
  LotType = record
    S,          { площадь    }
    P,          { периметр   }
    a, b,       { катеты     }
    c    :Real; { гипотенуза }
    Exists :Boolean; { признак корректности соотношения }
                     { геометрических характеристик     }
  end;

  RootsType = record
    Complex: Boolean;
    X1, X2: Real;
  end;

implementation

end.
{ ========================================================================== }
```

Поскольку способ реализации процедур `Equation` и `Sides` очевиден, мы без дополнительных комментариев приведем окончательный вид библиотеки `Lib2`:

```
{ ========================================================================== }
{ Пример 7.13 }
{ Библиотека вспомогательных подпрограмм - финальный вариант }
Unit Lib2;
interface
uses DataLib1;

procedure Equation(Lot: LotType; var Root :RootsType);
procedure Sides(var Lot: LotType; Root :RootsType);

implementation

procedure Equation(Lot: LotType; var Root: RootsType);
var
  A, B, C, D: Real;
begin
  with Root do
  begin
    A := Lot.P + Lot.P;
    B := -(Sqr(Lot.P) + 4 * Lot.S);
    C := 4 * Lot.P * Lot.S;
    D := Sqr(B) - 4 * A * C;
    Complex := D < 0;
    if not Complex then
    begin
```

```
      X1 := (-B + Sqrt(D)) / (A + A);
      X2 := (-B - Sqrt(D)) / (A + A);
    end;
  end;
end;

procedure Sides(var Lot: LotType; Root: RootsType);
begin
  with Lot, Root do
  begin
    Exists := not Complex;
    if Exists then
    begin
      a := X1;
      b := X2;
      c := P - a - b;
      Exists := (c >= Min_c) and (a >= Min_a) and (b >= Min_b);
    end;
  end;
end;

end.
{ ========================================================================== }
```

Итак, сделаем некоторые выводы из рассмотренного нами примера нисходящей разработки программы и сформулируем ряд общих положений этой технологии.

Часто можно услышать мнение, что основная трудность решения задачи с использованием компьютера заключается в разработке и реализации математического метода. Не отрицая ни в коем случае основополагающую роль решения этой проблемы, следует отметить, что, как видно из примера, процесс проектирования и реализации частей, формально не относящихся к собственно расчетным процедурам, а также проектирования системы в целом занимает существенный, а часто и значительно больший объем работ.

Полученная в результате пошагового проектирования “сверху-вниз” структура программы может быть интерпретирована как иерархия инструментальных слоев или, как еще говорят, *виртуальных машин*. В нашем примере программа `Sizes`, можно сказать, реализована на виртуальной машине `Lib1` плюс `DataLib1`, а `Lib1` в свою очередь на виртуальной машине `Lib2`. Модификация программы может производиться путем замены виртуальной машины нижнего уровня без изменений в верхней части иерархии. Например, если мы решили усовершенствовать интерфейс с пользователем в нашей программе, нам достаточно переработать “машину” `Lib1`, в то время как головная программа остается без изменений.

Рассмотренная технология обеспечивает возможность параллельной коллективной разработки в целом. Хотя это, конечно, простой пример, но, тем не менее, отметим, что программу `Sizes` можно было начинать реализовывать и отлаживать даже до того, как придуман математический метод решения задачи.

Выводы

Настоящая глава посвящена рассмотрению одной из ключевых концепций в проектировании и разработке программ – модульному программированию. Основная идея модульного программирование – разбиение задачи на более простые, максимально независимые друг от друга подзадачи – модули, явилась мощным катализатором в развитии процесса коллективной разработки программ. Возможность создавать отдельные модули, впоследствии объединяя их в одну программу, привела к появлению технологий разделения труда между членами программистского коллектива, и, наконец-то, позволила перевести количество рабочих рук (или, что вернее, голов) в качество и быстроту разработки программного продукта.

В главе рассмотрены основные элементы технологии модульного программирования и их реализация в языке программирования Object Pascal. Изучены как базовые (понятие подпрограммы, модуля), так и сложные вопросы (процедурный тип данных, полиморфизм, нисходящее проектирование).

8. Методы работы с внешней памятью. Файлы

Где-то далеко в памяти моей…
Слова из песни.

Внимательный Читатель уже должен был отметить один известный методический принцип, формулируемый обычно как “повторенье – мать ученья”, который авторы не раз применяли на протяжении книги. Действительно, на многие важные положения мы сознательно обращали внимание неоднократно, иногда в точности повторяя первоначальную формулировку, иногда чуть ее видоизменяя. Хотим подчеркнуть, мы отнюдь не являемся сторонниками усваивания информации путем “зубрежки”, на таком подходе, на наш взгляд, далеко не уедешь. Понимание и умение формулировать – вот истинные показатели уровня владения материалом! К величайшему сожалению бесчисленных поколений школяров всех веков и эпох до сих пор не придумано способа гарантированного запоминания необходимой человеку информации. Вот и разрабатываем мы различные методики, мнемонические правила, целые школы тренировки памяти. Вот и изощряются фантасты, кто придумает наиболее эффектный и элегантный прибор, способный “затолкнуть” человеку в голову требуемые сведения. Ну а пока все эти сказки еще не стали былью, человечеству, как и во многих других случаях, пришлось искать решение в создании “дополнительного органа” для хранения информации – “внешней” памяти. Сначала наскальные рисунки, потом глиняные таблички и папирусы, затем пергаментные и бумажные книги, наконец, в последние полвека эту роль взяли на себя компьютеры. Сегодня средних размеров жесткий диск персонального компьютера способен вместить содержимое десятков тысяч книг. Впрочем, перевести информацию в электронный вид – только полдела. Гораздо важнее обеспечить возможность с этой информацией работать в привычном человеку виде. Как мы отмечали в главе 3, общепринятый способ организации информации в долговременной памяти – файлы. Работа с файлами составляет существенную часть необходимых умений квалифицированного программиста. В данной главе мы рассмотрим основные приемы работы с файлами, методы чтения и записи информации, а также соответствующий аппарат языка Pascal.

Файлы: основные понятия

Не вдаваясь в подробности физического устройства гибких, жестких, CD и DVD дисков, флэш-драйвов, ZIP и ZIV дисков, стримеров, и других внешних носителей заметим, что с общих позиций принципы представления информации в них и принципы доступа к ней идентичны. В любом устройстве информация представляет собой двоичные последовательности, доступ к которым осуществляется по физическим адресам. Производители устройств создают специальные программы – драйвера, обеспечивающие выполнение всех необходимых

низкоуровневых операций, а разработчики операционных систем предоставляют прикладным программистам высокоуровневый интерфейс, позволяющий оперировать с данными в терминах файлов. Напоминаем, под *файлом* понимается поименованный набор данных, размещенный во внешней[1] памяти компьютера. Очевидно, что над файлом нужно уметь производить как минимум два основных действия: писать в него и читать из него. Также очевидно, что файл должен иметь некоторую внутреннюю структуру, отражающую содержание хранимой в нем информации. Разберем эти моменты более подробно.

Записи файлов

Структурную единицу информации внутри файла, принято называть *записью* (не следует путать это понятие с типом данных "запись" языка Pascal). Разделение файла на записи производится по двум причинам, первая из которых связана с физическими принципами представления данных и обмена информацией между оперативной и внешней памятью, а вторая – с тем или иным взглядом на логическую структуру информации файла.

Физические записи (их еще иногда называют *блоками*) представляют собой атомарную часть файла с точки зрения обмена с оперативной памятью. Другими словами, за один раз записывается на диск или считывается с диска ровно одна физическая запись. Количество физических записей в файле и размер каждой из них в общем случае задаются программистом средствами языка программирования.

Все записи файла могут иметь либо одинаковый размер, то есть одинаковое число байт, так называемые *записи фиксированной длины*, либо иметь разные размеры, *записи переменной длины*.

В различных языках программирования реализованы разные способы задания характеристик файла. Смысл вариаций состоит в том, что программисту либо даются средства прямо указать размеры записей и их число (и это он должен делать при объявлении каждого файла), либо применяется система умолчаний, при которой характеристики файла связываются с его типом. В языке Pascal можно найти как первый способ: так называемые, *бестиповые файлы*, – так и второй: *текстовые* и *типизированные файлы*, – определения физических характеристик файла.

Разбиение файла на *логические записи* зависит от структуры информации. Это разбиение явным образом не отражается в представлении данных на носителе, а фиксируется в алгоритме обработки файла. При этом в различных контекстах обработки строение файла может считаться разным. Скажем, в файле, содержащем информацию о фамилиях, должностях и зарплатах сотрудников некоторого предприятия, в качестве логической записи в одном случае может рассматриваться строка из трех полей, а в другом нас могут интересовать лишь фамилия и должность конкретного работника.

В зависимости от потребностей обработки возможны различные соотношения между логическими и физическими записями. Во-первых, они могут совпадать. Во-вторых, одна логическая запись может представлять последовательность физических записей – в этом случае говорят, что логическая запись сегментирована. И, наконец, возможно, что в одной физической записи размещается несколько логических записей – это называется блокированием логических записей. Некоторые языки программирования имеют средства явной установки соотношения

[1] в данном случае под "внешней памятью" мы понимаем любые средства хранения информации, кроме оперативной памяти

между физическими и логическими записями. В языке Pascal такого аппарата фактически нет, что ни в коей мере не затрудняет программирование, так как эта проблема решается сама собой выбором того или иного типа файла.

Физический и логический файл. Связывание

В зависимости от контекста употребления термин “файл” может нести две различные смысловые окраски. Проиллюстрируем их на примере.

Допустим, наша программа предназначена для бухгалтерских расчетов. Одним из объектов обработки программы является список сотрудников предприятия, включающий фамилии и заработные платы. Указанная информация по ряду подразделений хранится в виде файлов. Когда мы говорим о файле данных по конкретному подразделению, мы имеем в виду конкретный список фамилий и значений зарплаты, размещенный на конкретном физическом носителе и имеющий заданное имя. В этом случае часто употребляют термин *физический файл*. В то же время, для составления программы бухгалтерских расчетов существенны не конкретные фамилии сотрудников и их заработные платы, а структура файла и формат его компонент, то есть логическое описание файла. В этом контексте говорят о *логическом файле*. Здесь вполне уместна аналогия с константой: физический файл, – и переменной: логический файл.

Программа, ориентированная на обработку некоторого логического файла, в процессе выполнения имеет дело с одним из его экземпляров, то есть физическим файлом. Перед началом работы программы должно быть осуществлено ее *связывание* с конкретным физическим файлом. Процедуру связывания часто называют *открытием* физического файла. В открытие файла входит, в частности, поиск его на диске.

В процессе работы программа может завершить обработку одного физического файла и перейти к работе над другим аналогичным файлом. Для этого требуется “отключить” программу от первого файла или, как говорят, *закрыть* этот файл и открыть новый.

Методы доступа

Итак, мы выяснили, что работа с файлами заключается в выполнении операций над их записями, соответственно одним из существенных вопросов в этой работе является вопрос о способе доступа к записям. В принципе здесь возможны два подхода.

Работа в режиме так называемого *последовательного доступа* означает возможность выбора очередной записи только после обработки записи, физически расположенной перед ней. Здесь можно провести аналогию с воспроизведением очередной песни, записанной на магнитофонной кассете. Доступ к ней можно получить либо, проиграв предыдущую, либо, осуществив ее перемотку. Способ доступа, конечно же, диктуется алгоритмом решения конкретной задачи, многие из них требуют именно последовательной обработки данных. Кроме того, последовательный метод доступа обладает важным достоинством: он не требует указания адреса очередной записи и поиска ее на носителе.

Более универсальным методом доступа к записям файла является так называемый *прямой доступ*. Он означает возможность выбора в любой момент произвольной записи файла вне зависимости от ее местоположения и того, какая запись обрабатывалась до этого. Продолжая “музыкальную” аналогию, этот способ доступа к информации можно сравнить с выбором песни

на аудиодиске. Заметим, что при этом сохраняется возможность и последовательного прослушивания песен без указания местоположения (адреса) очередной песни.

Прямой доступ к записям является универсальным методом, на основе которого может быть запрограммирован любой алгоритм обработки файлов. Очевидно, реализация прямого доступа требует наличия в языке программирования средств “именования” записей и указания требуемой записи по ее “имени”. В языке Pascal, разумеется, реализованы оба рассмотренных нами метода доступа, при этом в качестве имен записей выступают номера.

Доступ на чтение и на запись

Базовые операции в работе с файлами – чтение и запись данных. Давайте подумаем вот над каким вопросом: возможно ли в рамках одной “сессии” работы с файлом (от открытия до закрытия) поочередное выполнение этих операций? Вопрос не так прост, как может показаться. Ответ на него не однозначен и, в целом, связан со способом организации работы с файлами и соглашениями, принятыми в конкретной системе программирования.

Поговорим для начала о модификации файла, то есть добавлении к нему новых записей и удалении ставших не нужными. Если удалять данные из файла не требуется, то и с добавлением никаких проблем не возникнет. Дописываем новые записи в конец файла, пока на диске есть место – все в порядке. А вот если данные требуется в процессе работы удалять?

Во-первых, надо понять, а что значит “удалить запись”? Или, другими словами, чем “удаленная” запись отличается от “нормальной”? Отличить записи по содержимому возможно только в частных случаях. Например, если записи хранят числа и из задачи известно, что допустимы лишь положительные значения, то в удаленные записи можно разместить, скажем, число –1. В общем случае, таких “выделенных” значений, которые можно использовать в качестве признаков, конечно же, нет. Выходов два: либо запоминать номера удаленных записей, что имеет смысл только при регулярной структуре файла (если записи переменной длины, номера смысла не имеют), либо в каждой записи размещать служебную информацию – признак удаленности.

Допустим, с удалением мы разобрались, теперь, во-вторых, надо решить, каким образом мы будем добавлять новые записи. Можно, как и ранее, дописывать их в конец, что естественным образом приведет к “разбуханию” файла и лишним затратам дискового пространства. Можно попробовать использовать освободившееся место в середине файла. Снова получаем две ситуации: если записи постоянной длины, все в порядке – одну удалили, на ее место в точности “войдет” другая, при переменной же длине может оказаться, что свободное место внутри файла есть, но нет ни одного достаточного по размеру непрерывного фрагмента. Потребуется перепаковка – сдвиг фрагментов в начало файла, а значит затраты времени.

Следствием рассмотренных сложностей является тот факт, что во многих языках программирования и, в том числе, в языке Pascal на файлы, допускающие модификацию во время работы, накладывается ограничение – такие файлы должны состоять из записей одинаковой длины.

Что касается файлов с записями переменной длины, то в связи с их нерегулярной структурой работа с ними, кроме частных случаев, возможна только в режиме последовательного доступа. По этой же причине “последовательные” файлы в рамках одной сессии могут использоваться либо только в режиме чтения, либо только в режиме записи.

Виды файлов в Object Pascal

В языке Pascal файлы разбиты на три категории по областям применения.

В особый класс выделены *текстовые файлы*. Это последовательные файлы с записями переменной длины, содержащие кроме текста специальные символы типа “переход на следующую строку”. Важно понимать, что текстовый файл не является файлом “строк”, то есть содержащим записи из констант типа `String`. Такой файл тоже можно создать, но он уже будет относиться к другому виду файлов. Для объявления текстовых файлов используется предопределенный идентификатор типа `Text`.

Типизированные файлы – это файлы прямого доступа с записями фиксированной длины. Каждая запись этого файла представляет собой константу одного общего для всего файла типа данных. Этот вид файлов можно рассматривать как основной и наиболее универсальный. Конкретный вид типизированного файла задается при объявлении типа, которое записывается следующим образом:

```
type
  <имя файлового типа> = file of <тип записи>;
```

например,

```
type
   FileOfInteger = file of Integer;
```

Третий вид файлов – *бестиповые файлы*. Бестиповые файлы предназначены для программирования, что называется, на физическом уровне, то есть близком к уровню машинного языка. Они применяются в случае, когда требуется эффективная по времени и памяти реализация процедур обмена, такая, что можно пожертвовать наглядностью программы и дополнительными усилиями по программированию обмена. Кроме того, такая организация файла применяется, когда производится обмен без обработки информации, например в процедуре копирования. Бестиповые файлы объявляются следующим образом:

```
type
  <имя файлового типа> = file;
```

например,

```
type
  FileToCopy = file;
```

Для работы с конкретным физическим файлом в программе требуется создать файл логический – объявить переменную файлового типа. Например,

```
var
  T: Text;
  FInt: FileOfInteger;
  FC: FileToCopy;
```

Операторы связывания логического и физического файлов

Средства работы с физическими файлами в языке Pascal отражают соответствующие возможности операционных систем типа DOS и семейства Microsoft Windows. Как известно, в этих системах физический файл идентифицируется путем доступа к нему и именем.

Например,

```
C:\Temp\Example1\Main.pas
A:\BOOK\CHAPTER7.TXT
```

Связывание имен логического и физического файла производится процедурой языка Object Pascal, обращение к которой имеет следующий вид:

```
AssignFile(<имя логического файла>, <имя физического файла>);
```

Здесь `имя логического файла` есть ранее объявленная файловая переменная, а `имя физического файла` может быть представлено строковым выражением, задающим в конечном итоге путь к файлу.

Заметим, что эта процедура – одна из немногих, чье наименование изменилось по сравнению с предыдущими версиями языка[1]. Из соображений обратной совместимости можно использовать форму процедуры с именем `Assign`, однако делать это не рекомендуется.

Примеры обращения к процедуре:

```
AssignFile(T, 'A:\BOOK\CHAPTER7.TXT');
AssignFile(FC, Path + FileName);
```

Во втором примере предполагается, что предварительно сделаны объявления вида:

```
var
  Path: String[67];
  FileName: String[12];
```

и этим переменным присвоены некоторые значения.

Заметим, что в общем случае задание имени физического файла с помощью переменной предпочтительнее, чем с помощью константы. Записывая имя-константу, мы жестко привязываем программу к конкретному имени физического файла. Если же имя задано переменной, то его можно, например, вводить в диалоге с пользователем программы.

Процедура `AssignFile` носит, в определенном смысле, декларативный характер и не выполняет всех действий по открытию файла, в частности, не производит поиск файла. Это делается отдельными процедурами `Reset` или `Rewrite`. Первая из упомянутых процедур открывает уже существующий файл, вторая – создает новый (пустой) файл. Существуют некоторые особенности использования этих процедур для открытия каждого из видов файлов (текстовых, типизированных и бестиповых). В этом разделе мы рассмотрим общие для всех видов файлов аспекты работы процедур открытия.

Обращения к процедурам открытия файла имеют вид:

```
Reset(<имя логического файла>);
```

[1] Вторая процедура – CloseFile. В Borland Pascal 7.0 она называлась Close. В Object Pascal рекомендуется использовать новый вариант – CloseFile.

```
Rewrite(<имя логического файла>);
```

например,

```
Reset(T);
Rewrite(FInt);
```

Процедура `Reset` осуществляет поиск файла с физическим именем, связанным в данный момент с файловой переменной, и подготавливает его к работе. Если файл с заданным именем не найден, то в стандартной ситуации выдается сообщение об ошибке. Чтобы избежать выдачи маловразумительного для пользователя-непрограммиста сообщения, обычно применяют прием, основанный на использовании директивы `{$I-}`, позволяющей отказаться от вставки в исполняемый модуль команд, контролирующих выполнение процедур обмена. При этом, как мы уже не раз отмечали, для получения информации о результате операции можно использовать функцию `IOResult`. Ниже приведен фрагмент программы, запрашивающий имя файла до тех пор, пока не будет введено имя существующего на диске набора данных.

```
{ ========================================================================= }
{ Пример 8.1 }
{ Открытие файла }
program OpenFile;
var
  T: Text;
  FileName: String[80];
begin
  repeat
    Write('Введите имя файла: ');
    Readln(FileName);
    {$I-}
    AssignFile(T, FileName);
    Reset(T);
    {$I+}
  until IOResult = 0;
  . . .
end.
{ ========================================================================= }
```

Процедура создания нового файла `Rewrite` таит в себе опасность другого сорта. Если на диске уже существует файл с именем вновь открываемого файла, то старый набор данных будет уничтожен без какого либо предупреждения. Таким образом, пользователь может случайно уничтожить полезную информацию, если он проявил невнимательность при выборе имени нового файла. Поэтому в общем случае при создании нового файла полезно проверить наличие файла с таким именем и предупредить об этом пользователя.

Рассмотрим пример программы, осуществляющей предварительную проверку наличия файла с именем, совпадающим с именем вновь создаваемого файла. Программа пытается открыть файл процедурой `Reset` и в случае успеха запрашивает у пользователя разрешение на удаление старого файла. Если пользователь не дает разрешения, программа запрашивает новое имя для создаваемого файла. В примере использована функция `UpCase(<символ>)`. Она преобразует буквы из строчных в заглавные. Если аргумент не является строчной буквой, то результат совпадает с аргументом. Смысл применения функции – освободить пользователя от необходимости переключать регистр при ответе на запрос об удалении файла.

```
{ ========================================================================= }
{ Пример 8.2 }
```

```
{ Проверка наличия файла с именем вновь создаваемого файла }
program OpenWithCheck;
var
  T: Text;
  FileName: String[80];
  Answer: Char; { Ответ на вопрос об удалении файла }
  Exists: Boolean; { Признак наличия файла }
begin
  repeat
    Writeln('Введите имя файла: ');
    Readln(FileName);
    {$I-}
    AssignFile(T, FileName);
    Reset(T);
    {$I+}
    Exists := IOResult = 0;
    if Exists then
    begin
      CloseFile(T); { Закрытие существующего файла }
      Write('Такой файл уже есть. Удалить? (Y/N) ');
      { Ввод кода клавиши, гарантирующий реакцию только на }
      { допустимый ответ: Y или y (Да), либо N или n (Нет) }
      repeat
        Readln(Answer);
      until (UpCase(Answer) = 'Y') or (UpCase(Answer) = 'N');
      Writeln(Answer);
    end;
  until (not Exists) or (UpCase(Answer) = 'Y');
  Rewrite(T);
  . . .
end.
{ ************************************************************************ }
```

В вышеприведенном примере мы использовали процедуру закрытия файла, обращение к которой имеет следующий общий вид:

```
CloseFile(<имя логического файла>);
```

Закрытие файла следует производить перед переключением файловой переменной с одного физического файла на другой или перед завершением работы программы.

Текстовый файл

Текстовый файл представляет собой последовательный файл с записями переменной длины. В данном случае записи – это строки текста. Разделяются записи между собой специальным признаком “конец строки”, формирующимся, например, при нажатии клавиши Enter.

При открытии файла определяется способ его обработки. Файл, открытый процедурой `Reset`, предназначен только для чтения его записей. В файл, созданный процедурой `Rewrite`, можно только записывать информацию.

Модификация уже существующего текстового файла возможна только путем добавления новых записей в конец файла. Для этого он должен быть открыт специальной процедурой

```
Append(<имя логического текстового файла>);
```

Мы уже практически знакомы с операциями обработки текстового файла. Дело в том, что ввод с клавиатуры – это не что иное, как ввод записей текстового файла, связанного с физическим файлом, "размещенным" на клавиатуре. Вывод же на дисплей – это вывод в текстовый физический файл, "размещенный" на дисплее. Эти стандартные файлы имеют предопределенные в библиотеке `System` (которая, напомним, всегда автоматически подключается к программе) имена: `Input` – для входного файла и `Output` – для выходного. Заметим, что имеется возможность связать эти логические файлы и с другими устройствами, например, назначив вывод из текстового файла `Output` на диск.

Общая форма записи операторов чтения и записи включает в себя упоминание имени логического файла:

```
Read([<имя логического текстового файла>,] <список ввода>);
Readln([<имя логического текстового файла>,] <список ввода>);
Write([<имя логического текстового файла>,] <список вывода>);
Writeln([<имя логического текстового файла>,] <список вывода>);
```

Если имя логического файла не указано, подразумевается `Input` для операторов чтения и `Output` для операторов записи.

Обратим еще раз внимание на важную особенность операторов обмена с текстовым файлом. Они обеспечивают автоматическое преобразование информации из текстового вида к типу переменных, стоящих в списке, при вводе и из внутреннего формата представления данных в текстовый вид при выводе.

Если, обрабатывая некоторый ранее созданный текстовый файл, мы не знаем заранее количество записей-строк, как же определить, когда файл закончится? Проблема состоит в том, что по самому файлу определить, сколько в нем строк невозможно – в языке Pascal операции для этого не предусмотрено. Все что нам программистам доступно – это функция

```
Eof(<имя логического файла>);
```

которая возвращает значение **`true`** при достижении конца файла.

Аналогичного сорта проблема может возникнуть и при чтении отдельной записи. Составляя программу, разработчик должен знать формат каждой строки обрабатываемого файла (например, сколько в ней чисел, в каком формате эти числа представлены), чтобы правильно использовать оператор чтения. Если отсутствует информация о количестве элементов строки, то можно воспользоваться функцией

```
Eoln(<имя логического текстового файла>);
```

которая принимает значение **`true`**, если встретился признак "конец строки".

Заметьте, что в соответствии с описанием функция `Eof` применима не только к тестовым, но и к файлам других видов, в отличие от функции `Eoln`.

Вариантами этих функций, полезными при чтении числовой информации из текстового файла, являются булевские функции

```
SeekEof(<имя логического текстового файла>);
```

и

```
SeekEoln('имя логического текстового файла');
```

Первая из них выдает сообщение о конце файла, если в последних строках нет символов, отличных от пробелов или символов табуляции. Вторая функция сообщает о конце строки, если в ней не осталось непрочитанных символов, отличных от пробелов и табуляций.

Рассмотрим пример использования этих языковых средств. Допустим, с помощью текстового редактора подготовлен текстовый файл, содержащий вещественные числа. По некоторым причинам числа размещены в “свободном формате”, то есть строки содержат разное число чисел, содержат пробелы в конце строк, пустые строки могут встретиться в любом месте файла. Пример такого текста приведен ниже:

```
-2.3 34.5 3 45 45.56


34 -4.67
36.90

23 4 -89 4.67

12.45 4 567

-56.7

4
```

Требуется отформатировать этот текст, поместив на каждой строке только по три числа и удалив все лишние пробелы и пустые строки, то есть получить текст вида:

```
  -2.30    34.50     3.00
  45.00    45.56    34.00
  -4.67    36.90    23.00
   4.00   -89.00     4.67
  12.45     4.00   567.00
 -56.70     4.00
```

Эту задачу решает следующая программа:

```
{ ========================================================================== }
{ Пример 8.3 }
{ Форматирование текстового файла }
Program Format;
var
  Old,       { исходный файл       }
  New: Text; { результирующий файл }
  Count: Word;
  X: Real;
begin
  AssignFile(Old, 'OLD.TXT');
  Reset(Old);
  AssignFile(New, 'NEW.TXT');
  Rewrite(New);
  Count := 0;
  while not SeekEof(Old) do    { пока в файле есть непустые строки }
  begin
    while not SeekEoln(Old) do { пока в строке есть числа          }
    begin
      Inc(Count);
      Read(Old, X);            { читаем одно число из строки       }
```

```
      Write(New, X:8:2, ' ');   { записываем это число              }
      if Count = 3 then         { если записали 3 числа             }
      begin                     { то                                }
        Count := 0;
        Writeln(New);           { переходим на новую строку         }
      end;
    end;
  end;
  CloseFile(Old);
  CloseFile(New);
end.
{ ========================================================================== }
```

Типизированный файл

Типизированные файлы являются файлами прямого доступа с фиксированной длиной записи, определяемой типом записи при объявлении файла. Тип записи – это один из рассмотренных нами типов языка Pascal, за исключением файлового типа. Другими словами нельзя создать “файл файлов”, однако можно создать, например, такой экзотический файл как “файл процедур”, в котором будут храниться значения переменных процедурного типа. Примеры объявления конкретных файловых типов:

```
type
  FileOfReal = file of Real;

  Matrix = array[1..10, 1..20] of Word;
  FileOfMatrix = file of Matrix;

  Person = record
    Name: String;
    Age: Byte;
  end;
  PersonList = file of Person;
```

Типизированный файл может обрабатываться как в режиме последовательного так и в режиме прямого доступа к записям. Во втором случае операторы обмена должны знать адрес требуемой записи. Как уже отмечалось, адресация записей производится по их порядковым номерам, которые присваиваются записям в процессе создания файла. Условно говоря, с файлом связан “курсор ” – указатель позиции в файле, который показывает на текущую обрабатываемую запись. После открытия существующего файла курсор указывает на первую запись. Выполнение последовательных операторов обмена производит перемещение курсора на следующую по порядку запись. Выполнение операций обмена с прямым доступом переносит курсор на запись с номером, указанным в этом операторе.

Обмен с текущей (на которую указывает курсор) записью производится операторами:

```
Read(<имя логического файла>, <список ввода>);
Write(<имя логического файла>, <список вывода>);
```

Списки ввода и вывода должны состоять по крайней мере из одной переменной, принадлежащей к тому же типу, что и все записи файла. После выполнения этих операторов курсор передвигается на следующую запись.

Для осуществления прямого доступа используется процедура установки курсора на запись с заданным номером:

```
Seek(<имя логического файла>, <номер записи>);
```

Второй параметр процедуры представляет собой выражение типа `Longint`.

Чтобы определить номер записи, на которую в данный момент указывает курсор, используется функция

```
FilePos(<имя логического файла>);
```

Типизированные файлы допускают их модификацию путем замены любых записей, добавления новых с номерами, большими, чем максимальный текущий номер записи, а также удаления записей. При этом неважно, каким образом был открыт файл – был ли он вновь создан процедурой `Rewrite`, или был открыт существовавший ранее файл процедурой `Reset`. В любом случае после открытия файла можно как читать записи, так и записывать в него информацию.

Рассмотрим более подробно приемы модификации типизированного файла. Поскольку все записи файла принадлежат одному типу, то замена записи производится просто: следует установить курсор на заменяемую запись и осуществить вывод в нее новой информации.

Вставка новых записей в файл возможна только путем добавления их в конец файла. Другими словами, нельзя вставить новую запись между двумя уже существующими записями (“раздвинуть” записи). Чтобы добавить в конец файла новую запись, следует в процедуре `Seek` указать номер последней записи файла. Это число можно узнать, воспользовавшись функцией

```
FileSize(<имя логического файла>);
```

которая возвращает количество записей в типизированном файле (заметим, что она не применима к текстовым файлам). Таким образом, пара процедур:

```
Seek(F, FileSize(F));
Write(F, Info);
```

добавляет в конец файла `F` новую запись с информацией из переменной `Info`.

Интересно, что номер добавляемой записи не обязательно должен быть на единицу большим общего числа записей в файле. Например, если файл имеет 50 записей, мы можем поместить в него запись с номером 60. При этом будут созданы и промежуточные 9 записей с номерами от 51 до 59, которые могут быть заполнены полезной информацией позже.

Удаление записей также возможно только в конце файла, то есть методом “отсечения” его “хвостовой” части. Для этого предназначена процедура

```
Truncate(<имя логического файла>);
```

которая отсекает все записи файла, начиная с той, на которую указывает в данный момент курсор.

Следует обратить внимание на одну важную особенность обмена с типизированными файлами. Обмен информацией между внешней и оперативной памятью всегда происходит без преобразования данных. При записи информация заносится на диск в своем внутреннем представлении. Таким образом, содержимое типизированного файла в общем случае нельзя посмотреть с помощью текстового редактора, точнее посмотреть, конечно, можно, а вот понять вряд ли. Отсутствие преобразования информации к текстовому виду при записи в типизированный файл существенно ускоряет процесс обмена.

В заключение обзора средств работы с типизированными файлами отметим, что функция определения конца файла `Eof` и процедура закрытия файла `CloseFile` также применяются для их обработки.

Рассмотрим пример работы с типизированным файлом.

```
{ ========================================================================== }
{ Пример 8.4 }
{ Создание и модификация типизированного файла }
Program TypedFile;
type
  Person = record { строка ведомости }
    Name: String[20];
    Salary: Word;
  end;
  PersonFile = file of Person;
var
  P: Person;
  F: PersonFile;
  S: String[80]; { буфер для вводимой текстовой строки   }
  StarPos,       { номер позиции строки, где находится * }
  Code: Integer; { код завершения процедуры Val           }
                 { (не анализируется в данной программе) }

{ Преобразование входной текстовой строки к типу Person }
procedure Perform;
begin
  StarPos := Pos('*', S);
  P.Name := Copy(S, 1, StarPos - 1);
  Val(Copy(S, StarPos + 1, Length(S) - StarPos), P.Salary, Code);
end; { Perform }

{ Заполнение типизированного файла }
procedure Create;
var
  T: Text;
begin
  AssignFile(T, 'LIST.TXT');
  Reset(T);
  while not SeekEof(T) do
  begin
    Readln(T, S);
    Perform;
    Write(F, P);
  end;
  CloseFile(T);
end; { Create }

{ Вывод ведомости на экран с печатью итоговой суммы }
procedure Display;
var
  Sum: Word;
begin
  Reset(F);
  Sum := 0;
  with P do
```

```
  begin
    while not Eof(F) do
    begin
      Read(F, P);
      Write(Name, '  ');
      Writeln(Salary);
      Sum := Sum + Salary;
    end;
  end;
  Writeln('   ', 'Итого: ', Sum);
end; { Display }

{ Замена или добавление строки }
procedure ChangeOrAdd;
var
  N, i: Integer;
begin
  Writeln('Введите новую строку');
  Readln(S);
  Val(Copy(S, 1, Pos('.', S) - 1), N, Code);
  if N > FileSize(F) then
  begin
    P.Salary := 0;
    for i := FileSize(F) + 1 to N - 1 do
    begin
      Str(i, P.Name);
      P.Name := P.Name + '.';
      Write(F, P);
    end;
  end;
  Perform;
  Seek(F, N - 1);
  Write(F, P);
end; { ChangeOrAdd }

begin
  AssignFile(F, 'LIST.PSL');
  Rewrite(F);
  Create;
  Display;
  Readln;
  ChangeOrAdd;
  Display;
  Readln;
end.
{ ========================================================================== }
```

Приведенная программа включает процедуру создания типизированного файла (информация читается из заранее подготовленного текстового файла), процедуру вывода на экран содержимого файла и процедуру замены или добавления новой записи в файл. Первые две процедуры работают с файлом в режиме последовательного доступа, третья – в режиме прямого доступа.

Информация, заносимая в файл, представляет собой “ведомость” на получение заработной платы. Первоначально ведомость подготавливается с помощью текстового редактора в следующем виде:

```
  1.Иванов И.И.*5000
```

```
2.Петров П.П.*6500
3.Сидоров С.С.*7000
4.Королев Н.Н.*5500
```

Каждая строка начинается с номера, отделяемого от остальной части точкой. Номер строки используется при прямом доступе к записям. Фамилия сотрудника отделяется от его заработной платы с помощью символа “*”.

При выводе ведомости на экран печатается итоговая сумма по заработной плате. Для этого требуется представление заработной платы в числовом виде.

Замена или добавление новой строки ведомости производится по следующим правилам. Запрашивается новая строка, которая должна быть введена в вышеуказанном формате. Если номер строки не превышает числа строк в ведомости, то будет заменена соответствующая строка. Если номер строки больше числа строк в ведомости, то эта строка будет добавлена в файл. Причем допускается вводить строки с номерами, более чем на единицу превышающими число строк в ведомости. При этом будут созданы все промежуточные строки, в которые будут помещены только их номера. Например, добавление строки

```
7.Кузнецов А.А.*8000
```

к вышеуказанному файлу приведет к появлению на экране (в случае вызова процедуры печати) текста:

```
1.Иванов И.И.  5000
2.Петров П.П.  6500
3.Сидоров С.С.  7000
4.Королев Н.Н.  5500
5.  0
6.  0
7.Кузнецов А.А.  8000
  Итого: 32000
```

В программе с целью сокращения объема опущены всевозможные проверки, обеспечивающие надежность работы. Например, не проверяется наличие текстового файла, не контролируется синтаксис вводимых строк и тому подобное.

Бестиповые файлы

Текстовые файлы – наиболее близкий и понятный человеку формат представления информации, файлы типизированные отражают тот факт, что человек любую информацию, с которой приходится работать, стремится упорядочить, структурировать. Именно поэтому два этих вида наиболее распространены. Однако существует ряд задач, в которых внутреннее устройство файла либо не имеет значения, например, упоминавшаяся уже задача копирования файла, либо интерпретация содержимого может меняться в зависимости от потребности, скажем, в одной ситуации запись обрабатывается в программе как строка символов, а в другой в ней требуется выделять числовые поля. Во всех этих случаях использование текстового или типизированного вида файлов является не слишком удобным. На помощь приходит бестиповый файл.

Бестиповые файлы, объявляемые как

```
type
  <имя файлового типа> = file;
```

являются файлами с прямым доступом и записями постоянной длины.

Открывая бестиповый файл, программист может указать длину записи файла в байтах. Это относится и к ранее созданным файлам, причем не требуется, чтобы указываемая длина совпадала с той длиной записи, которая указывалась (явно или неявно) при создании открываемого файла. Если длина записи не указана при открытии, то она по умолчанию принимается равной 128 байт. Задание длины записи производится в уже известных нам операторах открытия файла `Reset` и `Rewrite`, где она указывается вторым параметром. Например, процедура

```
Reset(F, 256);
```

открывает существующий файл `F`, и задает длину его записи в 256 байт.

Процедура

```
Rewrite(F, 1);
```

создает новый файл с длиной записи, равной 1 байту.

Для ускорения обмена передаваемые записи могут блокироваться в одну физическую запись-блок. На необходимость блокирования записей указывают параметры процедур обмена, которые для бестиповых файлов имеют особую форму, не совпадающую с формой записи операторов обмена, рассмотренных нами ранее.

Оператор чтения блока с диска в оперативную память имеет вид:

```
BlockRead(<имя логического файла>, <буфер>, <число записей>
  [, <число прочитанных записей>]);
```

Размер читаемого блока определяется как произведение третьего параметра на размер записи. Второй параметр задает область памяти, в которую помещается блок. Это должна быть переменная, число байт в которой не меньше чем в блоке.

Если количество байт информации файла не делится нацело на длину блока, то при чтении последнего блока признак “конец файла” может встретиться раньше, чем будет прочитан блок. Если последний параметр не был задан, возникнет ошибка ввода-вывода, которая приведет к аварийному завершению программы (конечно, если не использовалась директива `{$I-}` и соответствующая обработка аварийной ситуации). Если же последний параметр задан, то в нем будет содержаться число прочитанных записей блока. Однако здесь надо учитывать одну тонкость. В этот параметр передается число полностью прочитанных записей. Если же запись состояла более чем из одного байта, то могло случиться так, что конец файла встретился до того, как запись была полностью прочитана. В этом случае эта “недочитанная” запись не учитывается.

Процедура записи блока на диск имеет вид:

```
BlockWrite(<имя логического файла>, <буфер>, <число записей>
  [, <число прочитанных записей>]);
```

Здесь первые три параметра имеют тот же смысл, что и в предыдущей процедуре. В последний параметр возвращается число реально переданных записей. В подавляющем большинстве случаев оно равно числу записей в блоке, то есть третьему параметру. Однако если при записи на диск возникла ситуация “диск полон”, то в этот параметр вернется число записей блока, которые уместились на диске. Также как и при чтении, если запись не удалось записать на диск целиком, она не войдет в общее количество.

Если последний параметр процедуры опущен, то при переполнении диска произойдет аварийное завершение программы.

Учитывая выше рассмотренные особенности операторов обмена, можно рекомендовать объявлять файлы с длиной записи, равной одному байту. В этом случае не возникнут неприятности, связанные с “недоучтенными” записями.

Все ранее рассмотренные процедуры (кроме `Read` и `Write`) для типизированных файлов так же применимы и к бестиповым файлам.

В качестве примера использования бестиповых файлов рассмотрим процедуру копирования файла.

```
{ ========================================================================== }
{ Копирование файла }
Program CopyFile;
var
  FromF, ToF: file; { исходный и результирующий файл  }
  NumRead,          { число прочитанных записей блока }
  NumWritten        { число записанных записей блока  }
   : Integer;
  NumCopied         { общее число скопированных записей }
   :Longint;
  Answer: Char;
  buf: array[1..2048] of Char; { буфер для размещения блока }

begin
  {$I-}
  AssignFile(FromF, ParamStr(1));
  Reset(FromF, 1); { длина записи - 1 байт }
  {$I+}
  if IOResult <> 0 then
  begin
    Writeln('Исходный файл ',ParamStr(1), ' не найден');
    Halt;
  end;

  {$I-}
  AssignFile(ToF, ParamStr(2));
  { Проверка на наличие файла с заданным именем }
  Reset(ToF);
  {$I+}
  if IOResult = 0 then
  begin
    CloseFile(ToF);
    Write('Файл ', ParamStr(2), ' уже есть. Переписать? (Y/N) ');
    repeat
      Readln(Answer);
    until (UpCase(Answer) = 'Y') or (UpCase(Answer) = 'N');
    if UpCase(Answer) = 'N' then
      Halt;
  end;

  Rewrite(ToF, 1); { длина записи - 1 байт }
  Writeln('Копирую ', FileSize(FromF), ' байт...');
  NumCopied := 0;
  repeat
    BlockRead(FromF, buf, SizeOf(buf), NumRead);
    BlockWrite(ToF, buf, NumRead, NumWritten);
```

```
    NumCopied := NumCopied + NumWritten;
  until (NumRead = 0) or (NumWritten < NumRead);

  if NumWritten < NumRead then
    Writeln('Диск полон. Скопировано ', NumCopied, ' байт')
  else
    Writeln('Копирование завершено');

  CloseFile(ToF);
  CloseFile(FromF);
end.
{ ======================================================================== }
```

Сделаем пояснения по примеру. Чтобы выполнить копирование файла с помощью нашей программы требуется запустить ее с параметрами командной строки, например, так

```
copyfile c:\user\data.txt a:data.txt
```

В связи с таким способом вызова программы возникает проблема передачи параметров, записанных в командной строке. В языке Pascal имеется предназначенная для этих целей функция

```
ParamStr(<номер аргумента>);
```

аргументом которой является порядковый номер параметра из командной строки, а результатом – сам параметр в виде строковой (тип `String`) константы. Так, результатом обращения `ParamStr(1)` при условии вызова программы копирования выше приведенной командной строкой будет имя исходного файла `c:\user\data.txt` . Заметим, что вызов функции с номером параметра `0` даст первый элемент командной строки, то есть имя программы (в нашем случае `copyfile`).

Некоторые возможности управления файловой системой

Язык Object Pascal включает набор процедур, отражающих возможности операционной системы по управлению файлами. В их число входят процедуры поиска, удаления файлов, смены текущего каталога и другие. Все эти процедуры позволяют разрабатывать собственные информационно-поисковые системы, базы данных и тому подобные программные комплексы обработки и хранения больших массивов информации.

В языке имеется более десятка соответствующих процедур и функций, часть из которых размещается в библиотеке `System`, часть – в библиотеке `SysUtils`. Кроме того, в библиотеке `SysUtils` определен ряд типов данных и переменных, облегчающих составление программ управления файлами. Мы не будем рассматривать все существующие средства, а ограничимся некоторым набором подпрограмм, необходимым для реализации примера.

В качестве примера рассмотрим программу удаления всех файлов с расширением .EXE из каталога, заданного при вызове программы в командной строке. Причем, если каталог не указан, удаление будем производить в текущем каталоге. Программа должна сообщать имена удаляемых файлов и их размеры в байтах. В завершение программа должна напечатать число свободных байт на логическом диске, где расположен каталог.

Рассмотрим стандартные средства, требуемые для реализации этой программы.

Процедура

```
ChDir(<каталог>);
```

устанавливает в качестве текущего каталог, заданный параметром. Если такого каталога нет, возникает ошибка, которая может быть обработана с использованием функции `IOResult`.

Процедура

```
Erase(<имя логического файла>);
```

удаляет физический файл, связанный с логическим файлом, заданным параметром. В момент удаления файл должен быть закрыт. В нашем примере мы просто не будем открывать удаляемый файл, лишь связав его с файловой переменной оператором `AssignFile`.

Рассмотренные процедуры размещаются в библиотеке `System`. Для использования следующих подпрограмм необходимо подсоединить библиотеку `SysUtils`.

Пара процедур `FindFirst` и `FindNext` обеспечивают поиск файлов. Прежде всего рассмотрим предопределенный в библиотеке тип данных `TSearchRec`, переменные которого используются в качестве параметров этих процедур. Это запись, объявленная как

```
type
  SearchRec = record
    Time: Integer;
    Size: Integer;
    Attr: Integer;
    Name: TFileName;
    ...
end;
```

Поля этой записи имеют следующий смысл:

Поле	Содержание
Time	Дата и время создания файла (упакованные)
Size	Размер файла в байтах
Attr	Атрибуты файла
Name	Имя файла

В примере для нас будут важны атрибуты `Name` и `Size`, смысл которых очевиден.

Процедура

```
FindFirst(<маска файла>, <атрибут>, <данные о файле>);
```

производит поиск первого файла с заданными характеристиками в каталоге, указанном в первом параметре. Вообще, первый параметр – это специальная “маска” или “шаблон” для поиска. В нем задается вид искомых файлов с использованием спецсимволов '`*`' и '`?`',. Например, маска “`*.PAS`” говорит, что требуется найти первый файл с расширением `.PAS` в текущем каталоге. Второй параметр задает атрибут файла в смысле классификации, принятой в операционной системе. Файлы могут быть системными, предназначенными только для чтения, скрытыми и тому подобное. Третий параметр представляет собой переменную типа `TSearchRec`, в которую заносится информация о найденном файле.

Процедура

```
FindNext(<данные о файле>);
```

находит очередной файл с характеристиками, заданными в предыдущей процедуре. Информация об этом файле заносится в единственный параметр процедуры, который должен быть переменной типа `TSearchRec`.

По завершении процедуры `FindFirst` и `FindNext` возвращают код, характеризующий результат выполнения операции. Код ноль означает, что операция завершилась успешно.

В заключение рассмотрим использованную в примере функцию:

```
DiskFree();
```

которая выдает количество свободных байт (тип результата `Int64`) на логическом диске. Диск задается параметром – целой константой. Константа `0` означает текущий диск, `1` – диск A, 2 – диск B и так далее.

Ну а теперь собственно текст программы.

```
{ ========================================================================== }
{ Удаление всех файлов с расширением .EXE из заданного каталога }
Program DelEXE;
uses SysUtils;
var
  Dir: String;
  DirInfo: TSearchRec;
  Code: Integer;
  f: file;
begin
  {$I-}
  { Установка нового текущего каталога }
  Dir := ParamStr(1);
  if Dir <> '' then
  begin
    ChDir(Dir);
    if IOResult <> 0 then
    begin
      Writeln('Каталог не найден');
      Halt;
    end;
  end;
  { Поиск и удаление файлов }

  Code := FindFirst('*.EXE', faAnyFile, DirInfo);
  while Code = 0 do
  begin
    WriteLn('Удален файл ', DirInfo.Name, ' ', DirInfo.Size, ' байт');
    AssignFile(f, DirInfo.Name);
    Erase(f);
    Code := FindNext(DirInfo);
  end;

  Writeln('Свободных ', DiskFree(0),' байт');
end.
{ ========================================================================== }
```

Выводы

В этой главе мы рассмотрели классические приемы работы с внешней памятью средствами языка Object Pascal, выяснили, что в нем на логическом уровне поддерживаются три вида файлов – типизированные, текстовые и бестиповые, что полностью покрывает все возможные потребности программистов по обмену данными между программой и внешней памятью. Заметим, что современные средства программирования, в частности, Borland Delphi, содержат и более развитые механизмы осуществления файловых операций. Познакомившись в главе 10 с объектно-ориентированным программированием, желающие могут изучить класс TFileStream из модуля Classes, предоставляющий еще более удобные средства для чтения/записи данных.

9. Динамическое управление памятью

Все жалуются на свою память, но никто не жалуется на свой разум.
Франсуа де Ларошфуко

Умелая *работа с памятью* – один из наиболее важных элементов мастерства программиста. От того, как будет написан соответствующий код, обычно зависит не только производительность программы, но и ее работоспособность. Ошибки, допущенные при работе с памятью, обходятся очень дорого на этапе отладки программы, требуя привлечения больших временных ресурсов для их обнаружения.

Наш опыт подсказывает, что для грамотной работы с памятью, прежде всего, необходимо понимать, как эта работа устроена изнутри, какие действия происходят, когда компилятор преобразует наши скупые синтаксические конструкции в *машинный код*. Для достижения этого понимания нужно немного потрудиться, чем мы сейчас и займемся. В начале мы рассмотрим общие принципы, не зависящие от языка программирования и реализации компилятора, и лишь потом конкретную реализацию изученных механизмов в языке Object Pascal и системе Borland Delphi.

Приступим!

Проблемы работы с памятью в многозадачной операционной системе

Для начала, слегка коснемся некоторых важных системных вопросов. Конечно, заявленная в заголовке раздела тема значительно лучше подходит для курса “Операционные системы” и, вообще говоря, заслуживает длительного разговора [16, 17]. Мы постараемся ограничиться в данной книге минимумом, необходимым для понимания материала.

Итак, будем считать, что мы работаем в *многозадачной операционной системе*, в частности, Windows.

> Принцип многозадачности подразумевает возможность одновременной работы нескольких программ. Вспомним, как происходит работа в однозадачной системе типа MS DOS. Вот Вы запустили какую-то программу, например, игрушку и не успели погрузиться в таинственный мир лабиринта и нацелить любимую базуку на вражеский корабль, как к Вам подскочил приятель и сообщил “радостную” новость: необходимо срочно отформатировать дискету. Что делать? Увы, Вы вынуждены записываться, выходить из программы, форматировать дискету, запускать программу снова,

загружаться... Неудобно, не правда ли? А все потому, что в однозадачной операционной системе не может выполняться более одной программы единовременно.

Попробуем задуматься над вопросом, как работает многозадачность? Компьютер, на котором исполняется множество программ, один, процессор в нем (по крайней мере на момент написания этой книги) чаще всего тоже, и память, и видеоадаптер и т.д. Не вдаваясь в подробности, можно заключить, что несколько одновременно запущенных программ должны каким-то образом делить между собой системные ресурсы, в частности процессорное время и оперативную память. Подумаем, какие проблемы могут возникнуть при совместной работе программ с памятью?

Прежде всего, кто-то должен следить за тем, чтобы программе необходимая ей память была своевременно предоставлена. Затем необходимо решение проблемы несанкционированного доступа одной программы к памяти, занятой другой. Далее нужно решать вопросы, связанные с надежностью, производительностью и, наконец, просто возможностью нехватки оперативной памяти, вытекающей из ее конечного объема. Все это берет на себя операционная система, фактически устраняя программиста от работы с аппаратурой напрямую. Так, все попытки запросить или освободить память, равно как и операции доступа проходят под чутким контролем операционной системы. Более того, функции по работе с памятью (выделение, удаление и другие), предоставляемые языками программирования и в частности языком Object Pascal (соответствующие возможности будут рассмотрены далее в этой главе), на самом деле реализованы на функциях операционной системы из состава так называемого Windows API – Application Programming Interface.

Что касается конечного объема оперативной памяти, установленной в компьютере, частично эта проблема решается с помощью механизма операционной системы под названием *виртуальная память*. Коротко этот механизм позволяет интерпретировать свободное место на жестком диске как продолжение оперативной памяти, а его “физическим” проявлением является создание на диске *файла подкачки*. Конечно, эта виртуальная память работает существенно медленнее оперативной (жесткий диск, как известно, устройство с элементами механики), что заставляет операционную систему применять различные оптимизационные алгоритмы, чтобы повысить производительность работы программ. Не вдаваясь в детали этих алгоритмов, укажем, что они основаны на страничной организации оперативной памяти и стратегиях замещения страниц.

Все случаи разделения ресурсов между несколькими потребителями порождают проблему одновременного доступа потребителей к одному ресурсу. В случае с памятью Windows решает эту проблему с помощью механизма *виртуального адресного пространства* – адреса оперативной памяти, с которыми работают одновременно запущенные программы, на самом деле являются логическими, трансляцией их в реальные занимается операционная система, исключая в результате конфликты использования программами одних и тех же участков памяти.

Адресное пространство программы

Понимание внутреннего устройства адресного пространства программы – важный шаг в изучении данной главы. Поговорим об этом подробнее.

Пусть у нас есть некоторая программа на Object Pascal. В силу определений стандарта языка она выглядит примерно так:

```
Program <Имя программы>;
uses <Список подключаемых модулей>;
<Блок объявлений глобальных переменных, констант и типов данных
```

```
    (var, const, type)>
  <Блок подпрограмм (procedure, function)>
  begin
    <Тело программы>
  end.
```

Рассмотрим пример программы, которая реализует функцию, вычисляющую среднее арифметическое двух чисел.

```
{ ========================================================================== }
{ Пример 9.1 }
{ Вычисление среднего арифметического двух чисел }
Program Simple;

{$APPTYPE CONSOLE}

var
  a, b: Integer;
  c: Real;

function Average(first, second: Integer): Real;
var
  res: Real;
begin
  res := (first + second) / 2;
  Result := res;
end;

begin
  Write('Введите числа: ');
  ReadLn(a, b);
  c := Average(a, b);
  WriteLn('Average(a, b) = ', c);
  ReadLn;
end.
{ ========================================================================== }
```

Попробуем разобраться, что из себя представляет эта программа, будучи запущенной на исполнение.

Не вдаваясь в подробности, связанные с устройством и хранением служебной информации и еще в некоторые системные вещи, ситуация выглядит так:

1. При запуске программы ей выделяется некоторый *диапазон адресов*. В 32-разрядных системах Microsoft Windows адрес представлял собой целое неотрицательное число, содержащееся в 32 двоичных разрядах. Соответственно, мы имеем 2^{32} разных адресов, что позволяет адресовать 4 Гигабайт памяти. Надо понимать, что реальный объем ОЗУ меньше 4-х Гигабайт, и, начиная с некоторого момента, адресуемые ячейки памяти на самом деле будут находиться в так называемой виртуальной памяти, т.е. в файле подкачки на жестком диске. Кроме того, существенная часть этих 4 Гигабайт занята служебной

информацией и не может быть использована нами. Тем не менее, того, что остается обычно более чем достаточно. Если же нет? Это тема отдельной книги[1].

2. Выделенный программе диапазон адресов – адреса в диапазоне $0 – 2^{32}-1$. Такой диапазон имеет каждая программа. Как мы уже говорили, адреса, которыми оперируют программы, являются виртуальными, а Windows производит их трансляцию в реальные, обеспечивая то, что адресные пространства разных программ не пересекаются. Более того, тем самым обеспечивается защита данных и кода, так как одна программа не может добраться до адресного пространства другой программы. Так, для рассмотренного примера *Сегмент кода* содержит одну подпрограмму и головную программу.

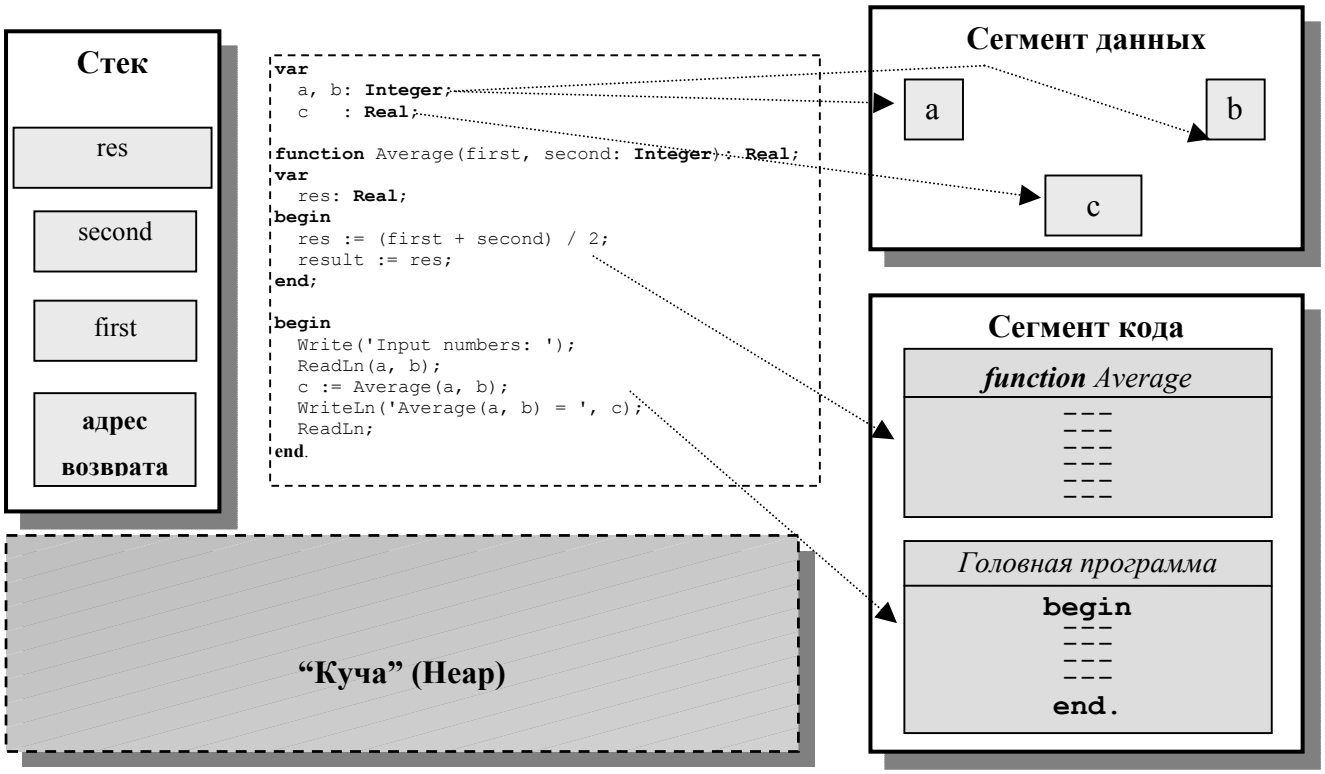


Рис. 9.1. Адресное пространство программы

3. При запуске программы в адресном пространстве выделяются несколько областей. Во-первых, это *Сегмент кода* – область памяти, в которой располагаются машинные инструкции, которые, собственно, и составляют программу. Можно выделить инструкции, составляющие головную программу, и инструкции, составляющие тела подпрограмм. Во-вторых, это *Сегмент данных* – область памяти, в которой располагаются глобальные переменные программы. Так, для рассмотренного примера это переменные a, b и c. В-третьих, это *Сегмент стека*. *Стек* – специальная структура, работающая по следующему принципу: пришедший первым элемент уходит последним (представьте себе, что вы складываете в стакан монеты одну за другой, а потом начинаете их доставать). В стеке размещаются локальные переменные подпрограмм, а также вся информация, которая требуется для их корректной работы. В частности, это, например, адрес возврата. Заметим, что в отличие от Сегмента кода и Сегмента данных, информация в которых

[1] Интересующимся прежде всего рекомендуем прочитать книгу Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows/Пер, англ - 4-е изд. - СПб; Питер; М.: Издательско-торговый дом "Русская Редакция", 2001. - 752 с.

существует с момента старта программы до ее окончания, данные в стеке находятся лишь в момент работы подпрограммы. После завершения работы подпрограммы вся связанная с ней информация из стека удаляется. Для рассмотренного примера в момент работы подпрограммы `Average` стек будет содержать адрес возврата и локальные переменные `first`, `second` и `res`. По окончании работы функции будут удалены из стека локальные переменные и выбран адрес возврата для перехода по нему в точку, из которой был осуществлен запуск функции.

4. Заметим на картинке таинственную *Кучу*. *Куча* (от английского *Heap*) – специальная область памяти, которая будет изучена в следующем разделе.

Динамическое управление памятью в языке Object Pascal

Рассмотрев устройство адресного пространства программы, самое время перейти к проблеме его практического использования. Как мы уже знаем, память для глобальных и локальных переменных, а также параметров подпрограмм выделяется компилятором автоматически. Более того, работа с адресами памяти, по которым размещаются переменные, скрыта от нас за их именами. А что если мы хотим поработать с адресами самостоятельно? Для этого существует специальная конструкция языка программирования Object Pascal – *указатель*.

Указатели бывают двух видов – *типизированные* и *бестиповые*. Начнем изучение с наиболее часто используемых типизированных указателей.

Работа с адресами. Типизированные указатели

Типизированный указатель – адрес области оперативной памяти, содержимое которой интерпретируется в соответствии с заявленным при объявлении указателя типом данных. Синтаксис объявления типа данных – указателя – выглядит так.

```
type
  <Имя типа данных-указателя> = ^<Тип данных>;
```

Так, например, в следующем примере мы объявляем новый тип данных (указатель) `PInteger`. Переменные типа `PInteger` будут хранить адреса ячеек оперативной памяти, причем содержимое этих ячеек будет интерпретировано как значение типа `Integer`.

```
type
  PInteger = ^Integer;
```

В дальнейшем мы можем использовать объявленный тип данных так:

```
var
  p: PInteger;
```

Заметим, что можно объявить указатель не только на стандартный тип данных, но и на тип, созданный программистом, как в следующем примере:

```
type
  TPerson = record
    FIO:       String[50];
```

```
    BirthdayY: Word;
    BirthdayM: Byte;
    BirthdayD: Byte;
    Phone:     String[15];
  end;
  PPerson = ^Person;
```

Кроме того допустим сокращенный синтаксис объявления указателя, подразумевающий объявление переменных, минуя создание типа.

```
var
  p1: ^Integer;
```

Охарактеризуем указатели с точки зрения определения типа данных. Множество значений типа указатель напрямую зависит от размера виртуального адресного пространства, в котором работает программа. В 32-разрядных операционных системах этот размер, как мы уже обсуждали, равен 4 гигабайтам. Поскольку минимально адресуемой ячейкой памяти является один байт, мы получаем 2^{32} адресов, то есть множество значений типа указатель составляют целые числа от 0 до 2^{32}-1. Заметим, что в 64-разрядных операционных системах, работающих на соответственно 64-разрядных процессорах, и указатели имеют диапазон значений от 0 до 2^{64}-1. Размер памяти под переменные типа указатель в точности определяется диапазоном значений и составляет 4 байта в 32-разрядном случае и 8 байт в 64-разрядном.

Разговор об операциях, применимых к указателям начнем с инициализации. Наиболее часто используемый способ установки начального значения указателя состоит во взятии адреса существующей переменной (или подпрограммы). Для этого можно применять практически эквивалентные: операцию взятия адреса `@` и функцию `Addr` библиотеки `System`, автоматически подключаемой к программе. Рассмотрим первый вариант, связанный с использованием операции взятия адреса (о втором варианте ниже при обсуждении бестиповых указателей).

```
{ ========================================================================== }
{ Пример 9.2 }
{ Операция взятия адреса }
Program Pointers;

{$APPTYPE CONSOLE}

type
  PInteger = ^Integer;
var
  p1, p2, p3: PInteger;
  i1, i2, i3: Integer;
begin
  i1 := 1;
  i2 := 2;
  i3 := 3;
  p1 := @i1;
  p3 := @i3;
  p2 := p1;
end.
{ ========================================================================== }
```

Не правда ли, все довольно просто? Тем не менее, опыт показывает, что работа с указателями относится к числу весьма тяжело усваиваемых тем, поэтому прокомментируем приведенный код:

5. В начале мы присваиваем переменным `i1`, `i2` и `i3` значения 1, 2 и 3 соответственно.

6. Далее сохраняем в переменной `p1` адрес переменной `i1`.
7. После этого сохраняем в переменной `p3` адрес переменной `i3`.
8. И, наконец, в переменную `p2` записываем тот же адрес, который хранится в переменной `p1`, т.е. адрес `i1`.
9. В результате мы настроили указатели следующим образом: `p1` и `p2` указывают на `i1`, а `p3` указывает на `i3`.

Итак, инициализировать указатели мы научились. Неплохо было бы теперь иметь возможность добраться до значения, лежащего в ячейках памяти, на которые он указывает. Конечно же в языке Object Pascal такая операция имеется. Эта операция в некотором смысле осуществляет действия, обратные операции взятия адреса, и называется *операцией разыменования*.

Операция разыменования для типизированного указателя p записывается как `p^`. Запись `p^` означает: “в оперативной памяти рассмотреть значения в ячейках, начиная с адреса p, и интерпретировать в соответствии с типом указателя”.

Усовершенствуем предыдущий пример:

```
{ ========================================================================== }
{ Пример 9.3 }
{ Операция разыменования }
Program Pointers;

{$APPTYPE CONSOLE}

type
  PInteger = ^Integer;
var
  p1, p2, p3: PInteger;
  i1, i2, i3: Integer;
begin
  i1 := 1;
  i2 := 2;
  i3 := 3;
  p1 := @i1;
  p3 := @i3;
  p2 := p1;
  p2^ := 5;
  p3^ := p1^ + p2^;
  WriteLn(i1, ' ', i2, ' ', i3);
  ReadLn;
end.
{ ========================================================================== }
```

Приведем комментарии к добавленным строкам кода.

1. По адресу, который хранится в `p2`, записываем число 5 (т.е. теперь `i1 = 5`).
2. По адресу, который хранится в `p3`, записываем сумму значений, хранящихся по адресам `p1` и `p2` (т.е. теперь `i3 = i1 + i1 = 10`).
3. Запускаем программу, результат выглядит так: `5 2 10`.

В заключение обращаем Ваше внимание на следующие важные моменты:

1. Операция взятия адреса не может быть применена к числу или константе! Адрес есть лишь у тех объектов, которые хранятся в памяти.
2. Память, выделенная под переменную-указатель при ее объявлении, инициализируется по умолчанию нулевым значением. В Object Pascal существует возможность явно присвоить указателю нулевой адрес, для этого введена специальная константа **nil** (пишут `p1 := nil;`). Попытка доступа к значению, хранящемуся по нулевому адресу, вызывает ошибку времени исполнения программы!
3. К сожалению, Object Pascal не обладает большим набором операций по работе с указателями. Так, мы не можем прибавить к адресу число (т.е. не можем напрямую при помощи встроенных средств, однако мы можем создать эти средства самостоятельно). Тем не менее, в дополнение к рассмотренным операциям к типизированным указателям могут быть применены операции сравнения = и <>, результат которых зависит от того, хранят указатели один и тот же адрес или нет.

Из рассмотренного выше типизированный указатель предстает перед нами как средство создания синонимов для имен переменных. На самом деле, его применение значительно шире, что и будет показано далее в этой главе.

Работа с адресами. Бестиповые указатели

Наряду с типизированными указателями, хранящими адрес вместе с информацией о типе тех данных, что располагаются по этому адресу, существуют так называемые *бестиповые указатели*, у которых отсутствует какая-либо привязка к конкретному типу данных. Для поддержки таких указателей существует тип `Pointer`.

Переменные типа `Pointer` практически во всем подобны типизированным указателям, кроме того факта, что к ним неприменима операция разыменования, поскольку неизвестно, как интерпретировать обнаруженную по адресу информацию. Тем не менее, добраться до значения, хранящегося по адресу в указателе типа `Pointer` возможно, выполнив предварительно приведение типа от `Pointer` к какому-либо типизированному указателю. Рассмотрим соответствующий пример.

```
{ ========================================================================== }
{ Пример 9.4 }
{ Бестиповые указатели }
Program Untype_pointers;
{$APPTYPE CONSOLE}

type
  PWord = ^Word;
var
  pi: PWord;
  p: Pointer;
  i: Word;
begin
  i := 32008;
  p := @i;
  pi := p;
  WriteLn(i, ' ', pi^);
  ReadLn;
```

```
end.
{ ========================================================================== }
```

Выполнив присваивание `pi := p`; мы скопировали адрес в переменную `pi` и получили возможность использовать операцию разыменования для доступа к значению, хранящемуся по этому адресу. В результате имеем: `32008 32008`.

Изменим пример, объявив `pi` как указатель на `Byte`.

```
{ ========================================================================== }
{ Пример 9.5 }
{ Бестиповые указатели. Особенности интерпретации }
Program Untype_pointers;

{$APPTYPE CONSOLE}

type
  PByte = ^Byte;
var
  pi: PByte;
  p: Pointer;
  i: Word;
begin
  i := 32008;
  p := @i;
  pi := p;
  WriteLn(i, ' ', pi^);
  ReadLn;
end.
{ ========================================================================== }
```

Запускаем программу и видим следующий результат: `32008 8`.

Почему же в результате мы получаем 8? Ответ прост: по адресу, на который указывает `pi`, располагается переменная `i`. Эта переменная принадлежит к типу `Word` и представляет собой двухбайтовое неотрицательное целое число. Попробуем понять, как хранится это число. Для этого рассмотрим следующее разложение: 32008 = 125 * 256 + 8. Таким образом, число 32008 хранится в виде совокупности старшего байта (равного 125) и младшего байта (равного 8). Архитектура IBM PC предполагает, что числа хранятся в памяти в перевернутом виде, сначала младший байт, потом старший. Соответственно, преобразовав `pi` к типу “Указатель на байт”, мы получаем при его разыменовании первый байт, относящийся к числу 32008, т.е. младший байт, равный 8.

Бестиповые указатели применяются в тех случаях, когда способ интерпретации значений, хранящихся в оперативной памяти, не известен. В последствии, как правило, происходит преобразование к типизированному указателю. Заметим, что тип `Pointer` удобен тем, что он совместим по присваиванию с указателем на любой тип данных.

В дополнение к существующим для типизированных указателей операциям язык Object Pascal предоставляет функции `Addr` и `Ptr`, которые могут применяться к бестиповым указателям.

```
function Addr(X): Pointer;
```

Функция `Addr` является аналогом операции `@` и выполняет сходные действия, возвращая адрес объекта `X`, хранящегося в памяти. Результат имеет тип `Pointer`.

```
  function Ptr(Address: Integer): Pointer;
```

Функция `Ptr` восполняет некоторый недостаток адресной арифметики Object Pascal, преобразуя целое число в бестиповый указатель. Рассмотрим небольшой пример.

```
{ ========================================================================== }
{ Пример 9.6 }
{ Бестиповые указатели. Функции Addr и Ptr }
Program Untype_pointers2;

{$APPTYPE CONSOLE}

type
  PByte = ^Byte;
var
  pi1, pi2: PByte;
  p: Pointer;
  i: Word;
begin
  i := 32008;
  p := Addr(i);
  pi1 := p;
  pi2 := Ptr(Integer(p) + 1);
  WriteLn(i, ': старший байт: ', pi2^, '; младший байт: ', pi1^);
  ReadLn;
end.
{ ========================================================================== }
```

Разберем переменную `i` на составляющие ее байты. Для этого известным нам способом получим доступ к младшему байту числа через указатель `pi1`. Для того чтобы получить доступ к старшему байту необходимо прибавить к `p` единицу. Сделать это можно, преобразовав `p` к целому типу, прибавив 1, а затем интерпретируя результат как адрес. В результате имеем на экране:

```
  32008: старший байт: 125; младший байт: 8.
```

В следующем примере применим другую технику для разбора числа на составные части, используя массивы:

```
{ ========================================================================== }
{ Пример 9.7 }
{ Указатели и массивы }
Program Pointers_and_arrays;

{$APPTYPE CONSOLE}

type
  PByte = ^Byte;
  TByteArray = array [0..1] of Byte;
var
  i: Word;
  pBuffer: ^TByteArray;
begin
  i := 32008;
  pBuffer := @i;
  WriteLn(i, ': старший байт: ', pBuffer[1], '; младший байт: ',pBuffer[0]);
```

```
  ReadLn;
end.
{ ========================================================================== }
```

Надеемся, что представленный код в дополнительных комментариях не нуждается.

Статическое и динамическое распределение памяти

Поговорим об управлении памятью. Принципиальный вопрос: когда выделяется необходимая программе память, и когда эта память освобождается, т.е. возвращается системе. Интуитивно понятны два возможных варианта.

Первый подразумевает, что вся необходимая программе (или подпрограмме) память выделяется в начале ее работы и освобождается по окончании. С этим вариантом мы работали на протяжении всех предыдущих глав. Соответствующие инструкции автоматически подставляются в код компилятором. В этом случае говорят о *статическом распределении памяти*. К достоинствам данного подхода следует отнести простоту, производительность (возможность оптимизации времени работы) и сравнительно небольшое количество "узких мест", приводящих к ошибкам. Недостаток же всего один, но зато очень серьезный. Он состоит в том, что в рамках статического распределения памяти мы обязаны заранее точно знать, сколько памяти нам понадобится, что бывает далеко не всегда. Вспомним ситуацию с объявлением массива. Ввиду того, что мы не знаем количества элементов на этапе написания программы, мы вынуждены заводить массив заведомо большего размера, в результате чего существенная часть памяти выделяется просто так и вообще не используется. Заметим также, что мы не имеем возможности вернуть неиспользуемую память системе. Справляется с этими недостатками второй вариант – *динамическое распределение памяти*.

Динамическое распределение памяти предполагает выделение и освобождение памяти в те моменты, когда это действительно необходимо во время работы программы. Язык Object Pascal предоставляет богатые возможности для работы с динамической памятью: кроме операций выделения/освобождения памяти, в язык встроены готовые динамические структуры, такие как динамические массивы и длинные строки, скрывающие от пользователя особенности внутренней реализации и существенно упрощающие работу. В предыдущих главах мы изучили практически все, что касается статической организации данных, перейдем теперь к работе с данными динамическим способом.

Динамическое распределение памяти в языке программирования Object Pascal

Вспомним рисунок 9.1 и комментарии к нему в начале главы. Один из элементов рисунка – *Куча (Heap)* – остался неизученным. Настала пора с ним разобраться. Итак, куча – это специальная область в адресном пространстве программы, в которой выделяется память, запрашиваемая динамически, в момент работы программы.

Object Pascal содержит две группы подпрограмм для осуществления операций выделения/освобождения памяти в куче. Рассмотрим их.

Работа с памятью в стиле GetMem, FreeMem

Первая группа предназначена для выделения (освобождения, изменения размеров) памяти с заданием размера обрабатываемого блока ячеек в байтах и работы с этим блоком при помощи механизма бестиповых указателей.

```
procedure GetMem(var P: Pointer; Size: Integer);
```

Процедура `GetMem` выделяет блок памяти размером `Size` (в байтах) и возвращает адрес выделенного блока в переменную-указатель `P`. В случае если выделить память не удалось, возникает ошибка времени исполнения.

```
procedure FreeMem(var P: Pointer);
```

Процедура `FreeMem` применяется при необходимости освободить память, выделенную ранее при помощи `GetMem`. Переменная `P` содержит указатель на эту память. Освобождение памяти означает, что память становится свободной и может быть выделена при следующем обращении к `GetMem`. Заметим, что повторное освобождение одного и того же участка памяти приводит к ошибке. Значение переменной `P` по окончании `FreeMem` не определено.

```
procedure ReallocMem(var P: Pointer; Size: Integer);
```

Процедура `ReallocMem` производит перераспределение ранее выделенного участка памяти, адрес начала которого содержится в переменной `P`, а новый размер – в переменной `Size`. При изменении размера выделенного блока данные, находившиеся в блоке, сохраняются.

Кроме рассмотренных средств, Delphi содержит еще несколько подпрограмм, например, `AllocMem`, которая отличается от `GetMem` тем, что инициализирует память нулевыми значениями. С полным списком подпрограмм и их назначением желающие могут ознакомится в документации или справочной системе Delphi.

Рассмотрим в качестве демонстрационного примера математическую задачу: найти сумму квадратов первых n натуральных чисел.

Издавна известна следующая формула: $1^2 + 2^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$, где $n \in N$ (9.1)

Приведем ее доказательство, воспользовавшись *методом математической индукции*, который состоит в том, что доказываемое утверждение проверяется для начального n (обычно для n = 1) и показывается, что из справедливости при n = k следует справедливость при n = k + 1.

Доказательство:

1. При $n = 1$ имеем: $1^2 = 1$,

$\frac{1(1+1)(2\square 1+1)}{6} = 1$, что и требовалось.

2. Предположим справделивость утверждения при $n = k$, т.е.

$1^2 + 2^2 + ... + k^2 = \frac{k(k+1)(2k+1)}{6}$, где $k \in N$.

Покажем справедливость утверждения при $n = k + 1$, т.е. то, что

$1^2 + 2^2 + ... + (k+1)^2 = \frac{(k+1)(k+2)(2k+3)}{6}$, где $k \in N$. (1)

Для доказателства рассмотрим левую часть (1). $1^2 + 2^2 + ... + (k+1)^2 = 1^2 + 2^2 + ... + k^2 + (k+1)^2 = (2)$

Воспользуемся предположением индукции:

$$(2) = \frac{k(k+1)(2k+1)}{6} + (k+1)^2 = \frac{(k+1)(2k^2+7k+6)}{6} = \frac{(k+1)(k+2)(2k+3)}{6},$$

что и требовалось доказать □

Напишем программу, которая запрашивает n, выделяет память для хранения квадратов первых n натуральных чисел, вычисляет эти квадраты и записывает их в массив, после чего производит суммирование и сравнивает результат с теоретическим, полученным по формуле (9.1).

```
{ ========================================================================== }
{ Пример 9.8 }
{ Указатели и массивы }
Program GetMemEx;

{$APPTYPE CONSOLE}

type
  PByte = ^Word;
var
  n: Word;     { Количество чисел }
  P: Pointer; { Указатель на буфер для хранения квадратов чисел }
  Pt: PWord;  { Указатель, используется при расчетах }
  i: Integer; { Счетчик цикла }
  S: Integer; { Сумма квадратов }
begin
  Write('Введите количество чисел: ');
  ReadLn(n);
  { Выделяем память }
  GetMem(P, n * SizeOf(Word));
  { Вычисляем квадраты и записываем их в буфер }
  { Используем вспомогательный типизированный указатель Pt }
  { для перемещения по буферу. }
  { Компенсируем отсутствие операций + для указателей }
  Pt := P;
  for i := 1 to n do
  begin
    Pt^ := i * i;
    Pt := Ptr(Integer(Pt) + SizeOf(Word));
  end;
  { Считаем сумму }
  Pt := P;
  S  := 0;
  for i := 1 to n do
  begin
    S := S + Pt^;
    Pt := Ptr(Integer(Pt) + SizeOf(Word));
  end;
  WriteLn('По формуле S = ', n * (n + 1) * (2 * n + 1) div 6);
  WriteLn('В результате расчета S = ', S);
  { Освобождаем память }
  FreeMem(P);
  ReadLn;
end.
{ ========================================================================== }
```

Разумеется, результаты прямого суммирования и расчета по формуле совпадут. Обратите внимание на способ перемещения по буферу, состоящий в инициализации вспомогательного указателя значением P, преобразовании его к целому типу, прибавлении размера одного числа в байтах, преобразовании к типу-указателю при помощи функции Ptr и, наконец, разыменовании.

Отметим, что вовсе необязательно считать сумму квадратов чисел, используя буфер для их хранения. В данном случае без этого можно обойтись, но чаще всего обрабатываемые данные бывают нужны неоднократно в ходе работы программы, что приводит к необходимости их хранения.

Работа с памятью в стиле New, Dispose

Теперь поговорим о второй группе подпрограмм, позволяющих выделять и освобождать память динамически. Этот способ ориентирован на работу с типизированными указателями и состоит в использовании процедур New и Dispose.

```
procedure New(var P: Pointer);
```

Процедура New предназначена для выделения памяти с использованием типизированных указателей.

```
procedure Dispose(var P: Pointer);
```

Процедура Dispose предназначена для освобождения памяти с использованием типизированных указателей.

Рассмотрим пример.

```
{ ========================================================================== }
{ Пример 9.9 }
{ New и Dispose }
Program NewEx;

{$APPTYPE CONSOLE}

type
  TPerson = record
    FIO:       String[50];
    BirthdayY: Word;
    BirthdayM: Byte;
    BirthdayD: Byte;
    Phone:     String[15];
  end;
  PPerson = ^TPerson;
var
  P: PPerson;
  F: String[50];
  Y: Word;
  M: Byte;
  D: Byte;
  Ph: String[15];
begin
  Write('Введите ФИО: ');
  ReadLn(F);
  Write('Введите год рождения: ');
```

```
  ReadLn(Y);
  Write('Введите месяц рождения: ');
  ReadLn(M);
  Write('Введите день рождения: ');
  ReadLn(D);
  { Выделяем память }
  New(P);
  { Сохраняем данные }
  P^.FIO := F;
  P^.BirthdayY := Y;
  P^.BirthdayM := M;
  P^.BirthdayD := D;
  P^.Phone := Ph;

  // Обработка

  { Освобождаем память }
  Dispose(P);
  ReadLn;
end.
{ ========================================================================== }
```

Как видите, все достаточно просто. Дополнительное удобство состоит в том, что более не нужно указывать размер выделяемой памяти, так как он известен по типу указателя.

Динамические структуры языка Object Pascal

Динамические массивы

Рассмотрим одну из часто используемых возможностей языка Object Pascal – *динамические массивы*. Разработчикам компилятора удалось создать замечательный механизм по своей простоте и удобству, что бывает совсем не часто. Всем нам хорошо известен обычный (статический) массив, многократно использованный в этой книге к настоящему моменту. В этой главе мы узнали, что можно устранить недостаток, связанный с необходимостью указания размера массива на этапе компиляции программы, при помощи реализации динамических массивов посредством бестиповых указателей, процедур `GetMem`, `FreeMem`, `ReallocMem` и “самодельных” операций перемещения указателя по массиву (см. пример 9.8). Оказывается, Object Pascal реализует подобную структуру на уровне языка. Посмотрим, как ей можно воспользоваться.

Введение в динамические массивы

Для объявления динамического массива используется стандартная конструкция **`array`** без указания диапазона индексов.

```
  type
    <Имя типа данных> = array of <Тип элемента>;
```

Объявим массив элементов типа `Word`.

```
type
  TWordArray = array of Word;
var
  M: TWordArray;
```

Интересно, что из себя представляет переменная `W` – массив неизвестного размера? Все очень просто, на самом деле `M` – это просто указатель. Изначально, после объявления, массив `M` состоит из 0 элементов, соответственно, `M` по смыслу близка к нулевому указателю.

Размер массива устанавливается при помощи процедуры

```
procedure SetLength(var S; NewLength: Integer);
```

Это процедура осуществляет выделение необходимой памяти для хранения `NewLength` элементов типа, указанного при объявлении массива `S`.

Доступ к элементам динамического массива осуществляется обычным образом при помощи операции индексации "`[]`", с учетом того, что диапазон индексов `0..NewLength-1`. Заметим, что компилятор сам заботится не только о выделении памяти, но и о запоминании параметров массива, в частности, количества его элементов.

Вызов `SetLength` с параметром 0 осуществляет освобождение памяти, занимаемой элементами массива, вызов `SetLength` со значением, не равным нулю и отличным от текущего значения, приводит к перераспределению памяти с сохранением значений элементов.

Перепишем рассмотренный ранее пример с вычислением суммы квадратов с использованием динамических массивов.

```
{ ========================================================================== }
{ Пример 9.10 }
{ Указатели и массивы }
Program DynArrEx;

{$APPTYPE CONSOLE}

type
  TWordArray = array of Word;
var
  n: Word;        { Количество чисел }
  i: Integer;     { Счетчик цикла    }
  S: Integer;     { Сумма квадратов  }
  A: TWordArray; { Массив           }
begin
  Write('Введите количество чисел: ');
  ReadLn(n);
  { Выделяем память }
  SetLength(A, n);
  { Вычисляем квадраты и записываем их в массив }
  for i := 1 to n do
    A[i - 1] := i * i;
  { Считаем сумму }
  S := 0;
  for i := 1 to n do
    S := S + A[i - 1];
  WriteLn('По формуле S = ', n * (n + 1) * (2 * n + 1) div 6);
  WriteLn('В результате расчета S = ', S);
  { Освобождаем память }
```

```
  SetLength(A, 0);
  ReadLn;
end.
{ ========================================================================== }
```

Отметим, что текст программы стал намного короче и понятнее. Также обращаем внимание на конструкцию `A[i - 1]` – напоминаем, что динамические массивы всегда индексируются с нуля. Для того чтобы застраховать себя от неприятностей с диапазонами индексов, часто пишут так:

```
  for i := Low(A) to High(A) do
    ...;
```

Функции `Low` и `High` возвращают индексы первого и последнего элементов массива соответственно (первый всегда имеет индекс 0), пользуясь тем, что компилятор хранит информацию о количестве элементов массива[1].

Присваивание и копирование. Сравнение

Известно следующее изречение, неплохо характеризующее то, что многие пользователи думают о компьютерах и программах: "*компьютеры бесподобны: за несколько минут они могут совершить такую грандиозную ошибку, какую не в состоянии сделать множество людей за многие месяцы*" (М. Мичэм).

Надо понимать, что динамическое распределение памяти одновременно является и мощным инструментом программирования, и потенциальным источником труднонаходимых ошибок. Одно из таких "узких мест" – присваивание и копирование динамических массивов. Рассмотрим этот вопрос подробно.

Пусть `A` и `B` есть динамические массивы одинакового типа и размерности. Зададимся вопросом, что будет, если мы напишем в программе `A := B`? Это тем более интересно, если предварительно для `A` не выделялась память. Рассмотрим пример:

```
  var
    A, B: array of Integer;
  begin
    SetLength(B, 10);
    B[5] := 5;
    A := B;
    A[5] := 7;
    ...
  end.
```

На самом деле, подобное присваивание разрешено, и для него не требуется выделения памяти под массив `A`. Однако присваивание в данном случае приводит к созданию второго имени, синонима для доступа к массиву `B`. Теперь он доступен и как `A`. Таким образом, в отличие от обычных массивов, в которых в результате присваивания между однотипными массивами происходило копирование данных, в динамических массивах такого не происходит. Значит, `A[5] := 7` – приводит к тому, что `B[5]` также равно 7.

---

[1] Эстеты могут открыть справочную систему Delphi и обнаружить, что по смещению -4 от начал массива компилятор сохраняет его длину (4 байта), а по смещению -8 – количество ссылок, т.е. информацию о том, сколько раз используется данный массив. Так, например, после присваивания A := B количество ссылок равно 2. Заметим, что от версии к версии компилятора эта структура может меняться, т.е. на это не рекомендуется ориентироваться в программах.

Как осуществить копирование? Простейший способ выглядит так:

```
var
  A, B: array of Integer;
begin
  SetLength(B, 10);
  B[5] := 5;
  SetLength(A, 10); { Выделяем память! }
  for i := Low(A) to High(A) do
    A[i] := B[i];
  ...
end.
```

Существует и встроенная функция `Copy`, которая позволяет осуществить подобные действия.

```
var
  A, B: array of Integer;
begin
  SetLength(B, 10);
  B[5] := 5;
  A := Copy(B, 0, 10);
  ...
end.
```

В данном примере независимо, от того, была ли выделена память под массив `A`, создается новый массив из 10 элементов, в который копируются 10 элементов массива `B`, начиная с 0-го.

В заключение заметим, что применение к динамическим массивам одного типа операции сравнения приводит к тому, что сравнивается не содержимое массивов, а ссылки, т.е. проверяется, не есть ли это синонимы.

Многомерные динамические массивы

Динамические массивы могут быть многомерными. Рассмотрим ситуацию на примере двумерного массива.

Для его создания необходимо использовать конструкцию **`array of`** два раза. Создадим тип данных “Матрица” для хранения таблицы целых чисел и объявим переменную этого типа.

```
type
  TMatrix = array of array of Integer;
var
  M: TMatrix;
```

Для установки размеров матрицы необходимо использовать `SetLength` с 2-мя параметрами – количеством строк и количеством столбцов. Так,

```
SetLength(M, 10, 20);
```

создает матрицу размером 10 x 20. Доступ к элементам реализуется обычным образом, так мы можем написать `M[5, 7] := 9`.

Рассмотрим еще одну интересную возможность, связанную с многомерными динамическими массивами. Оказывается, мы можем написать так:

```
SetLength(M, 10);
```

Что будет в результате? Десять одномерных динамических массивов нулевой длины. Впоследствии мы можем указать для каждого из них свое количество элементов, получив

непрямоугольную таблицу. Рассмотрим применение этого механизма на примере практической задачи.

Многомерные динамические массивы в задаче о поиске оптимального пути

Рассмотрим задачу.

“В Тридевятом королевстве жил да был король. И решил он назначить первым министром самого мудрого из всех возможных кандидатов. Для этого король устроил следующее состязание. Лучшие строители королевства построили лабиринт, устроив его так: в начале лабиринта находилась одна единственная дверь. Пройдя через нее, мудрец оказывался перед двумя дверьми, на следующем уровне дверей было уже три, затем четыре и т.д. По правилам состязаний кандидат на почетную должность в каждый момент времени мог идти только *через одну из двух ближайших дверей*. За проход через каждую дверь мудрец платил фиксированный штраф от 1 до 50 тугриков. Победителем считался мудрец, заплативший наименьший штраф. Схема лабиринта из 500 уровней с указанием штрафов была предоставлена кандидатам за день до соревнований”.

Вооружившись компьютером, помогите мудрецу одержать победу.

Анализ задачи

Попробуем разобраться в постановке задачи.

Что нам дано?

1. Дана схема лабиринта, в котором на первом уровне 1 дверь, на каждом следующем уровне количество дверей увеличивается на единицу.
2. Дан штраф за проход через каждую из дверей.
3. Установлена схема перемещения, в соответствии с которой в каждый момент времени мы можем идти вперед только через одну из двух ближайших дверей.

Что нам надо найти?

Необходимо найти такой порядок прохождения через двери, чтобы заплатить наименьший штраф к моменту выхода из лабиринта (всего нужно пройти 500 уровней).

Некоторые дополнительные соображения:

1. По-видимому, 500 уровней – достаточно много, а времени на раздумья всего один день. Придется учитывать это при решении задачи.
2. Будем далее считать, что уровней у нас n и решать задачу в общем случае, понимая, что n достаточно большое.

Математическая модель

Предлагаем следующую математическую модель для данной задачи: будем представлять лабиринт в виде равностороннего треугольника из чисел, в котором в первой строке – 1 число, во второй строке – 2 числа, ..., в n-ой строке – n чисел. Число соответствует штрафу.

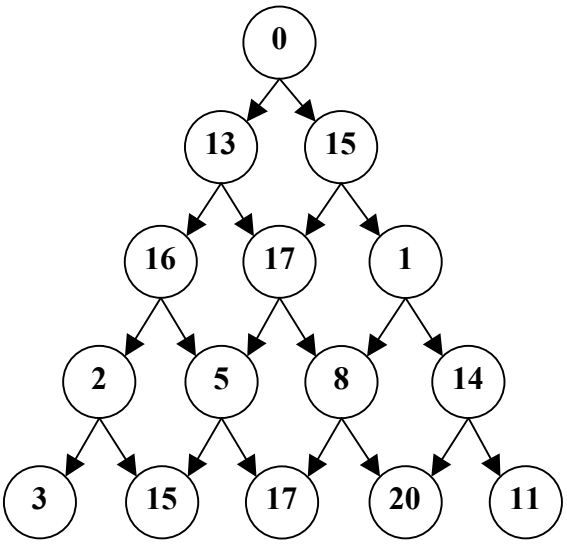


Рис. 9.2. Модель лабиринта (пример)

На рисунке 9.2 приведен пример того, как может выглядеть соответствующий треугольник, где круги – двери, числа – штрафы, стрелками отмечены разрешенные направления движения. Любая из дверей на последнем уровне – выход из лабиринта[1].

Попробуем разработать алгоритм для решения задачи.

Алгоритм 1. Полный перебор

Первое, что приходит в голову, построить переборный алгоритм, который проанализирует все возможные варианты перемещения от начальной двери до последнего уровня, сосчитает сумму штрафа для каждого пути и найдет путь с минимальным штрафом.

Для того, чтобы запрограммировать подобный алгоритм, необходимо прежде всего понять, как построить перебор путей. Нетрудно видеть, что для этого надо знать, что такое путь. Вернее, необходимо построить *модель пути*. Попробуем это сделать.

Заметим, что независимо от выбора двери в каждый момент времени длина всех путей одинакова и составляет n-1. Доказательство этого факта очевидно: за один шаг (под шагом будем понимать осуществление выбора из 2-х дверей) мы независимо от результатов выбора продвигаемся вперед (или вниз, если смотреть на картинку) на один уровень. Таким образом, за 1 шаг мы попадаем на 2-ой уровень, за 2 шага – на 3-ий, а за n-1 шагов – на n-ый.

Таким образом, все пути имеют длину n-1. Теперь разберемся с тем, как закодировать факт выбора. Каждый выбор двери представляет собой переход “влево-вперед” или “вправо-вперед”. Будем обозначать 0 движение влево, а 1 – вправо, тогда в итоге получаем, что путь есть последовательность нулей и единиц длины n-1.

Формально, путь $P = \{p_1, p_2, ..., p_{n-1}\}$, где $p_i \in \{0;1\}$, по $i = \overline{1, n-1}$.

Теперь мы можем перебрать все такие последовательности, для каждой из них сосчитать сумму элементов треугольника и найти путь, на котором сумма достигает минимума, решая поставленную задачу.

---

[1] В таком виде (начиная с треугольника из чисел) с точностью до замены минимизации максимизацией задача предлагалась на международной олимпиаде по информатике в 1994г. в Швеции. Смотри, например, http://olympiads.win.tue.nl/ioi/ioi94/

Попробуем оценить, сколько времени потребуется мудрецу на получение решения. Для этого оценим количество путей. Количество путей есть количество разных последовательностей длины n-1, элементы которой могут иметь 2 разных значения. Известно, что это количество равно 2^{n-1}.

Берем хороший инженерный калькулятор и вычисляем для n = 500,

$2^{499} = 1{,}636695303948070935006594848413 8 * 10^{150}$.

Впечатляет? Похоже, мудрец не доживет до того времени, когда сгенерируются все возможные пути, не говоря уже о подсчете сумм.

Для тех, кого не убедило приведенное число астрономических размеров, представим текст программы для экспериментов. Некоторые пояснения к программе:

1. Программа не решает исходную задачу, она просто генерирует все возможные последовательности 0 и 1 длины n.
2. Идея алгоритма получения последовательностей – взятие {0, 0, …, 0} в качестве исходной последовательности и двоичное прибавление единицы для получения каждой следующей цепочки.

   В результате, например, цепочки длины 3 будут получены в следующем порядке:

   ```
   0 0 0
   0 0 1
   0 1 0
   0 1 1
   1 0 0
   1 0 1
   1 1 0
   1 1 1
   ```

3. Двоичное прибавление единицы осуществляется так: ведется анализ текущей последовательности справа налево. Если последний символ нуль, то он заменяется на единицу, после чего получена новая цепочка. Если последний символ единица, то все стоящие перед ним единицы заменяются нулями, а первый встретившийся нуль заменяется на единицу, что опять же дает новую цепочку. При этом устанавливается ограничение, чтобы алгоритм закончил работу, если последовательность уже состоит из одних единиц.
4. Программа содержит вывод на экран для того, чтобы можно было оценить ее работоспособность при малых n. Для проведения экспериментов для больших n вывод на экран нужно закомментировать.
5. Реализация алгоритма не претендует на оптимальность. Целью являлось сделать ее по возможности максимально понятной. Желающим ознакомиться с этим и другими сходными алгоритмами рекомендуем прекрасную книгу [21], в которой этому посвящен целый раздел. Однако, даже если реализацию можно ускорить в 10 раз (что выглядит сомнительным), ситуация принципиально не изменится. Уже при n = 50 дождаться генерации всех цепочек будет проблематично. Учтите, что данный алгоритм лишь перебирает все пути, не вычисляя суммы.

Итак, вот обещанный текст программы:

```
{ ========================================================================== }
{ Пример 9.11 }
{ Полный перебор - получение всех путей }
Program Enumeration_of_possibilities;
```

```
{$APPTYPE CONSOLE}

var
  np: Word;
  a: array of Byte;

procedure Print;
var
  i: Word;
begin
  WriteLn;
  for i := Low(a) to High(a) do
    Write(a[i], ' ');
end;

procedure Enum(n: Word);
var
  i: Integer;
begin
  SetLength(a, n - 1);
  { Начальная последовательность }
  for i := Low(a) to High(a) do
    a[i] := 0;
  { Получена новая цепочка. Печатаем }
  Print;
  { Прибавление единицы }
  while true do
  begin
    i := High(a);
    if a[i] = 0 then
    begin
      a[i] := 1;
      { Получена новая цепочка. Печатаем }
      Print;
    end
    else begin
      { второе условие гарантирует то, что алгоритм закончит работу }
      while (a[i] = 1) and (i >= Low(a)) do
      begin
        a[i] := 0;
        i := i - 1;
      end;
      if i < Low(a) then
        break; // Уже получена последняя цепочка. Выход
      a[i] := 1;
      { Получена новая цепочка. Печатаем }
      Print;
    end;
  end;
  SetLength(a, 0);
end;

begin
  Write('Введите n: ');
  ReadLn(np);
```

```
  Enum(np);
  WriteLn('Все последовательности получены');
  ReadLn;
end.
{ ========================================================================== }
```

Алгоритм 2. Динамическое программирование (метод Беллмана)

Итак, перебор показал свою бесперспективность. Как помочь мудрецам? Подумаем над нашей задачей еще раз.

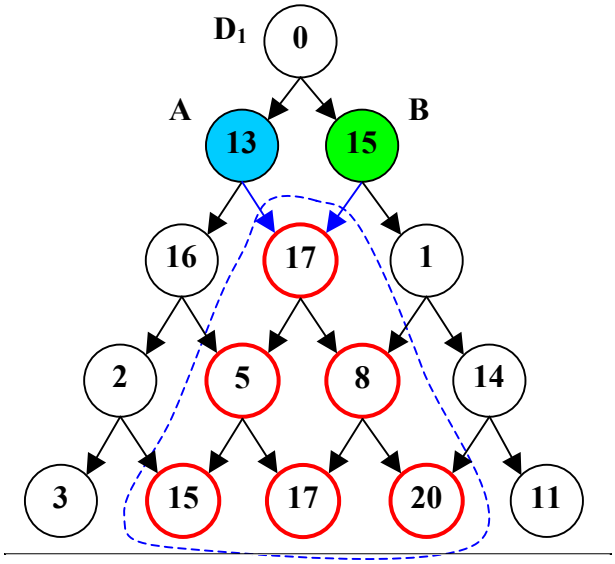


Рис. 9.3. Модель лабиринта. Повторные вычисления

Рассмотрим двери A и B на рисунке 9.3. Нетрудно видеть, что при переборе мы вынуждены обсчитать выделенную пунктиром область 2 раза, один раз для двери A и один раз для двери B. Очевидно, подобные повторы встречаются почти для каждой двери. Это и есть тот ресурс, за счет которого мы должны попытаться ускорить алгоритм – исключить многократные расчеты одного и того же. Как это сделать?

Ответ на этот вопрос дает известный метод Беллмана (часто в литературе подход носит название Динамическое программирование [11]) – любой участок оптимального пути должен быть оптимальным. Что это значит?

Это значит, что если последовательность дверей $\{D_1, \ldots, D_n\}$ – оптимальный (лучший, в нашей задаче с минимальной суммой штрафа) путь с уровня 1 (дверь D_1 фиксирована) до уровня n, то любая ее часть $\{D_i,\ldots,D_j\}$ – оптимальный путь из двери D_i к двери D_j. Действительно, ведь если это не так, то из D_i в D_j есть лучший путь, то можно взять его и вставить в наш “оптимальный вариант пути”, улучшив его, что невозможно по предположению об оптимальности.

Применим эти соображения. Рассмотрим любую дверь предпоследнего уровня (пусть это дверь со штрафом 5 на рис. 9.4). Пройдя через эту дверь, можно выйти из лабиринта ровно двумя способами.

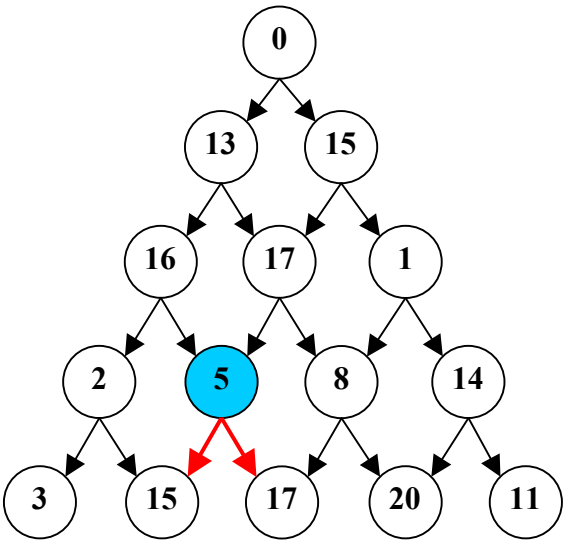


Рис. 9.4. Метод Беллмана. Дверь со штрафом 5

Через дверь со штрафом 15 и через дверь со штрафом 17. Очевидно, что в целях минимизации расходов нужно идти в дверь со штрафом 15, что даст в результате суммарный штраф от прохода двух уровней 5 + 15 = 20.

Это значит, что, попав в отмеченную на рисунке дверь, мы в лучшем случае выйдем из лабиринта, заплатив штраф равный 20. Начав с предпоследнего ряда, мы можем последовательно дверь за дверью рассчитать минимальные штрафы указанным выше способом. В результате, добравшись до входной двери мы получим минимальную сумму штрафа, а вспомнив, в какой из двух вершин достигался минимум, – оптимальный путь из этой вершины в конец. Таким образом, мы построим оптимальный путь.

Приведем на рисунке значения штрафов, рассчитанных “с конца” указанным способом.

Синим выделены стрелки, по которым нужно осуществлять переход от верхней вершины к нижним. Заметим, что оптимальный путь может получиться не один (см. 2 стрелки, выходящие из вершины с оптимальным штрафом 26).

Оценим трудоемкость алгоритма. В соответствии с написанным выше нам необходимо будет рассчитать $1+2+...+n=\frac{n(n+1)}{2}$ значений, что намного лучше предыдущего варианта.

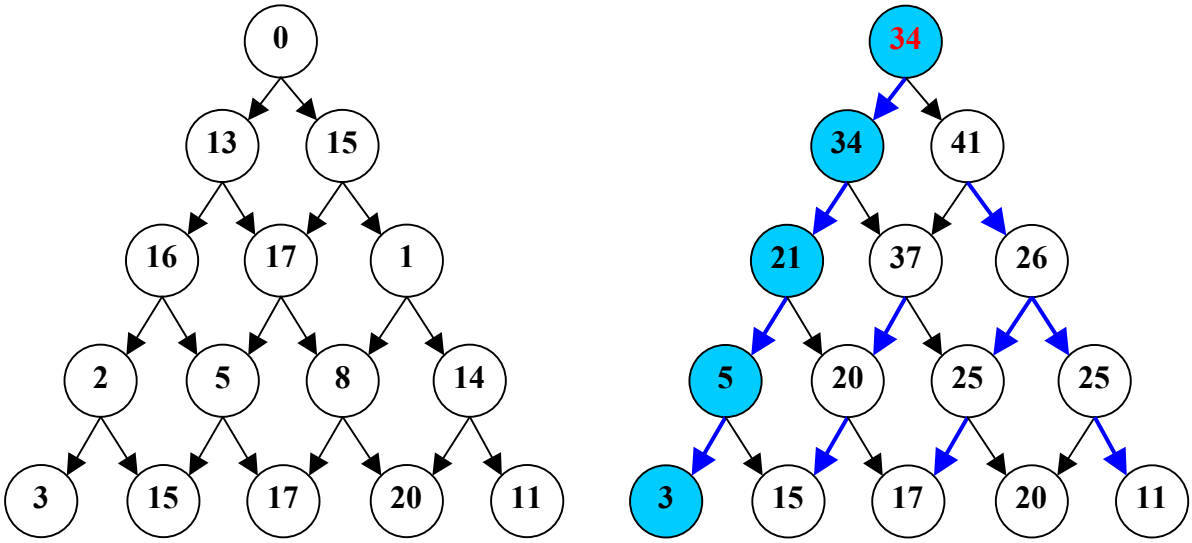


Рис. 9.5. Метод Беллмана. Вспомогательные расчеты

Проектирование

Рассмотрим вопрос проектирования программы, решающей поставленную задачу по второму алгоритму.

Первый вопрос – как представить данные. Язык программирования не содержит средств реализации треугольных структур данных. Будем представлять исходные данные (штрафы) в виде двумерного динамического массива, в первой строке которого 1 элемент, во 2-ой строке – 2 элемента, в n-ой строке – n элементов. Обозначим эту структуру A.

Дополнительно заведем точно такой же массив для вспомогательных расчетов по определению минимального штрафа. Обозначим ее S.

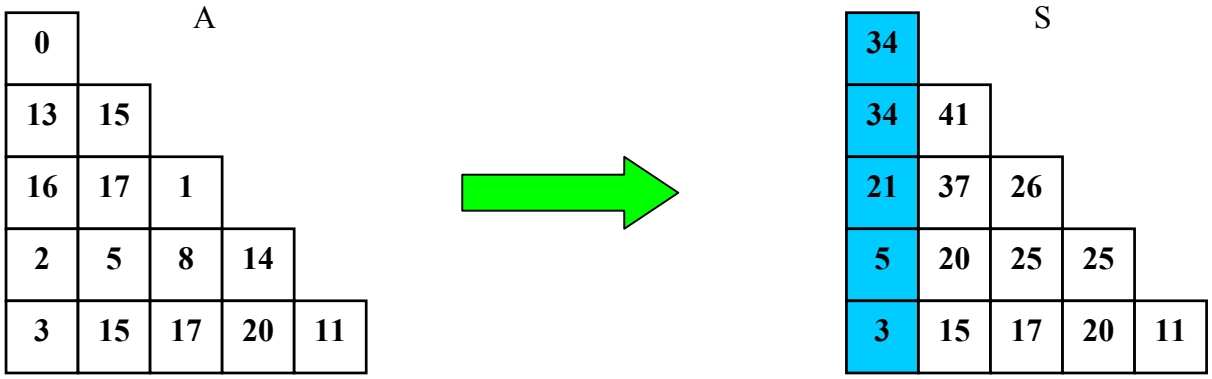


Рис. 9.6. Проектирование данных. Переход к двумерному массиву

Заметим, что рассматриваемые структуры существенно лучше обычного двумерного массива для этой задачи из-за большой (почти двукратной) экономии памяти.

Опишем математически алгоритм получения массива S, рассмотренный выше в словесной форме.

$$S_{n,j} = A_{n,j}, \text{ при } j = \overline{1,n}$$

$$S_{k,j} = A_{k,j} + \min\left\{S_{k+1,j}, S_{k+1,j+1}\right\}, \text{ при } k = \overline{1,n-1}, \quad j = \overline{1,k}$$

Реализация

Не забывая о том, что динамические массивы индексируются с нуля, а матрица S должна пересчитываться с конца, выполним следующую программную реализацию. Матрицу A будем генерировать случайным образом, каждый из Вас легко может организовать здесь ввод данных из файла.

```
{ ========================================================================== }
{ Пример 9.12 }
{ Метод Беллмана - получение оптимального пути }
Program Bellman;

{$APPTYPE CONSOLE}

uses Math;

const
  { Максимальный штраф }
  MaxPenalty = 10;
type
  { Штраф }
  TPenalty = 1..MaxPenalty;
  { Исходный массив штрафов }
  TPenaltyArr = array of array of TPenalty;
  { Вычисляемый массив минимальных штрафов }
  TSumArr = array of array of Cardinal;
var
  { Исходный массив штрафов }
  A: TPenaltyArr;
  { Вычисляемый массив минимальных штрафов }
  S: TSumArr;
  { Количество уровней }
  n: Word;

{ Инициализация массивов, ввод данных }
procedure InitArrays;
var
  i, j: Integer;
begin
  Randomize;
  Write('Введите n: ');
  ReadLn(n);
  { Установка размеров }
  SetLength(A, n);
  SetLength(S, n);
  for i := 0 to n - 1 do
  begin
    SetLength(A[i], i + 1);
    SetLength(S[i], i + 1);
  end;
  { Инициализация и ввод данных }
  for i := 0 to n - 1 do
    for j := 0 to i do
    begin
      S[i, j] := 0;
      A[i, j] := Random(MaxPenalty - 1) + 1;
```

```
    end;
end;

{ Освобождение памяти }
procedure DoneArrays;
var
  i: Integer;
begin
  for i := 0 to n - 1 do
  begin
    SetLength(A[i], 0);
    SetLength(S[i], 0);
  end;
  SetLength(A, 0);
  SetLength(S, 0);
end;

{ Вывод результатов на экран }
procedure OutputResults;
var
  i, j: Integer;
begin
  WriteLn('Matrix A: ');
  for i := 0 to n - 1 do
  begin
    for j := 0 to i do
      Write(A[i, j], ' ');
    WriteLn;
  end;
  WriteLn('Matrix S: ');
  for i := 0 to n - 1 do
  begin
    for j := 0 to i do
      Write(S[i, j], ' ');
    WriteLn;
  end;
  WriteLn('Минимальный штраф ', S[0, 0]);
end;

{ Вычисление S }
procedure Calculate;
var
  i, j: Integer;
begin
  for j := 0 to n - 1 do
    S[n - 1, j] := A[n - 1, j];
  for i := n - 2 downto 0 do
    for j := 0 to i do
      S[i, j] := A[i, j] + Min(S[i + 1, j], S[i + 1, j + 1]);
end;

begin
  InitArrays;
  Calculate;
  OutputResults;
  DoneArrays;
```

```
  ReadLn;
end.
{ ========================================================================== }
```

Запускаем... Программа работает моментально даже при `n = 1000`. Теперь мы победим, даже если на подготовку к состязанию дадут 5 минут. Естественно, при больших `n` нужно отключить вывод массивов `A` и `S` на экран.

Вы были внимательны? Тогда Вы должны были заметить, что мы нашли только сам штраф, но не нашли путь. Задача нахождения пути не приведет к большим изменениям в программе. Один из вариантов – завести еще один массив той же конфигурации, в котором запоминать, какую из дверей мы выбираем на каждом шаге. После заполнения `S` необходимо будет пройти от первой двери к последнему уровню, руководствуясь теми данными, которые будут в третьем массиве. Предлагаем Вам произвести изменения в программе самостоятельно.

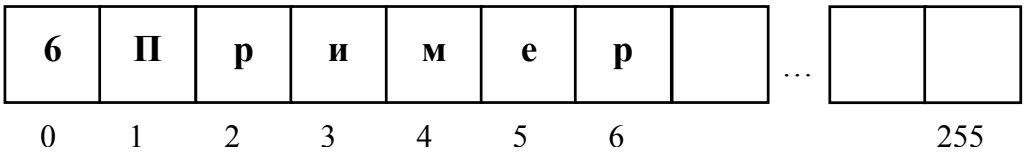
Обработка строковой информации. Короткие и длинные строки

В главе 4 мы сказали несколько слов о строках в языке Object Pascal. Теперь мы знаем достаточно для того, чтобы поговорить о строковых типах более подробно.

Строки в стиле Pascal 7.0

Как говорилось в главе 4, обычные строки в языке Pascal представляли собой массив символов длиной не более 255. Нулевой байт содержал фактическую длину строки, остальные 255 – символы. Сам строковый тип данных при этом назывался `String`, а элемент строки принадлежал типу `Char`. Работа со строкой велась как с обычным массивом, доступ к нулевому символу был несколько затруднен (требовал преобразования типа), вместо этого существовал способ получения длины строки при помощи функции `Length`.

```
var
  S: String;
begin
  S := 'Пример';
end.
```



Разумеется, в ячейках содержатся не сами буквы, а их коды согласно таблице кодов ASCII.

При написании программ мы могли пользоваться следующими возможностями (индексация символов везде ведется с единицы):

```
function Length(S): Integer;
{ Возвращает длину строки }

procedure Insert(Source: String; var S: String; Index: Integer);
{ Вставляет подстроку Source в строку S с позиции Index }

procedure Delete(var S: String; Index, Count: Integer);
{ Удаляет из строки S Count символов с позиции Index }

function Copy(S; Index, Count: Integer): String;
{ Копирует из строки S Count символов с позиции Index }

function Pos(Substr: String; S: String): Integer;
{ Ищет первое вхождение слева подстроки Substr в строке S   }
```

```
  { Возвращает номер символа, с которого начинается подстрока }
  { или 0, если она не обнаружена                              }
```

В дополнение можно отметить следующее:

1. `S := ''` – создание пустой строки;
2. `c := s[i]` – взятие i-го символа строки;
3. `s[i] := c` – запись символа в i-ю позицию в строке;
4. `s := s1 + s2` – строка `s2` пристыковывается к строке `s1` и результат записывается в `s`.
5. `s1 := s2` – копирование содержимого строки `s2` в строку `s1`.
6. `s1 < s2` (и другие операции сравнения) – лексикографическое сравнение (по алфавиту).

Кроме того, мы могли ограничить количество символов в строке, написав, например, `String[20]` в качестве типа данных.

В принципе, перечисленного набора операций достаточно, для того чтобы выполнить любые действия над строковыми данными.

Рассмотрим задачу.

“Дана строка текста. Удалить все цифровые символы”.

Решение 1

Рассмотрим простой вариант реализации с использованием дополнительной памяти. Будем действовать так: создадим новую строку. Будем проверять все ее символы по очереди. Символы, не являющиеся цифровыми, будем переписывать в новую строку.

```
{ ========================================================================== }
{ Пример 9.13 }
{ Удаление цифр из строки с использованием дополнительной памяти }
Program SimpleDelete;

{$APPTYPE CONSOLE}

var
  i: Integer;
  s1, s2: String;
begin
  Write('Введите строку: ');
  ReadLn(s1);
  s2 := '';
  for i := 1 to Length(s1) do
    { Пользуемся упорядоченностью кодов символов в таблице кодов ASCII }
    if (s1[i] < '0') or (s1[i] > '9') then
      s2 := s2 + s1[i];
  WriteLn(s2);
end.
{ ========================================================================== }
```

Решение 2

Теперь попробуем устранить недостаток, связанный с использованием дополнительной памяти. Кажется, что самый простой способ состоит в том, чтобы удалять из строки `s1` каждый найденный цифровой символ. Вносим изменения в предыдущую реализацию:

```
{ ========================================================================== }
{ Удаление цифр из строки. Пример НЕ РАБОТАЕТ! }
Program MistakeInDelete;

{$APPTYPE CONSOLE}

var
  i: Integer;
  s: String;
begin
  Write('Введите строку: ');
  ReadLn(s);
  for i := 1 to Length(s) do
    { Пользуемся упорядоченностью кодов символов в таблице кодов ASCII }
    if (s[i] < '0') or (s[i] > '9') then
      Delete(s, i, 1);
  WriteLn(s);
end.
{ ========================================================================== }
```

Данная реализация представляет собой пример типичной ошибки, которую допускают многие начинающие программисты. Вспомним, как функционирует цикл **for**. Во-первых, значения пределов изменения переменной цикла вычисляются до начала работы цикла. А это значит, что удаление символов никак не влияет на количество итераций, оно подсчитано в начале. Во-вторых, после удаления символа строка сдвигается влево и на `i-e` место встает новый символ, также подлежащий проверке. А цикл **for** заботливо увеличит `i` на единицу, в результате чего мы этот символ вообще выпустим из рассмотрения.

Напрашивается вывод: в случаях, когда строка изменяется в размерах в ходе цикла, оператор **for** обычно не годится для решения задачи. Используем **while**.

```
{ ========================================================================== }
{ Пример 9.14 }
{ Удаление цифр из строки без использования дополнительной памяти }
Program RightDelete;

{$APPTYPE CONSOLE}

var
  i: Integer;
  s: String;
begin
  Write('Введите строку: ');
  ReadLn(s);
  i := 1;
  while (i <= Length(s)) do
    { Пользуемся упорядоченностью кодов символов в таблице кодов ASCII }
    if (s[i] < '0') or (s[i] > '9') then
      Delete(s, i, 1)
    else
      i := i + 1;
  WriteLn(s);
end.
{ ========================================================================== }
```

Строки в стиле Object Pascal

В языке Object Pascal ситуация кардинально изменилась. Рассмотренный только что тип данных был переименован в `ShortString` (отдельный символ по прежнему имеет тип `Char`), а в дополнение к этому введен тип данных `AnsiString` (отдельный символ имеет тип `AnsiChar`), который представляет динамически распределяемые строки. Длина этих строк лимитирована ... 4 Гигабайтами!

`ShortString`, `AnsiString` – какая-то путаница, может воскликнуть наш читатель. А что же делать со старыми программами, где был просто `String`? Ничего страшного, с большими шансами старые программы будут работать и дальше. Директива компилятора `{$H}` отвечает за то, какие именно строки используются в программе при упоминании типа `String`. По умолчанию установлено `{$H+}`, что означает `String = AnsiString`. Т.е. по умолчанию все строки – длинные, а если надо иначе, можно использовать тип `ShortString`. Если Вы хотите изменить это соглашение, установите директиву `{$H-}`, тогда `String = ShortString`, а при необходимости использовать длинные строки Вы можете написать `AnsiString`. Относительно символов все еще проще – `AnsiChar` и `Char` – это одно и то же.

Итак, будем в дальнейшем считать, что `String` – длинная строка. Прежде всего, хотелось бы отметить, что к таким строкам применимы все те операции и функции, которые были рассмотрены в предыдущем разделе. Дополнительно, в отличие от Pascal 7.0 система программирования Delphi содержит огромное количество подпрограмм обработки строк, содержащихся в модуле `StrUtils`. Чего там только нет: интеллектуальные поиски, удобные операции взятия подстрок, преобразование к другому регистру, перекодировка из одной кодировки символов в другую, выделение слов и многое другое. Так, например, функция `PosEx` является расширением функции `Pos`, позволяя проводить поиск не с начала строки, а с любого символа. Другой модуль, `SysUtils` содержит функции форматирования строк, основная из которых, `Format`, позволяет производить подстановку параметров разных типов в строку, управлять форматом вывода числа и т.д. Обязательно познакомьтесь с содержимым этих модулей в справочной системе (формат книги, к сожалению, не позволяет вместить еще и эту информацию).

По своей организации длинные строки в существенной степени похожи на изученные нами динамические массивы. Но есть и серьезные отличия. Рассмотрим их.

Итак, длинные строки объявляются также как обычные. Длина строки может быть установлена при помощи подпрограммы `SetLength`. Кроме того, строка автоматически перераспределяется при попытке присвоить ей значение (у динамических массивов это не так). Заметим, что также как и для массивов язык Object Pascal использует механизм подсчета ссылок, чтобы удалить строку из памяти, когда она более никому не нужна. Фундаментальное отличие состоит в использовании для строк так называемой “copy-on-write” техники. Для понимания этого вопроса рассмотрим пример:

```
var
  A, B: String;
begin
  A := 'Привет!';
  B := A;
  ...
end.
```

В данном примере после присваивания `B := A` строки `A` и `B` указывают на одну и ту же структуру в оперативной памяти. При этом, механизм подсчета ссылок запоминает, что на участок памяти ссылаются сразу 2 объекта.

Но! При попытке изменить содержимое строки `B`, компилятор создаст в памяти новую строку и мы получим 2 разных строки.

```
var
  A, B: String;
begin
  A := 'Привет!';
  B := A;
  B[3] := 'И';
  ...
end.
```

В результате `A = 'Привет!'`, но `B = 'ПрИвет!'`. Заметьте существенную разницу с динамическими массивами.

И еще о строках (осталось за кадром)

Мы считаем, что изложенной информации более чем достаточно для написания программ, работающих со строками. Однако язык Object Pascal содержит еще два строковых типа – `WideString` и `PChar`.

Тип `WideString` рассчитан на то, что символы могут принадлежать не таблице кодов ASCII (однобайтовый код), а таблице кодов Unicode (двухбайтовый код). Несмотря на то, что кодировка Unicode активно продвигается в последние годы, подавляющее большинство программ в Европе, Австралии и Америке по-прежнему работают с однобайтовыми символами. При необходимости, с типом `WIdeString` можно ознакомиться в документации или справочной системе, его устройство почти не отличается от уже изученного материала.

Тип `PChar` предназначен для написания программных систем, которые работают с функциями Windows API или другими программами/библиотеками, написанными на C/C++, где строки представляют собой последовательность символов, заканчивающиеся нулем. Потребность в работе с типом `PChar` возникает нечасто, при необходимости можно ознакомиться с данным материалом в справочной системе.

Выводы

В главе 9 мы познакомились с одним из наиболее сложных вопросов практического программирования – динамическим распределением памяти. Проникнув в дебри адресного пространства программы, мы бодро прошагали по всем его сегментам, забрались в кучу, изучили, как работать с указателями – специальным средством, позволяющем нам общаться на ты с этим непростым механизмом. При изучении материала мы рассмотрели большое количество примеров, иллюстрирующих как надо и как не надо работать с адресами и динамической памятью. Наряду с этим мы изучили встроенные в язык Object Pascal динамические массивы и длинные строки, два необычайно полезных инструмента для минимизации усилий по написанию мощных программ, способных решать сложные задачи.

Теперь мы по праву можем сказать, что умеем многое. Многое, но не все, ведь нет предела совершенству. В следующей главе мы изучим последнюю тему, затрагиваемую в этой книге, последнюю по порядку, но отнюдь не по ее важности – объектно-ориентированное программирование.

10. Объектно-ориентированное программирование

В 1980-х годах объектно-ориентированное программирование будет занимать такое же место, какое занимало структурное программирование в 1970-х. Оно всем будет нравиться. Каждая фирма будет рекламировать свой продукт как созданный по этой технологии. Все программисты будут писать в этом стиле, причем все по-разному. Все менеджеры будут рассуждать о нем. И никто не будет знать, что же это такое.

Rentsch, T. September 1982.
Object-Oriented Programming.
SIGPLAN Notices vol.17(12), p.51.

Ну вот, уважаемый читатель, мы и подошли к заключительной главе нашей книги. Нам кажется, что сейчас самое время остановиться, осмотреться и качественно оценить, что нам удалось изучить. Начали мы наше путешествие с основных этапов разработки программ: поговорили об анализе предметной области и постановке задачи, осознали важность построения математической модели и разработки алгоритма. Затем обсудили средства разработки и программирования в среде Windows, совершили стремительное восхождение к вершинам языка Object Pascal. Настало, наконец, время задаться последним вопросом нашей книги: “Что же такое *Object* в названии языка программирования, с которым мы работали все это время”?

В данной главе мы с Вами познакомимся со значением этого слова в названии языка, вернее с его смысловым наполнением. Узнаем, как сегодня разрабатываются по-настоящему большие программы. Выглядит заманчиво? Тогда за дело!

И снова о технологиях

Вернемся к материалу главы 5. В этой главе мы определили программирование как технологический процесс и рассмотрели понятие *технологии программирования*. В ходе дальнейшего изложения мы познакомились с двумя технологиями – структурным и модульным программированием. Достоинства этих подходов к разработке программ не вызывают сомнений. Это внесение порядка и стройности изложения в тексты программ, это разбиение задачи на более простые подзадачи и решение подзадач по частям, это некоторые возможности по раздельной компиляции (а, следовательно, и упрощению отладки) и повторному использованию кода и многое другое. Однако жизнь не стоит на месте. Аппаратная база компьютеров развивается семимильными шагами. Интуитивно ясно, что развитие вычислительной техники должно

приводить к возможности решения при ее помощи все более или более сложных задач, и это действительно так, однако обратная сторона медали состоит в необходимости разработки все более и более сложных программ. Программы становятся большими (даже очень большими), а разработка – коллективной. Объем работы увеличивается, коллективы разрастаются, код “разбухает”, и появляются новые проблемы, которых не было раньше. Неожиданно выясняется, что возможности структурного и модульного программирования ограничены и зачастую уже не позволяют добиваться желаемого результата (либо ничего не работает, либо проект не укладывается в сроки, либо в бюджет, либо через год после написания программы выясняется, что ее невозможно модифицировать и т.д.)

Для большего понимания сказанного выше приведем пример из другой предметной области. Вспомним, как радовались люди 30 лет назад, когда им удавалось приобрести телевизор. Быть может, Вы еще застали эти огромные ящики с лампами (например, используете их в качестве тумбочки). Современные телевизоры, стоящие у большинства из нас дома, сделаны по совершенно другой технологии. По своим техническим и прочим характеристикам они значительно превосходят своих ламповых предков. Понадобилось улучшить параметры – разработали новые технологии и новую элементную базу. В наши дни вы можете обнаружить в соответствующих магазинах телевизоры, в которых отсутствует электронно-лучевая трубка. Эти телевизоры являются “плоскими”, занимают мало места, их можно повесить на стенку и т.д. Понадобилось улучшить параметры – вновь появились новые технологии и новая элементная база.

Аналогично и в программировании. Нужно оптимизировать процесс создания программ – разрабатываются новые технологии. Одна из них основана на объектном подходе и в русскоязычной литературе носит название объектно-ориентированное программирование. В данной главе мы рассмотрим основные идеи этого подхода и способы его реализации в языке программирования Object Pascal. Сразу оговоримся, что данной теме посвящено немало прекрасных книг и монографий. Мы не ставим целью полностью рассмотреть все аспекты этой большой и сложной темы. Мы сделаем введение в тему, полагая, что наш читатель при необходимости (которая у него наверняка возникнет, если он захочет связать свою жизнь с программированием) продолжит изучение самостоятельно. Сразу оговоримся, что данная тема является достаточно сложной. При первой возможности мы будем иллюстрировать сказанное примерами для лучшего понимания материала.

Объектный подход

Не вдаваясь в историю развития *объектно-ориентированной технологии программирования* (прекрасный обзор содержится, например, в [5]), рассмотрим ее основное смысловое наполнение. В основе этой технологии лежит так называемый *объектный подход*[5, 33, 36, 39, 44]. Для того чтобы понять, в чем он заключается, рассмотрим его основные отличия от других традиционных подходов к разработке программ.

Алгоритмическая и объектная декомпозиции

Известный нам к настоящему моменту способ разработки связан с представлением программы как совокупности данных и алгоритмов, связанных друг с другом. В рамках этого подхода анализ

предметной области (а именно с него все и начинается) представляет собой попытку получить ответы на следующие вопросы:

- Какую информацию нам необходимо хранить (данные)?
- Какими алгоритмами мы будем обрабатывать эти данные?
- Каковы типовые сценарии работы с анализируемой областью (какие алгоритмы нужно и в каком порядке можно выполнять)?

Как мы обычно отвечаем на эти вопросы? Сначала определяем структуры для представления данных предметной области. После этого наращиваем функциональность программы, дописывая к ней алгоритмы, некоторые из которых связаны друг с другом. Эти алгоритмы получают на вход и возвращают в качестве выхода данные из предметной области. В результате, в качестве основного "*строительного блока*" в программе выступает *алгоритм*. Итак, наша обычная модульная программа есть

Данные (разбитые на модули) + Алгоритмы (разбитые на модули)

Обратим внимание на несколько моментов.

1. Как уже говорилось выше, алгоритмы связаны с обрабатываемыми ими данными.
2. Эти связи крайне трудно вычленить непосредственно из текста программы, они существуют лишь в голове программиста то непродолжительное время, пока он пишет соответствующий фрагмент программы. Через год после ее написания понять что, с чем и как связано – грандиозная проблема.

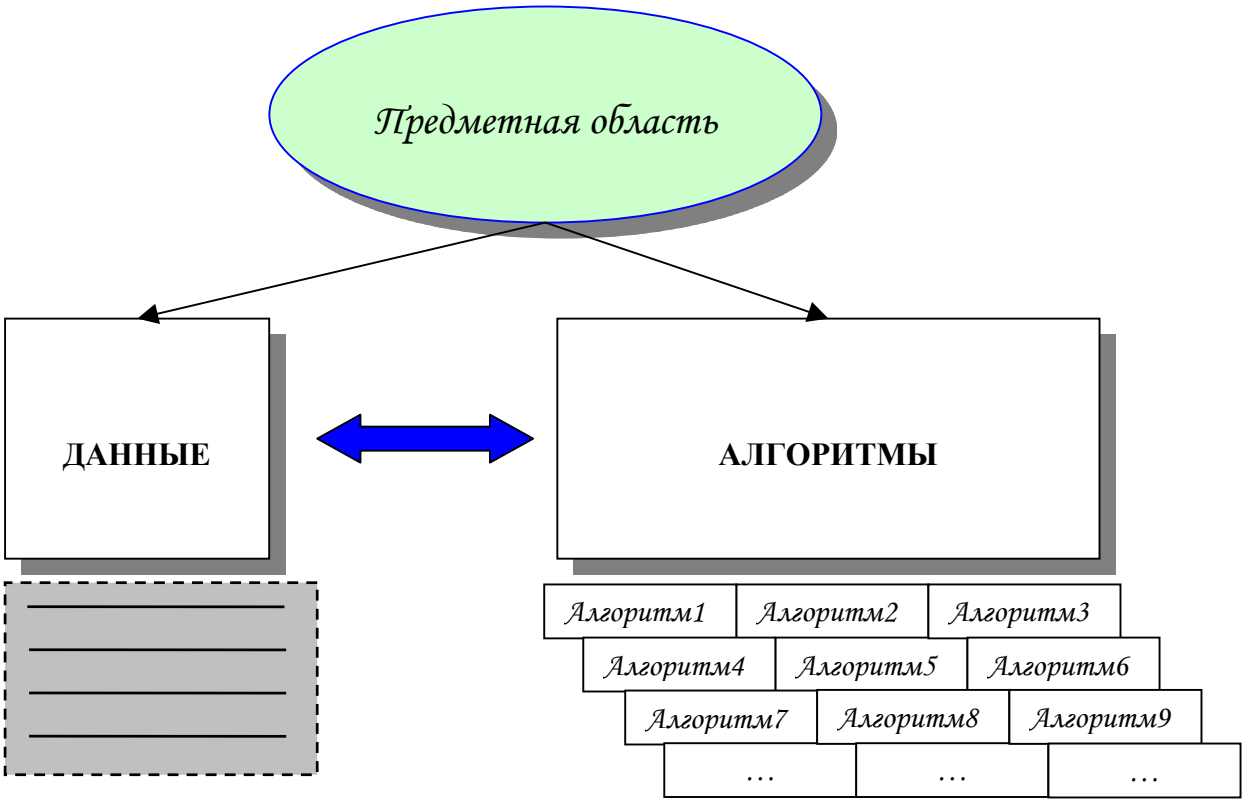


Рис. 10.1. Алгоритмическая декомпозиция. Основной "строительный блок" – алгоритм

3. Замена представления реально существующих объектов предметной области абстрактными структурами и алгоритмами для их обработки отрицательно сказывается на понимании того, “как оно там на самом деле все устроено”. Одним словом, подобное представление неестественно, что плохо.
4. Сплошь и рядом встречаются “перекрестные ссылки” между модулями. В простейшем случае А использует В, а В использует А. Поверьте, все бывает намного более запутано. На этапе написания проекта подобные “обходные пути” – перекрестные ссылки – строго запрещаются. Но как проконтролировать, что этому запрету будут строго следовать? Наивно полагать, что все 100% исполнителей до конца понимают суть проблемы, которая может возникнуть от того, что они поленятся использовать специальный сложный механизм, пойдя вместо этого напрямик. Итог: в модульных программах почти нет средств контроля правильности действий Ваших коллег (мы сами всегда действуем верно, не так ли?)
5. В модульных программах слабо представлены средства, которые позволили бы легко модифицировать код, заменяя отдельные структурные части программы другими, более совершенными. Декларировано, что такие средства есть, но их мощь оставляет желать лучшего.

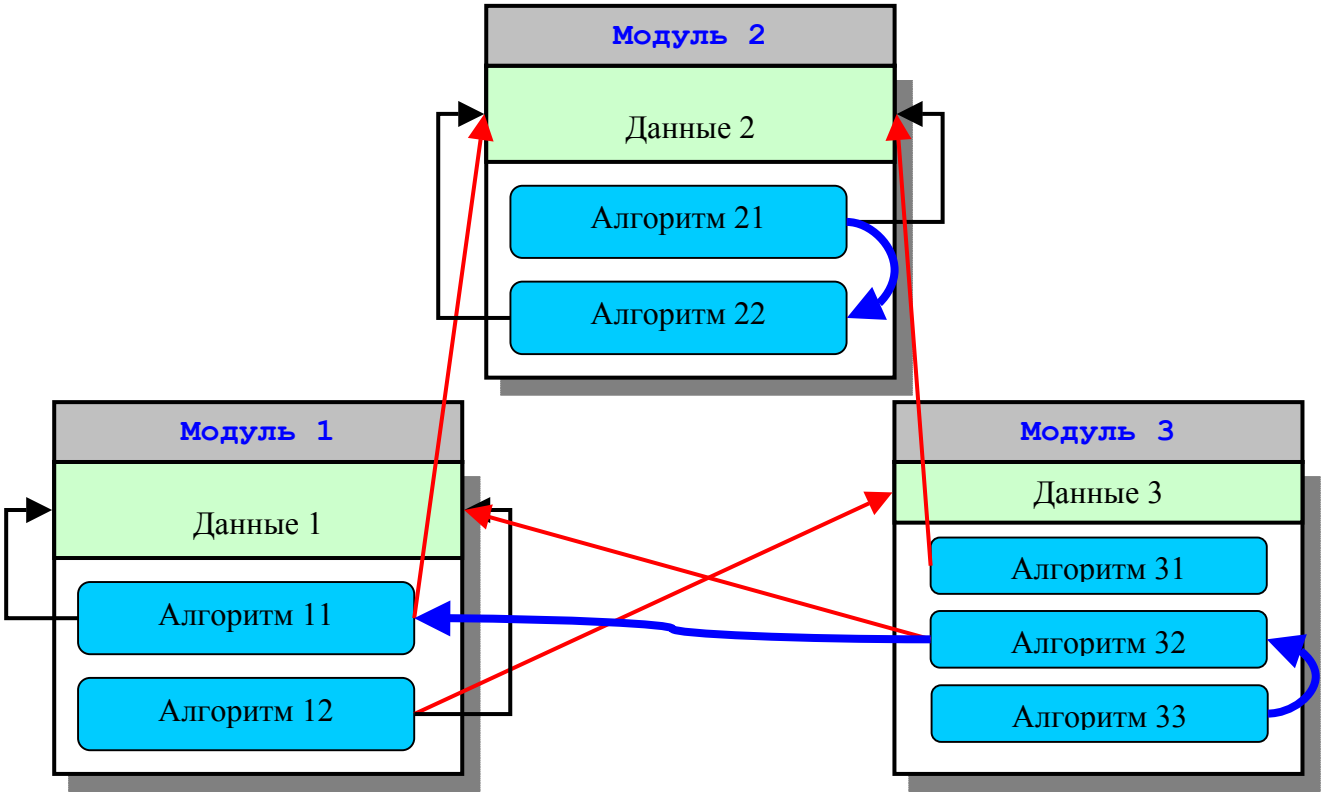


Рис. 10.2. Типовая схема модульной программы

Этот список можно продолжать. Мы показали лишь некоторые из проблем, но уже этого достаточно, чтобы понять необходимость изучения другого подхода к анализу, проектированию и разработке программ.

В основе объектного подхода лежит так называемая *объектная декомпозиция*, смысл которой состоит в том, что предметная область рассматривается как совокупность сущностей (абстракций), которые характеризуются своими данными и обладают строго определенным поведением, описываемым алгоритмами.

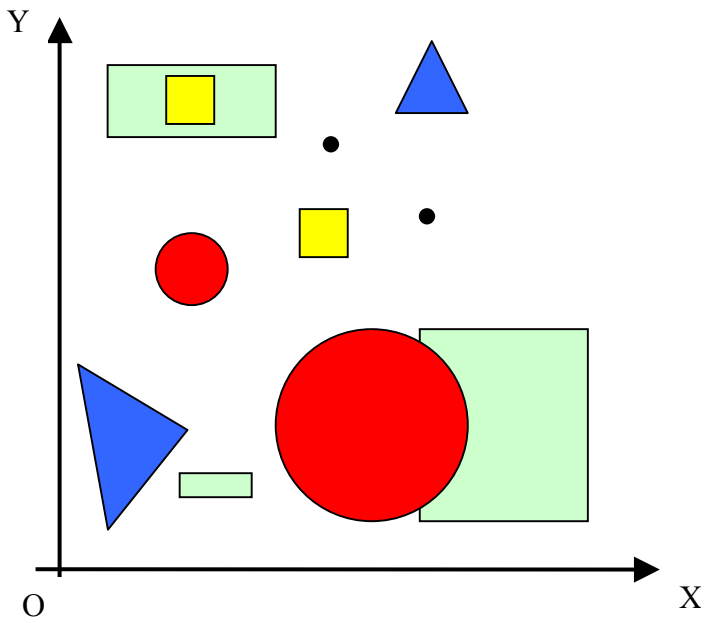


Рис. 10.3. Пример предметной области: координатная плоскость с фигурами

Тем самым, если раньше мы могли рассматривать координатную плоскость как иксы и игреки, а также алгоритмы рисования, скрытия, вычисления площадей и т.д., то теперь для нас координатная плоскость – прежде всего набор фигур определенных типов. Рассмотрим объектную декомпозицию для этого примера. Для начала нарисуем пример координатной плоскости с фигурами. Пусть нашей задачей является разработка программных средств для отображения подобных наборов фигур в декартовой системе координат на плоскости и выполнения над ними типовых операций (отображение, скрытие, перемещение, вычисление площадей). Будем считать, что на рисунке представлена типовая координатная плоскость для рассматриваемой задачи.

Следуя принципам объектной декомпозиции, проведем *анализ предметной области*. Для этого выделим основные ее элементы, концентрируя внимание не на том, сколько там каких фигур (на разных плоскостях будет по-разному), а на том, какие именно фигуры там есть.

Вот результат нашего анализа (основные *абстракции*, если говорить научно):

1. Точка.
2. Квадрат.
3. Круг.
4. Треугольник.
5. Прямоугольник.

Все? Больше ничего не видите? А если подумать?

Если подумать, то становится ясно, что сама Координатная плоскость – шестой пункт нашего списка основных абстракций. Она тоже имеет некоторые свои данные и четко выраженное поведение (понятно, какие алгоритмы ее обрабатывают).

Вот мы и сделали *первый шаг – выделили абстракции*.

Второй шаг – необходимо понять, какими данными характеризуются выделенные нами абстракции.

1. *Точка*: (x, y) – координаты.
2. *Квадрат*: (x, y, a) – левый нижний угол и сторона. Стоп! Внимательный читатель должен здесь заметить, что так мы описываем квадраты, стороны которых параллельны осям координат. Пусть для простоты именно такие квадраты нас интересуют.
3. *Круг*: (x, y, r) – центр и радиус.
4. *Треугольник*: (x1, y1, x2, y2, x3, y3) – координаты вершин.
5. *Прямоугольник*: (x1, y1, a, b) – левый нижний угол и длины сторон (то же замечание, что и для квадрата).
6. *Координатная плоскость* – набор фигур типов 1-5. Хотелось бы написать “массив фигур”. Хотелось бы, но … Элементы разнотипные. Бороться с этой проблемой мы научимся позже, а пока скажем обтекаемо – набор.

Некоторые соображения по поводу проведенного анализа данных:

- помимо указанного выше способа прямоугольник может быть представлен: координатами двух точек – противоположных углов – (x1, y1, x2, y2) или как два объекта “Точка” – Точка1 и Точка2;
- квадрат и круг с точки зрения реализации данных – одно и то же с точностью до обозначений;
- треугольник может быть представлен как (Точка1, Точка2, Точка3).

Как видим, результатом анализа редко бывает единственный возможный вариант описания предметной области. Искусство аналитика – сделать это описание наиболее удобным для дальнейшего использования.

Вот мы и сделали *второй шаг – определили, какими данными описываются наши абстракции*

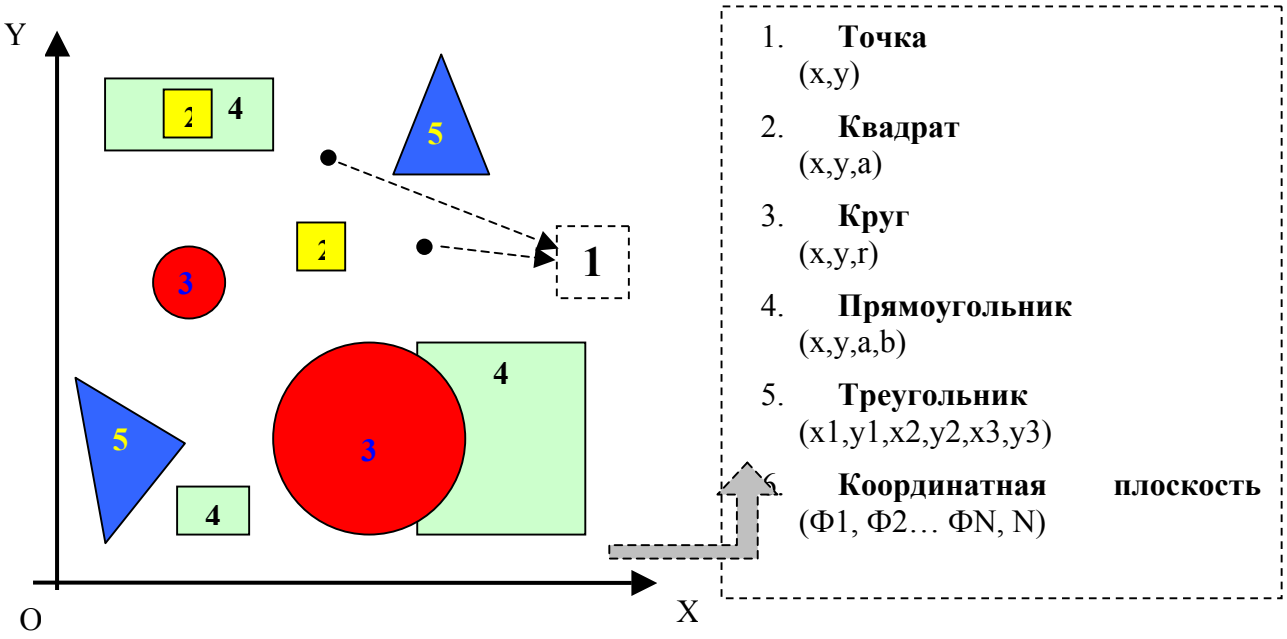


Рис. 10.4. Пример предметной области: координатная плоскость с фигурами. Параметры фигур

Третий шаг – определение, какими алгоритмами обрабатываются эти данные для каждого типа абстракций.

Введем основные операции:

1. **Show** – показать.
2. **Hide** – скрыть.
3. **S** – считать площадь (кроме точки).
4. **MoveTo**(dx, dy) – переместить на dx вправо и dy вверх (кроме координатной плоскости).

Список можно продолжать, но нам на первое время должно этого хватить. Заметим, что все эти операции есть у каждого типа абстракций 1-6, но вот работают они для них по-разному.

Ну вот, с анализом все или почти все. Осмотрим бегло складывающуюся картину и оценим, не забыли ли мы что-то важное? Конечно, забыли! Признак видимости. Если мы собираемся рисовать и скрывать фигуры, то у каждой из них обязательно должен быть признак видимости.

Итог: добавляем каждой из абстракций элемент данных `Visible` и операцию `IsVisible` – проверку, видна ли фигура на экране.

В итоге получаем:

1. *Точка*:
   - данные: (x, y, Visible);
   - операции: Show, Hide, MoveTo, IsVisible.
2. *Квадрат*:
   - данные: (x, y, a, Visible);
   - операции: Show, Hide, S, MoveTo, IsVisible.
3. *Круг*:
   - данные: (x, y, r, Visible);
   - операции: Show, Hide, S, MoveTo, IsVisible.
4. *Треугольник*:
   - данные: (x1, y1, x2, y2, x3, y3, Visible);
   - операции: Show, Hide, S, MoveTo, IsVisible.
5. *Прямоугольник*:
   - данные: (x1, y1, a, b, Visible);
   - операции: Show, Hide, S, MoveTo, IsVisible.
6. *Координатная плоскость*:
   - данные: (набор фигур, Visible);
   - операции: Show, Hide, S, IsVisible.

То, что мы получили, “в первом чтении” можно считать *результатом объектной декомпозиции*. Почему “в первом чтении”? Потому, что мы не выделили связи между обнаруженными абстракциями. Об этом несколько позже.

Немного терминологии

Прежде чем двигаться дальше, поговорим о терминах. Авторы книги надеются, что наш читатель будет активно изучать литературу по программированию. Без знания стандартной терминологии в ходе этого процесса не обойтись.

В прошлом разделе мы узнали, что процесс выделения основных абстракций в предметной области называется *объектной декомпозицией*. Заметим при этом, что сами выделенные абстракции называются *классами*, а их экземпляры с конкретными данными – *объектами*. Так, для приведенного на рисунке примера координатной плоскости мы имеем 6 классов (точка, квадрат, круг, треугольник, прямоугольник, координатная плоскость) и 12 объектов (2 точки, 2 квадрата, 2 круга, 2 треугольника, 3 прямоугольника и 1 координатная плоскость).

Теперь вкратце остановимся на используемой в литературе терминологии, касающейся разных частей объектного подхода. В западной литературе принятым является следующее представление:

Объектный подход = ОО Анализ + ОО Проектирование + ОО Разработка.

Объектно-ориентированный анализ (**OOA** – object-oriented analysis) – это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области [5].

Объектно-ориентированное проектирование (**OOD** – object-oriented design) – это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы [5].

Объектно-ориентированное программирование (**OOP** – object-oriented programming) – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования [5].

Как видим, собственно программирование начинается на третьем этапе, а технология регламентирует также и предшествующие программированию действия. Обратим внимание на небольшую терминологическую путаницу: в отличие от западной литературы в русскоязычной под термином *объектно-ориентированное программирование* (**ООП**), как правило, понимают все три составляющих объектного подхода. В дальнейшем изложении и мы не будем отступать от этой традиции.

Основные идеи объектного подхода

Итак, выше мы познакомились с примером объектной декомпозиции в простейшей задаче о фигурах на координатной плоскости. На том же примере мы постараемся рассмотреть основные идеи объектного подхода.

Прежде всего, необходимо понять, что хорошего в новом для нас способе анализа предметной области. Основной плюс состоит в том, что вместо “разрозненных” данных и алгоритмов теперь мы имеем в качестве результата набор абстракций (классов) с их данными и операциями (алгоритмами обработки этих данных). То есть предметная область предстает перед нами как совокупность классов и объектов, “внутри” которых находятся данные и алгоритмы.

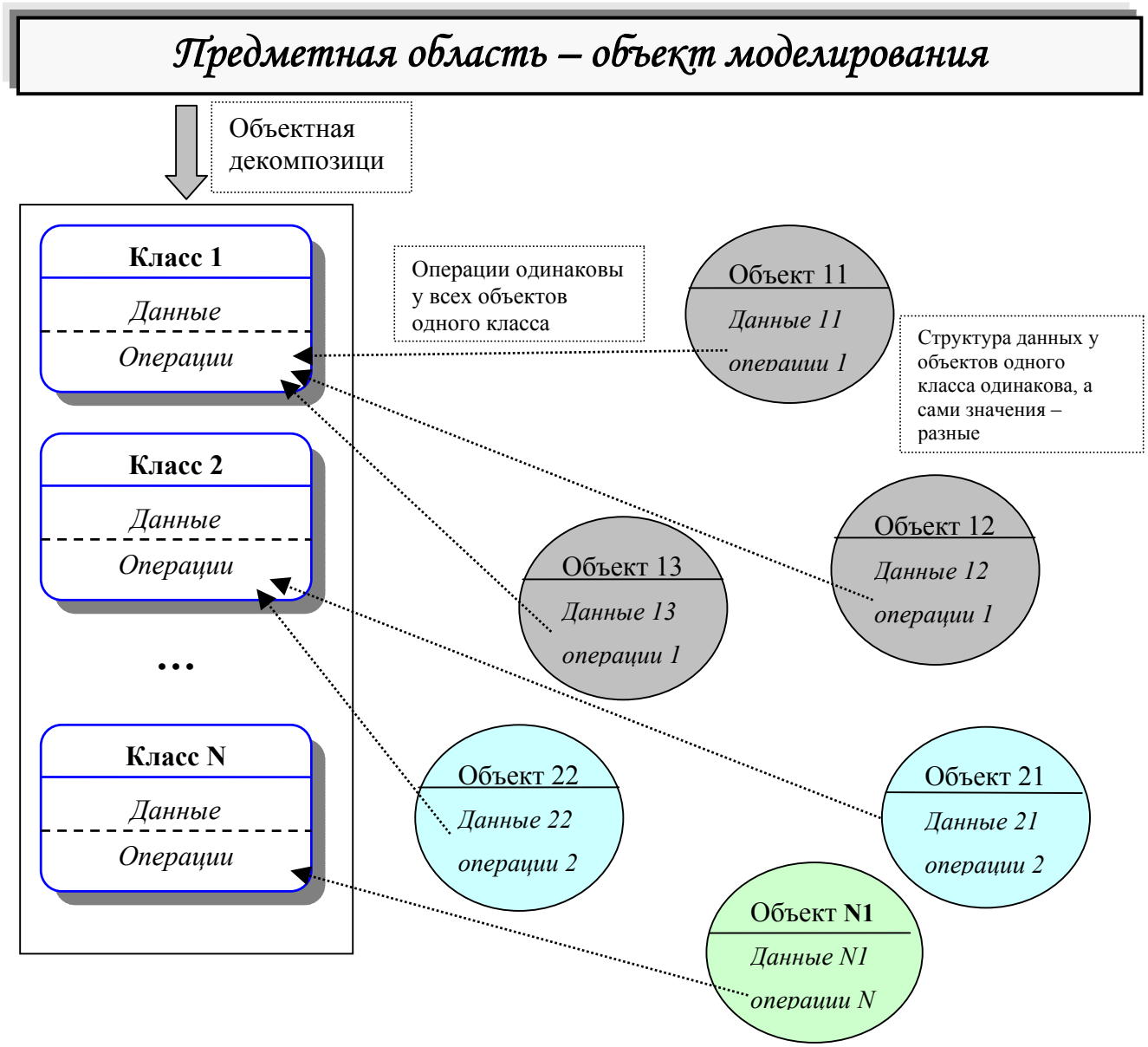


Рис. 10.5. Классы и объекты

Традиционно считается, что объектный подход базируется на трех основных понятиях: инкапсуляция, наследование и полиморфизм. Западные аналитики (см., например, в [5, 36, 37]) выделяют существенно больше составляющих объектного подхода, но мы ограничимся тремя перечисленными, считая их основными. Далее мы рассмотрим каждое из них более подробно.

Инкапсуляция

Под инкапсуляцией[1] понимаются следующие два тезиса:

1. Объединение данных и операций их обработки в рамках одной синтаксической структуры языка программирования.

[1] от английского “to encapsulate” – заключать в капсулу

2. Наличие механизма скрытия данных.

Первый тезис иллюстрирует важнейшее преимущество объектного подхода. Одна из основных проблем разработки программного обеспечения в рамках технологии модульного программирования состоит в том, что невозможно понять, какие алгоритмы обрабатывают те или иные данные. Синтаксически это никак не видно, информация об этом существует лишь в голове программиста то не долгое время, пока он пишет соответствующий фрагмент программы. Через несколько месяцев, чтобы понять, как связаны данные и алгоритмы, необходимо будет предпринять титанические усилия. В ООП мы вместо разрозненных алгоритмов и данных имеем классы, глядя на объявления которых можно сделать однозначный вывод о том, к каким данным какие операции в нашей модели предметной области применяются. Посмотрим на фрагмент объявления класса "Круг" (полностью правила объявления классов мы узнаем чуть позже):

```
TCircle = class
public
  x, y:    Real;    { Координаты центра }
  r:       Real;    { Радиус            }
  Visible: Boolean; { Признак видимости }

  function GetS: Real; { Расчет площади круга }
end;
```

Совершенно ясно, что конструкция **class** объединила данные и операции воедино. Представленное объявление само "говорит", что функция `GetS` обрабатывает данные класса `TCircle`. Какие именно? Для ответа на этот вопрос достаточно посмотреть на реализацию функции:

```
function TCircle.GetS: Real; { Расчет площади круга }
begin
  Result := PI * r * r;
end;
```

Итак, функция использует радиус и константу `PI`. Заметим также, что у функции отсутствуют аргументы – нужные ей данные она "берет" из класса, которому принадлежит.

Теперь перейдем ко второму тезису инкапсуляции. Что такое скрытие данных? В обычных модульных программах мы часто сталкиваемся с такой проблемой: как бы сделать так, чтобы наши коллеги своими действиями не смогли испортить результаты нашего труда. Не стоит думать при этом, что коллеги жаждут нам навредить. Понятно, что навредив, они отодвинут сроки реализации проекта, чем добавят в том числе и себе проблем и головной боли. Но! Вы будете удивлены, узнав, насколько часто они способны что-то испортить неумышленно (конечно, это не про Вас! Известно, что 99% программистов пишут код лучше, чем их коллеги).

Рассмотрим простой пример. Пусть Вы написали некоторую библиотеку, реализовав там необходимые структуры для хранения данных и набор функций для их обработки. После этого Вы передали библиотеку на использование другим программистам. В составленной Вами инструкции черным по белому написано: "Не использовать прямые обращения к структурам данных! Использовать только специальные функции". Многие ли последуют Вашим рекомендациям? Знайте, что как минимум половина программистов вообще не будет читать Вашу инструкцию. Они сначала попробуют разобраться с библиотекой "методом научного тыка" (да и плох тот программист, который не попробует…). У них все заработает, и они успокоятся. Вы спросите, в чем проблема? Проблема в том, что когда Вы захотите переписать структуры данных в библиотеке и поставить ее обновленную версию коллегам, у тех все перестанет работать, и они извергнут на Вас жуткие проклятья, искренне полагая, что они правы. Ведь у них же раньше все работало! И на Ваши оправдания, что на странице 347 инструкции было написано,

что прямые обращения к структурам хранения запрещены (и это правильно, иначе Вы не сможете развивать и оптимизировать свою библиотеку) никто не обратит внимания. Что с этим делать? Необходимы синтаксические механизмы скрытия данных и алгоритмов от несанкционированного использования. Такие механизмы – секции **private** (скрытые) и **public** (открытые) в классах языка Object Pascal.

Так, в следующем примере не удастся получить доступ извне к полю `Visible`, для этого предоставляется специальная функция `IsVisible`, которая позволяет узнать значение элемента данных `Visible`, но не изменить его. Заметим, что соображения за и против скрытия данных могут быть разными. В данном примере разумность скрытия `Visible` вытекает из того, что в противном случае пользователь класса (коллега-программист) получит возможность менять признак `Visible` как угодно, в то время как круг может быть виден на координатной плоскости, а может быть нет. Возможность свободного изменения параметра `Visible` может привести к рассогласованию внутренних данных объекта и его внешнего представления, чего, конечно, быть не должно.

```
TCircle = class
private
  Visible: Boolean; { Признак видимости }
public
  x, y: Real;       { Координаты центра }
  r:    Real;       { Радиус            }

  function GetS: Real;          { Расчет площади круга       }
  function IsVisible: Boolean; { Возврат признака видимости }
end;
```

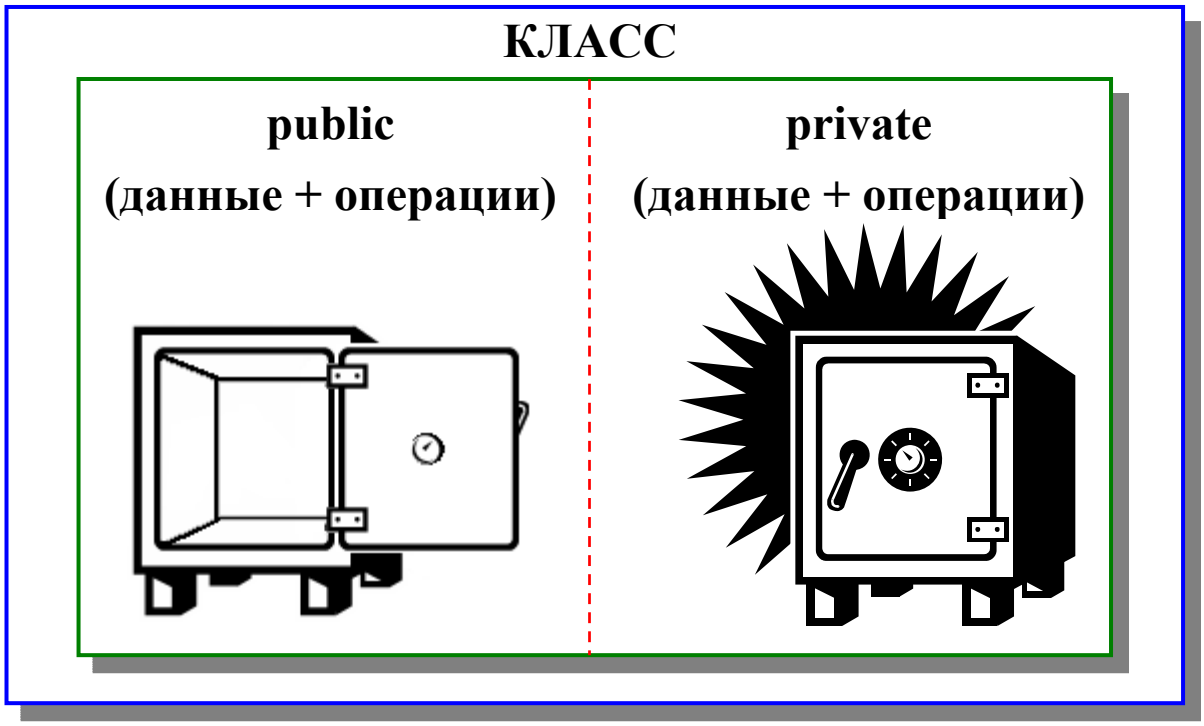


Рис. 10.6. Инкапсуляция – совместное проектирование данных и операций + разграничение доступа

Итак, инкапсуляция позволяет скрывать детали внутренней реализации, а значит, дает возможность проводить модификацию, не затрагивая код, который использует наши классы.

Агрегация и наследование

Одна из наиболее актуальных проблем в отрасли разработки программного обеспечения – многократная реализация одного и того же, так называемое “программирование в корзину”. Удивительно, как много людей в одно и то же время программируют сходные реализации одинаковых алгоритмов в идентичных задачах. Хуже того, один и тот же программист в своей практике неоднократно переписывает код, который уже был реализован ранее в одной из старых программ. В результате работа приобретает характер многократного бессмысленного преодоления одних и тех же препятствий. Предпосылки к возникновению подобной ситуации состоят еще и в том, что “выдрать” некие фрагменты кода из старой программы и перенести их в новую подчас является достаточно сложной задачей. Нередко оказывается проще переписать код заново, что и делается.

Одним из ключевых моментов в ООП является *возможность создания нового программного кода (реализующего новую функциональность) при помощи уже написанного ранее*. Преимущество такого подхода очевидно – ранее написанный код обычно является отлаженным и работоспособным. Достаточно важно и то, что мы можем использовать и новую, и старую редакции кода в одной и той же программе. Это обеспечивает существенное упрощение модификации и отладки. Посмотрим, как это делается.

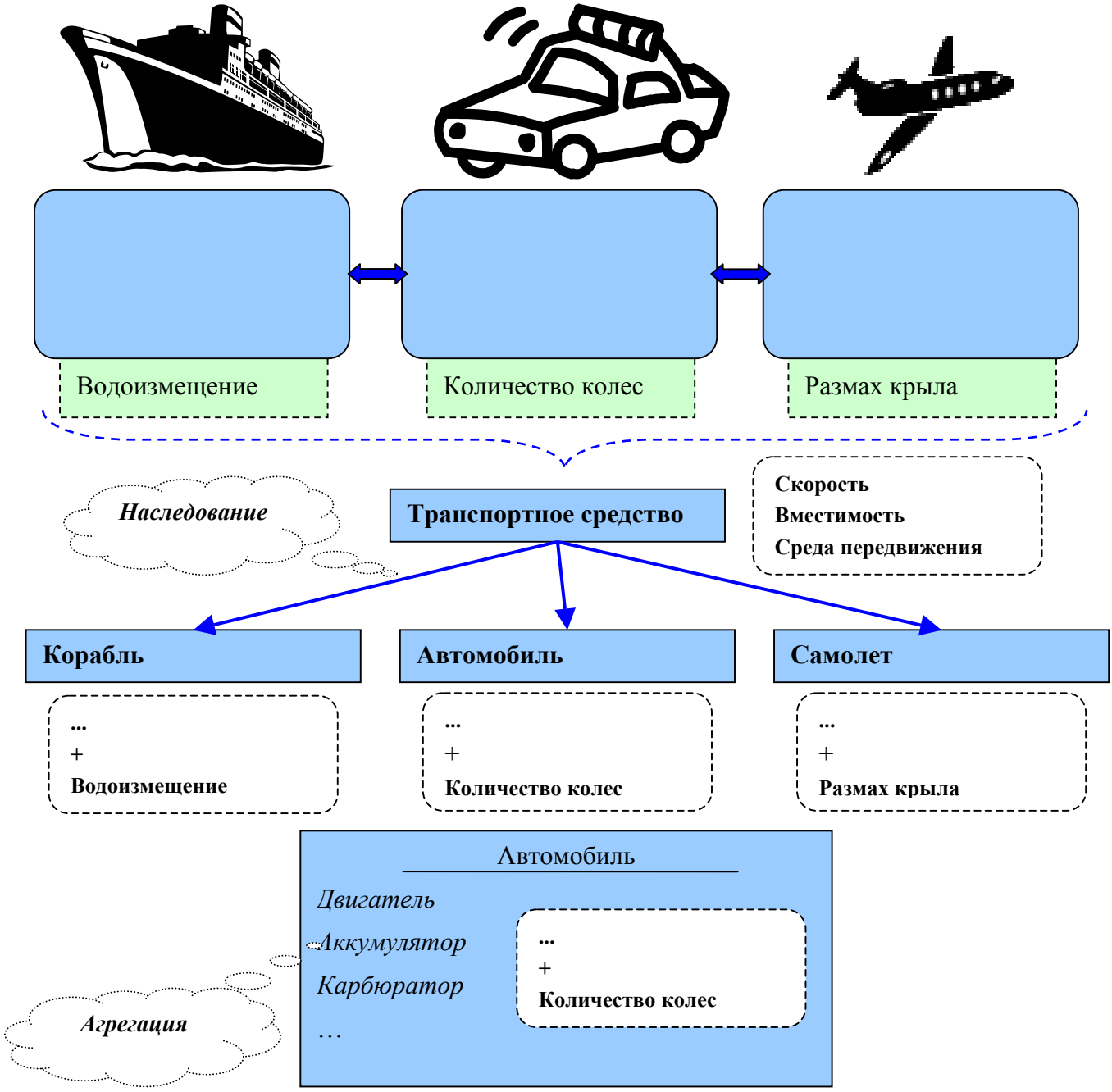


Рис. 10.7. Агрегация и наследование

В предыдущем пункте мы научились не только выделять абстракции в предметной области, но и слегка коснулись вопроса о скрытии деталей реализации. Применение этих составляющих объектного подхода существенно упрощает понимание задачи. Но и этого часто оказывается недостаточно. В таких случаях на помощь приходит установление связей между классами. Принципиально, можно выделить по крайней мере 2 типа отношений: “*это есть*” и “*быть частью*” [5]. Для их описания применяются два термина: *наследование* и *агрегация*.

Посмотрим на примеры.

“Это есть” (наследование)	“Быть частью” (агрегация)
КРУГ – это такая **ТОЧКА**, у которой есть все, что есть у точки, + добавился радиус и переписан ряд операций	**КРУГ** = **ТОЧКА** + радиус + некоторые новые операции
ТРЕУГОЛЬНИК – это такая точка, у которой есть еще 2 точки и переписан ряд операций	**ТРЕУГОЛЬНИК** = **ТОЧКА** + **ТОЧКА** + **ТОЧКА** + некоторые новые операции
ЖИВОТНОЕ – умеет перемещаться. **ПТИЦА** – **ЖИВОТНОЕ**, которое умеет летать. **СОЛОВЕЙ** – **ПТИЦА**, которая умеет не только летать, но еще и петь…	
	ПК = Монитор + Системный блок + Периферия Системный блок = Материнская плата (с устройствами) + Блок питания + Провода + … Материнская плата (с устройствами) = Процессор + Память + Видеоадаптер + … + Чипсет Видеоадаптер = …

Таблица 10.1 Примеры агрегации и наследования

Налицо два способа образования иерархий – *агрегация* и *наследование*. Объединение абстракций в иерархии позволяет рассматривать задачу последовательно на разных уровнях. Спускаясь вниз по дереву, мы увеличиваем степень детализации. Заметим, что кто-то понимает задачу на самом верхнем уровне и может лишь выкидывая половину компьютера устранять неисправность, кто-то на следующем уровне (может менять блоки), ну а кто-то для примера с ПК разбирается на уровне радиодеталей и может с паяльником в руках заменять вышедшие из строя запчасти. Чуть позже мы посмотрим, как абстракция и иерархия реализуются в синтаксисе языка Object Pascal.

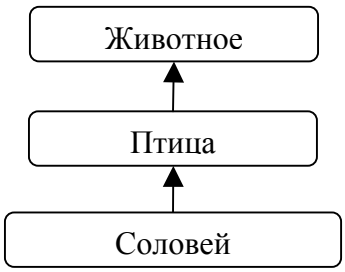


Рис. 10.8. Схема (иерархия) наследования для животных

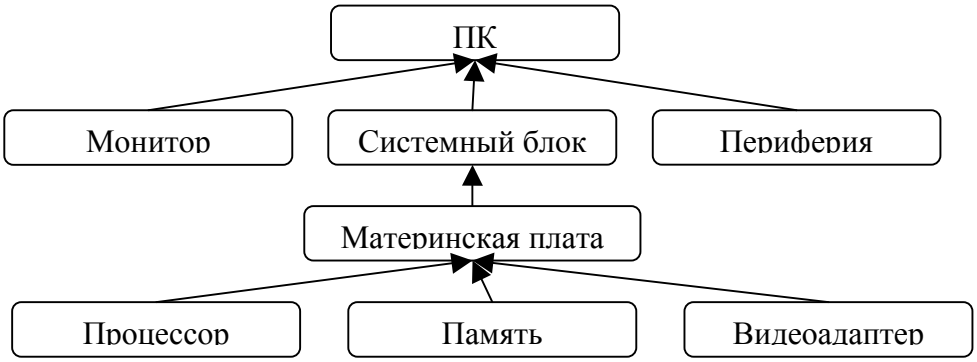


Рис. 10.9. Схема (иерархия) наследования для ПК

Полиморфизм

Слово полиморфизм происходит от греческого polymorphos – многообразный. Примеры полиморфного поведения нам уже известны. Так, обратим внимание на операцию “+” в языке Pascal. Очевидно, что для целых чисел, вещественных чисел и, наконец, для строк она выполняет совершенно разные действия, сохраняя семантику (смысл) использования и обозначение. Таким образом, “+” является полиморфной операцией.

Возникает следующий вопрос: полиморфизм – это очень удобно (подумайте, каким кошмаром для программистов обернулась бы необходимость по-разному обозначать “+” для разных типов данных), но он существует для встроенных типов данных. А можно ли реализовать полиморфное поведение во вновь создаваемых типах данных (классах)? Одно из весьма существенных достоинств объектного подхода состоит в том, что он (и его реализация в языке Object Pascal) позволяет реализовывать полиморфное поведение в классах!

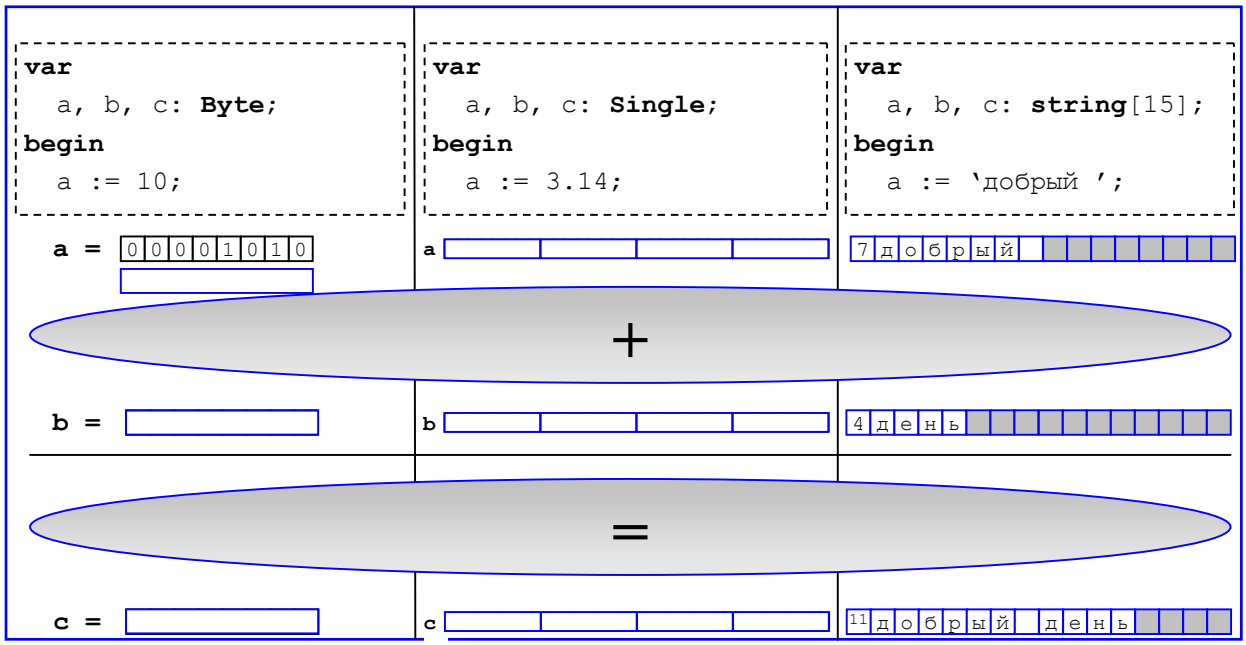


Рис. 10.10. Полиморфная операция “+” реализуется по-разному для разных типов данных

Резюме

В данном разделе мы сделали много анонсов, объясняя достоинства объектного подхода “на пальцах”. Самое время перейти к рассказу о его реализации в языке Object Pascal.

Объектно-ориентированное программирование на языке Object Pascal

Вспоминая о записях

Для того чтобы наилучшим образом понять суть вопросов, связанных с синтаксисом ООП в языке Object Pascal, вновь обратим свой взгляд на записи (**record**). В частности, давайте опишем в виде записи новый тип данных “круг”.

```
type
  TCircleRec = record
    X, Y: Real;       { Координаты центра }
    R: Real;          { Радиус            }
    Visible: Boolean; { Признак видимости }
  end;
```

Теперь, создавая переменные типа `TCircleRec`, мы получаем возможность хранить данные кругов. А как быть с операциями? Операции придется написать отдельно. Рассмотрим некоторые из них для примера. Дополним пример записью “квадрат”.

Будем “рисовать фигуры” при помощи условно существующих процедур рисования `Point`, `Circle` и т.д. Описания способов рисования в окне можно найти в справке по библиотеке VCL (см. методы класса `TCanvas`).

```
{ ========================================================================== }
{ Пример 10.1 }
{ Представление фигур на плоскости при помощи записей }
  type
    TSquareRec = record
      X, Y: Real;       { Координаты левого нижнего угла }
      A: Real;          { Сторона                         }
      Visible: Boolean; { Признак видимости               }
    end;

    TCircleRec = record
      X, Y: Real;       { Координаты центра }
      R: Real;          { Радиус            }
      Visible: Boolean; { Признак видимости }
    end;

  var
    cr: TCircleRec;
    sq: TSquareRec;
```

```
  procedure ShowSquare(_sq: TSquareRec); { Показать квадрат }
  begin
    if not _sq.Visible then
    begin
      _sq.Visible := True;
      Square(_sq.X, _sq.Y, _sq.A);
    end;
  end;

  procedure ShowCircle(_cr: TCircleRec); { Показать круг }
  begin
    if not _cr.Visible then
    begin
      _cr.Visible := true;
      Circle(_cr.X, _cr.Y, _cr.R);
    end;
  end;

  function SSquare(_sq: TSquareRec): Real; { Площадь квадрата }
  begin
    Result := _sq.A * _sq.A;
  end;

  function SCircle(_cr: TCircleRec): Real; { Площадь круга }
  begin
    Result := PI * _cr.R * _cr.R;
  end;

  { Основная программа }
  begin
    with cr do
    begin
      X := 100; Y := 100; R := 20;
    end;
    with sq do
    begin
      X := 200; Y := 100; A := 50;
    end;
    ShowCircle(cr);
    ShowSquare(sq);
    WriteLn('Площадь квадрата = ', SSquare(sq));
    WriteLn('Площадь круга = ', SCircle(cr));
  end.
{ ========================================================================== }
```

Проанализировав пример, можно сделать следующие выводы:

1. Операции “оторваны” от данных. Нет никакой возможности привязать подпрограммы к записям.
2. Реализуя аналогичные по смыслу операции для круга и квадрата, мы вынуждены каждый раз придумывать новые имена подпрограмм (`ShowCircle`, `ShowSquare`…). А что если типов фигур будет 10, плюс у каждой по 15 аналогичных операций. 10 x 15 = 150 подпрограмм. Не много ли?
3. Наличие разных имен не позволит нам записать, к примеру, показ всех фигур в цикле:

```
for i := 1 to N do
  Show(Figures[i]);
```

Как нетрудно понять, вызвано это тем, что у каждого типа фигур операция “показать” называется по-разному. Более того, нам даже не удастся объявить “массив фигур”, из-за того, что все они разнотипные. Эту проблему можно решить, объявив массив указателей на фигуры. Но как компилятор должен разобраться, как именно надо показывать фигуру `Figures[i]^`, если реальный тип содержимого указателя определится лишь на этапе работы программы?

4. Заметим, что любой программист может написать `cr.Visible := false;` в то время, как на самом деле фигура видима на экране. В записях нет никаких средств, чтобы защитить поле `Visible` от несанкционированного доступа.

Это лишь вершина айсберга, те немногие, но очень существенные проблемы, которые видны даже из этого тривиального примера. Существуют и другие трудности, практически непреодолимые без использования ООП. Посмотрим, как решает эти и некоторые другие проблемы объектная модель языка программирования Object Pascal.

Объявление класса. Поля и методы

Для лучшего понимания материала обратимся к самому началу применения объектного подхода – к анализу предметной области. Пусть наша предметная область – уже знакомая нам координатная плоскость (см. раздел “Алгоритмическая и объектная декомпозиции”). В результате анализа были выделены 6 абстракций, в том числе “Квадрат” со следующим описанием:

Квадрат:

- данные: (x, y, a, Visible);
- операции: Show, Hide, S, MoveTo, IsVisible.

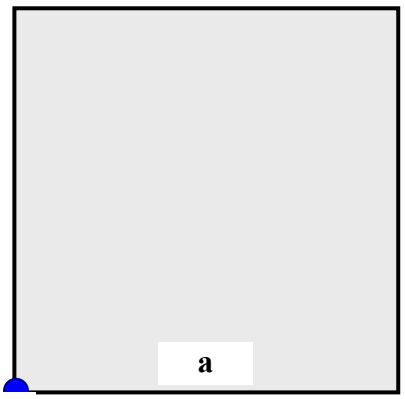


Рис. 10.11. Квадрат на плоскости

Важное замечание: считаем стороны квадрата параллельными осям координат.

Первая часть описания квадрата – результат проектирования данных.

Вторая часть (вместе с алгоритмами осуществления указанных операций, в данном примере они тривиальны, но так бывает не всегда) – результат проектирования операций.

В итоге перечень выполненных нами действий выглядит так:

- выделена абстракция, существенная для предметной области;
- определено, какими данными характеризуется каждая такая абстракция;
- определено, какие операции можно выполнять над такими абстракциями.

Эти положения задают две основных составляющих типа данных – *типа данных* “Квадрат”.

В терминах ООП мы получили характеристику класса. Таким образом, класс – абстрактный тип данных, средство моделирования объектов предметной области.

Заметим, что с языком программирования Object Pascal связаны две объектные модели – новая, появившаяся в Delphi, и старая, существовавшая еще в Borland Pascal для операционной системы MS DOS. В старых книгах Вы еще можете увидеть объявления класса в виде:

```
type
  TMyClass = object
  ...
  end;
```

Эта модель морально устарела, хотя и поддерживается до сих пор в целях обратной совместимости. Новая объектная модель является существенно более мощной. В данной книге мы рассматриваем новую модель, которую идентифицирует конструкция **class**.

Поскольку класс – тип данных, необходимо уметь его объявлять. Делается это как обычно в секции **type**. Простейший синтаксис объявления класса (в ходе изучения материала мы будем его уточнять) выглядит так:

```
type
  TMyClass = class
    <данные>
    <операции>
  end;
```

Описание класса “Квадрат” может в первом приближении выглядеть так:

```
type
  TSquare = class
    {----------  Данные  -------------}
    x, y:    Real; { Координаты левого нижнего угла }
    a:       Real; { Сторона квадрата               }
    Visible: Real; { Признак видимости              }
    {----------  Операции  -------------}
    procedure Show;                  { Показать              }
    procedure Hide;                  { Скрыть                }
    function  IsVisible;             { Вернуть Visible       }
    function  S: Real;               { Вычислить площадь     }
    procedure MoveTo(dx, dy: Real); { Переместить на dx, dy }
  end; { TSquare }
```

Таким образом, мы определили, какими данными будут обладать переменные класса “Квадрат”, а также какими операциями их можно обрабатывать. Немного терминологии:

1. `TSquare` – класс, переменные типа `TSquare` – объекты.
2. Данные класса обычно называются *полями* (в англоязычной литературе *member-fields*).
3. Операции класса обычно называются *методами* (в англоязычной литературе *member-functions*).

Итак, *класс = поля + методы*.

Заметим, что приведенный выше программный код представляет собой именно объявление класса и не содержит реализации методов. Об этом позже.

Инкапсуляция в действии. Спецификаторы доступа

Рассматривая ранее принцип инкапсуляции, мы указали, что он включает в себя два момента: объединение данных и операций в рамках одной синтаксической структуры и разграничение доступа. Первый момент виден из приведенного объявления класса `TSquare`. А как быть со вторым?

Уточним формат объявления класса:

```
type
  TMyClass = class
  private
    <поля>
    <методы>
  protected
    <поля>
    <методы>
  public
    <поля>
    <методы>
  published
    <поля>
    <методы>
  end;
```

Как мы теперь видим, объявление класса содержит 4 секции: **public**, **private**, **protected** и **published**.

> Заметим, что в целях обратной совместимости поддерживается еще и пятая секция – **automated**, но в последних версиях языка ее использовать не рекомендуется, поэтому мы и не будем говорить о ней в рамках данной книги.

Ключевые слова **public**, **private**, **protected** и **published** называются *спецификаторами доступа* и задают правила, по которым компилятор решает, предоставить программисту в данном месте программы доступ к полю (методу) или нет. Правила выглядят следующим образом:

Спецификатор доступа	Правила
public	Открытые поля и методы. Доступны из любого места программы.
private	Скрытые поля и методы. Доступны только в том модуле, в котором объявлен класс. Из другого модуля нельзя вызвать скрытый метод или изменить, прочитать скрытое поле (на самом деле, если нельзя, но очень хочется, то можно, но мы не будем рассматривать в книге приемы, вредные для коллективного программирования, в том числе и этот).
protected	Защищенные поля и методы. Доступны в рамках модуля, в котором объявлен класс, а также в потомках данного класса (см. Наследование).
published	Опубликованные поля и методы. Используются для поддержки работы технологии визуального программирования (библиотека визуальных компонентов VCL). В данной книге не рассматривается.

Таблица 10.2 Спецификаторы доступа

Таким образом, мы можем управлять доступом к полям и методам, размещая их в разные секции в соответствии со смыслом их использования. Наша общая рекомендация по размещению по секциям выглядит так: объявляйте в секции **private** те поля и методы, которые не предназначены для использования извне и относятся к деталям внутренней реализации Вашего класса. Все остальное размещайте в секцию **public** (использование секции **protected** будет объяснено в разделе "Наследование и полиморфизм. Виртуальные методы и позднее связывание").

Вернемся к примеру о квадрате. Ранее было замечено, что предоставление доступа к полю-признаку `Visible` потенциально опасно и может привести к рассогласованному состоянию предметной области и ее объектного описания в памяти компьютера. Таким образом, необходимо исключить изменение поля `Visible` вне класса. Скорректируем объявление:

```
type
  TSquare = class
  private
    {----------  Данные  -------------}
    Visible: Boolean; { Признак видимости }
  public
    {----------  Данные  -------------}
    x, y: Real; { Координаты левого нижнего угла }
    a:    Real; { Сторона квадрата               }

    {----------  Операции  -----------}
    procedure Show;                  { Показать              }
    procedure Hide;                  { Скрыть                }
    function  IsVisible: Boolean;    { Вернуть Visible       }
    function  S: Real;               { Вычислить площадь     }
    procedure MoveTo(dx, dy: Real);  { Переместить на dx, dy }
  end; { TSquare }
```

Теперь ситуация выглядит так: поле `Visible` нельзя изменить извне, но можно узнать его значение посредством вызова функции `IsVisible`.

Реализация методов класса

Посмотрев, как в первом приближении выглядит объявление класса, самое время узнать, где и как пишется реализация методов. Общий вид заголовка метода при его реализации выглядит так:

```
procedure <Имя класса>.<Имя метода>(<список параметров>);
```

или

```
function <Имя класса>.<Имя метода>(<список параметров>): <тип результата>;
```

Таким образом, имя метода "расширяется" именем класса. Действительно, многие классы должны иметь метод `Show`, как же компилятор разберется, чей именно `Show` мы сейчас реализуем? При помощи расширенного имени `<Имя класса>.<Имя метода>` – вот и первое проявление полиморфизма. Более не надо писать `ShowSquare`, достаточно просто `Show`.

Теперь реализуем несколько методов:

```
procedure TSquare.Show; { Показать квадрат }
begin
```

```
    if not Visible then
    begin
      Visible := true;
      Square(x, y, a);
    end;
  end;

  function TSquare.S: Real; { Площадь квадрата }
  begin
    Result := a * a;
  end;
```

Следующий момент связан с тем, где это все писать. Вообще, класс – просто тип данных, который объявляется в секции **type**, а методы класса – процедуры и функции, которые реализуются в том же модуле, как обычно. Однако, чаще всего один или несколько классов выносятся в отдельный модуль. Поступим так и мы, создав модуль `UFigures` для описания классов для представления фигур.

```
{ ========================================================================== }
{ Классы - фигуры на плоскости }
Unit UFigures;
interface

type
  TSquare = class
  private
    {----------  Данные  -------------}
    Visible: Boolean; { Признак видимости }
  public
    {----------  Данные  -------------}
    x, y: Real; { Координаты левого нижнего угла }
    a:    Real; { Сторона квадрата               }
    {----------  Операции  -----------}
    procedure Show;                  { Показать              }
    procedure Hide;                  { Скрыть                }
    function  IsVisible: Boolean;    { Вернуть Visible       }
    function  S: Real;               { Вычислить площадь     }
    procedure MoveTo(dx, dy: Real); { Переместить на dx, dy }
  end; { TSquare }

implementation

procedure TSquare.Show;  { Показать квадрат }
begin
  if not Visible then
  begin
    Visible := true;
    Square(x, y, a);
  end;
end;

function TSquare.S: Real; { Площадь квадрата }
begin
  Result := a * a;
end;
```

```
...
begin
end.
{ ========================================================================== }
```

Свойства

Еще один важный элемент объектной модели языка Object Pascal – *свойства* (в англоязычной литературе *properties*). Заметим, что свойства – существенное новшество языка Pascal по сравнению с языком программирования C++, в котором нет ничего подобного (хотя, разумеется, есть много других достоинств). Концепция свойств оказалось настолько популярной, что проникла в новый язык программирования, разработанный специально для платформы .Net – C#. Рассмотрим смысл и синтаксис данного понятия на примере.

Заметим, границы нашего квадрата при рисовании должны отображаться некоторым цветом. Для отражения этого факта в программном коде добавим в объявление типа `TSquare` открытое поле `Color`.

```
type
  TSquare = class
  private
    {----------  Данные  -------------}
    Visible:  Boolean;  { Признак видимости }
  public
    {----------  Данные  -------------}
    x, y:  Real;      { Координаты левого нижнего угла }
    a:     Real;      { Сторона квадрата               }
    Color: Cardinal; { Цвет границ                    }
    {----------  Операции  -----------}
    ...
  end; { TSquare }
```

Подумаем, как все это будет работать. Допустим, что воображаемая функция `Square` рисует квадрат на плоскости, получая координаты его левого нижнего угла и цвет границ. Тогда реализация метода `Show` могла бы выглядеть так:

```
procedure TSquare.Show; { Показать квадрат }
begin
  if not Visible then
  begin
    Visible := true;
    Square(x, y, a, Color);
  end;
end;
```

Суть проблемы состоит в следующем: что если пользователь класса (коллега-программист) изменил значение `Color` на другое? Было бы логично, чтобы квадрат автоматически был перерисован другим цветом. Иначе придется писать в программе так:

```
MySquare.Color := NewColor;
MySquare.Hide;
MySquare.Show;
```

Слишком много для такой тривиальной операции, не правда ли? К тому же, нет гарантии, что никто не забудет написать все 3 строки, а то получится как в известном анекдоте, где один копал ямы, а другой их сразу же закапывал. Исследование показало, что был еще и третий, он должен был сажать деревья, но не пришел.

Эта проблема решается посредством использования свойств. Синтаксис объявления свойства выглядит так:

```
property <Имя свойства>: <Тип данных> read <поле или метод>
  write <поле или метод>;
```

В отличие от поля, свойство не хранит данных! Это просто механизм для доступа. Наиболее популярный вариант использования – создание скрытых полей и открытых свойств для доступа к ним.

В нашем примере мы можем поступить так:

```
TSquare = class
private
  {----------  Данные  -------------}
  FVisible: Boolean;  { Признак видимости              }
  FColor:   Cardinal; { Цвет границ                    }
  Fx, Fy:   Real;     { Координаты левого нижнего угла }
  Fa:       Real;     { Сторона квадрата               }

  procedure SetColor(const Value: Cardinal);
public
  {----------  Свойства  -------------}
  property x: Real read Fx write Fx;
  property y: Real read Fy write Fy;
  property a: Real read Fa write Fa;

  property Color: Cardinal read FColor write SetColor;
  property Visible: Boolean read FVisible;
  {----------  Операции  -------------}
  procedure Show;                  { Показать              }
  procedure Hide;                  { Скрыть                }
  function  S: Real;               { Вычислить площадь     }
  procedure MoveTo(dx, dy: Real); { Переместить на dx, dy }
end; { TSquare }
```

Посмотрим, что мы сделали. Во-первых, мы скрыли все данные. Во-вторых, мы создали свойства `x`, `y`, `a`, которые после ключевых слов **`read`** и **`write`** содержат названия полей `Fx`, `Fy`, `Fa`. Это означает, что запись `MyObject.x := 1` или `q := MyObject.x` будет означать доступ к полю `Fx`. На этих примерах особой выгоды не видно. Обратимся к более содержательным случаям. Так, для свойства `Visible` можно заметить отсутствие секции **`write`**. Это означает, что хотя при чтении свойства мы берем значение из поля `FVisible`, запись является запрещенной. Метод `IsVisible` стал более не нужным и был исключен из объявления класса (предложение "**`if`** `Visible` **`then`**" теперь полностью его заменяет).

Отдельно рассмотрим свойство `Color`. В секции **`read`** мы имеем ссылку на поле `FColor`, а в секции **`write`** – упоминание метода `SetColor`. Метод `SetColor` получит управление при попытке изменить цвет посредством свойства `Color`. Параметр метода – новое значение цвета. Реализуем процедуру `SetColor` так:

```
procedure TSquare.SetColor(const Value: Cardinal);
begin
```

```
    Hide;
    FColor := Value;
    Show;
  end;
```

Теперь при попытке изменить цвет моментально будет перерисован квадрат, что и требовалось получить.

Итак, мы можем создавать свойства, делать их свойствами только для чтения (часто применяется) или только для записи (крайне редко), связывать их с полями напрямую или посредством специальных методов.

Используя механизм свойств, мы можем предоставить доступ на запись в свойстве `Visible`, предусмотрев специальный метод, который будет отображать фигуру, вызывая ее метод `Show`.

```
  TSquare = class
  private
    {----------  Данные  -------------}
    FVisible:  Boolean; { Признак видимости }
    ...
    procedure SetVisible(const Value: Boolean);
  public
    {----------  Свойства  -------------}
    ...
    property Visible: Boolean read FVisible write SetVisible default False;
    {----------  Операции  -------------}
    ...
  end; { TSquare }

  procedure TSquare.SetVisible(const Value: Boolean);
  begin
    if FVisible <> Value then
    begin
      FVisible := Value;
      if FVisible then
        Show
      else
        Hide;
    end;
  end;
```

Обратим внимание на ключевое слово **`default`**, которое в данном случае позволяет задать значение по умолчанию. Теперь, свойство при создании объекта будет установлено в **`false`**.

Следующий важный тип свойства – индексированное или массивное свойство.

Пусть некоторый класс содержит среди данных массив.

```
  TVector = class
  private
    FData: array[1..100] of Real;
    function GetData(Index: Integer): Real;
    procedure SetData(Index: Integer; MyData: Real);
    ...
  public
    property Data[Index: Integer]: Real read GetData write SetData;
    ...
  end; { TVector }
```

```
function TVector.GetData(Index: Integer): Real;
begin
  Result := FData[Index];
end;

procedure TVector.SetData(Index: Integer; MyData: Real);
begin
  FData[Index] := MyData;
end;
```

Теперь мы имеем возможность создать объект типа `TVector` (о том, как это делается, чуть позднее) и написать так:

```
v.Data[5] := 3.14;
```

При этом вызовется метод `SetData` с параметрами `Index = 5, MyData = 3.14`.

Заметим, что методы могут выполнять некоторые дополнительные действия. В принципе, массивное свойство – просто интерфейсный элемент, так оно может в действительности не быть связано с реальным массивом, но чаще всего дело обстоит так, как в рассмотренном примере.

Еще одно важное дополнение к рассмотренной функциональности состоит в том, что специально для массивных свойств ключевое слово **`default`** не только может быть применено, но и работает совершенно по-другому. Добавим ключевое слово **`default`** так:

```
property Data[Index: Integer]: Real read GetData write SetData; default;
```

Теперь свойство `Data` становится свойством по умолчанию, и мы можем написать

```
v[5] := 3.14
```

вместо

```
v[5].Data := 3.14.
```

Подводя итог, можно заключить, что свойства – мощный и удобный инструмент для разработки полноценных объектно-ориентированных программ.

Специальные виды методов. Конструкторы. Перегрузка конструкторов

Познакомившись с тем, как объявляются и реализуются методы в классах на языке Object Pascal, самое время изучить некоторые специальные виды методов. Сразу оговоримся, что их немало. Начнем с самого главного, с методов, без которых в реальности не существует ни один класс – с *конструкторов* и *деструкторов*.

Конструктор – специальный метод, который вызывается для создания переменных объектного типа, т.е. в нашей терминологии объектов. В чем основное предназначение данного метода, что отличает его от других? Прежде всего, вкладываемая в него смысловая нагрузка. Так, основные задачи конструктора можно сформулировать следующим образом:

- выделение памяти для хранения объекта;
- инициализация полей объекта начальными значениями;
- выделение памяти для специфических полей объекта (полей-объектов, полей указателей).

Все эти функции являются очень важными и заслуживают того, чтобы ради них ввели специальный вид метода.

Следующий вопрос состоит в том, как объявить конструктор. С точки зрения языка Object Pascal, конструктор есть подпрограмма, у которой вместо ключевого слова **procedure** присутствует ключевое слово **constructor**. Так, синтаксис объявления конструктора для класса `TSquare` может выглядеть так:

```
TSquare = class
private
  {----------  Данные  -------------}
  FVisible: Boolean;  { Признак видимости              }
  FColor:   Cardinal; { Цвет границ                    }
  Fx, Fy:   Real;     { Координаты левого нижнего угла }
  Fa:       Real;     { Сторона квадрата               }

  procedure SetColor(const Value: Cardinal);
public
...
  constructor Create;
end; { TSquare }
```

У Вас наверняка возник ряд вопросов. Сейчас мы сформулируем основные принципы, касающиеся объявления конструкторов, которые должны существенно прояснить ситуацию.

Для начала, обсудим вопрос об именовании конструкторов. Если у вас в классе объявлен один конструктор, настоятельно рекомендуется называть его `Create`. Конечно, Вы можете пренебречь этой рекомендацией, и все будет работать, но… Стиль есть стиль. С большими шансами коллеги не поймут, если в классе `TMySuperGreatClass` конструктор будет называться `MySuperGreatClassConstr`. Это непонимание выльется в то, что они откажутся писать этот ужас в своем коде. Вообще, здравый смысл подсказывает, что изобретение велосипедов хорошо лишь в тех областях, где а) нет велосипедов; и б) они вообще кому-то там нужны. Здесь нет ни а), ни б), поэтому мы все конструкторы по возможности будем именовать `Create`.

Ознакомившись для начала со стилистическим моментом написания программы, перейдем к существенно более сложным смысловым вещам. Так, рассмотрим, что за операции выполняет приведенный выше конструктор, если мы создадим для него “пустую” реализацию.

```
constructor TSquare.Create;
begin
end;
```

Оказывается, данная реализация при запуске выполняет целую тучу разных действий. Кто об этом позаботился, если мы не написали ни строки кода? Конечно, наш друг компилятор. Такая реализация обеспечивает следующие действия, выполняемые *автоматически* любым конструктором:

- Выделение памяти для полей объекта. В данном случае выделится память, необходимая для хранения полей `FVisible`, `FColor`, `Fx`, `Fy`, `Fa`.
- Инициализация полей нулевыми значениями. Стоп! В этом месте необходимо задержаться. Документация по системам программирования Borland Delphi действительно утверждает, что обнуление происходит. Это означает, что обычные поля инициализируются нулем, поля-указатели – значением **nil**, поля-объекты – нулевыми ссылками. Но! Ни в коем случае не рекомендуем Вам ставить работоспособность программы в зависимость от того, обнулил что-то компилятор сам или нет. Если для

программы важно гарантированное обнуление, выполняйте его самостоятельно. Тем самым Вы страхуете себя от того, что Ваша программа перестанет работать при переходе к другому компилятору.

- Создание в памяти довольно сложной структуры. Дело в том, что объект – это не только совокупность полей, но и информация о типе. Данный сложный вопрос будет рассмотрен нами чуть позже в разделе “Внутренняя структура объекта. Методы класса. Динамический контроль типов, операторы IS и AS”.

Все это означает, что для создания полей объекта нам не нужно писать каких-то сложных конструкций, компилятор все делает сам. А бывает ли необходимость все-таки что-то написать или всегда нужно оставлять пустую реализацию? Конечно, бывает. Сейчас мы рассмотрим несколько примеров. Первый случай, когда необходимо вмешаться в создание объектов класса, связан с желанием параметризовать процесс создания, т.е. позволить создавать объекты с инициализацией полей ненулевыми значениями. Так, для класса `TSquare` это может выглядеть следующим образом:

```
  TSquare = class
  private
    {----------  Данные  -------------}
    FVisible: Boolean;  { Признак видимости              }
    FColor:   Cardinal; { Цвет границ                    }
    Fx, Fy:   Real;     { Координаты левого нижнего угла }
    Fa:       Real;     { Сторона квадрата               }

    procedure SetColor(const Value: Cardinal);
  public
    ...
    constructor Create(ax, ay, aa: Real; aColor: Cardinal;
      aVisible: Boolean);
  end; { TSquare }

  constructor TSquare.Create(ax, ay, aa: Real; aColor: Cardinal;
    aVisible: Boolean);
  begin
    Fx := ax;
    Fy := ay;
    FColor := aColor;
    FVisible := False;
    if aVisible then
      Show;
  end;
```

Данный конструктор выполняет инициализацию полей переданными параметрами.

Второй случай вмешательства в действия компилятора – выделение памяти для полей объектов и полей указателей. Пусть мы создаем класс – динамический массив действительных чисел. Это может выглядеть так:

```
{ ========================================================================== }
{ Пример 10.2 }
{ Класс - динамический массив }
  TArrDouble = class
  private
    FDim: Integer;
    FValues: array of Double;
```

```
    procedure SetDim(const Value: Integer);
    function GetRValue(Index: Integer): Double;
    procedure SetRValue(Index: Integer; const Value: Double);
  public
    { Значение }
    property Dim: Integer read FDim write SetDim;
    property Values[Index: Integer]: Double read GetRValue
      write SetRValue; default;

    constructor Create(aDim: Integer);
  end; { TArrDouble }

  constructor TArrDouble.Create(aDim: Integer);
  begin
    FDim := aDim;
    SetLength(FValues, FDim);
  end;

  function TArrDouble.GetRValue(Index: Integer): Double;
  begin
    Result := FValues[Index];
  end;

  procedure TArrDouble.SetRValue(Index: Integer; const Value: Double);
  begin
    FValues[Index] := Value;
  end;

  procedure TArrDouble.SetDim(const Value: Integer);
  begin
    FDim := Value;
    SetLength(FValues, FDim);
  end;
{ ============================================================================ }
```

В представленном коде конструктор `Create` выделяет память под внутренний массив класса.

Следующий вопрос – а может ли в классе быть более одного конструктора? Ответ положительный. Несколько видоизменим предыдущий пример. Предоставим пользователю класса три способа создания массива – создание “пустого” массива, создание массива заданного размера и создание массива, данные которого хранятся в базе данных (БД).

Считаем, что схема хранения в БД выглядит так:

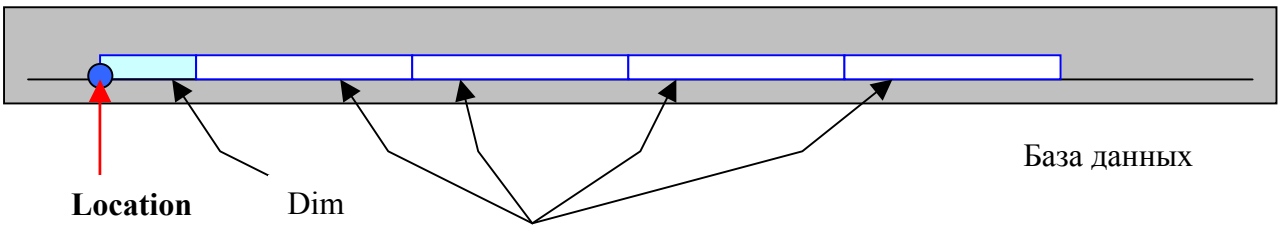


Рис. 10.12. Схема хранения массива в Базе данных

```
{ ============================================================================ }
{ Класс - динамический массив. Конструкторы }
  TArrDouble = class
  private
    FDim: Integer;
    FValues: array of Double;

    procedure SetDim(const Value: Integer);
    function GetRValue(Index: Integer): Double;
    procedure SetRValue(Index: Integer; const Value: Double);
  public
    { Значение }
    property Dim: Integer read FDim write SetDim;
    property Values[Index: Integer]: Double read GetRValue
      write SetRValue; default;

    { Создание пустого массива }
    constructor Create;
    { Создание массива из aDim элементов }
    constructor Init(aDim: Integer);
    { Создание массива с загрузкой данных из Базы данных }
    { Используем гипотетический тип данных TDBLocation - }
    { местонахождение в БД                               }
    constructor Load(Location: TDBLocation);
  end; { TArrDouble }

  constructor TArrDouble.Create;
  begin
    FDim := 0;
  end;

  constructor TArrDouble.Init(aDim: Integer);
  begin
    FDim := ADim;
    SetLength(FValues, FDim);
  end;

  constructor TArrDouble.Load(Location: TDBLocation);
  var
    i: Integer;
  begin
    // Некоторый код загрузки массива из Базы данных
    // Схема загрузки:
    // Читаем размерность (гипотетический метод, считаем, что объект
    //   "знает", где он хранится в базе данных
    ReadFromDB(Location, 0, FDim);
    // Выделяем память под массив
    SetLength(FValues, FDim);
    // Читаем массив из Базы данных
    for i := 1 to FDim do
      ReadFromDB(Location, i, FValues[i]);
  end;

{ ============================================================================ }
```

Теперь мы имеем сразу три конструктора, позволяющих полностью контролировать процесс создания объектов и выполнять его в соответствии с необходимостью либо методом `Create` (создание пустого массива, потом при необходимости установка размера `Dim = <значение>` и дальнейшая работа), либо создание массива заданной длины методом `Init`, либо загрузка массива из Базы данных методом `Load`. Осталось разобраться с нашей собственной рекомендацией об именовании. Ранее мы говорили, что лучше все конструкторы называть `Create`. В данном случае в классе их несколько, как же мы можем назвать несколько подпрограмм одинаково? Конечно, можем! Вспоминаем материал главы 7 о перегрузке подпрограмм. Необходимо сопроводить все конструкторы ключевым словом **`overload`**.

```
TArrDouble = class
private
  FDim: Integer;
  FValues: array of Double;

  procedure SetDim(const Value: Integer);
  function GetRValue(Index: Integer): Double;
  procedure SetRValue(Index: Integer; const Value: Double);
public
  { Значение }
  property Dim: Integer read FDim write SetDim;
  property Values[Index: Integer]: Double read GetRValue
    write SetRValue; default;

  constructor Create;                          overload;
  constructor Create(aDim: Integer);           overload;
  constructor Create(Location: TDBLocation); overload;
end; { TArrDouble }
```

Теперь мы имеем три *перегруженных* конструктора. Напомним, что факт перегрузки означает, что мы используем одно и то же имя для разных методов и эти методы должны быть различимы по списку параметров. Так, нам не удастся создать два конструктора с такими параметрами:

```
constructor Create(a: Integer);
constructor Create(b: Integer);
```

Компилятор в этом случае инициирует ошибку.

Для того чтобы завершить данный раздел, рассмотрим еще один пример.

```
{ ========================================================================== }
{ Обманем компилятор! }
  // Любителям обмана компилятора посвящается
  // Компилятор Object Pascal, среда Borland Delphi 7.0
  TExample = class
  public
    fd: Integer;

    constructor Create(a: Byte); overload;
    constructor Create(a: Integer); overload;
    procedure test;
  end; { TExample }

  constructor TExample.Create(a: Byte);
  begin
```

```
    fd := a;
  end;

  constructor TExample.Create(a: Integer);
  begin
    fd := a;
  end;
```

Интересный вопрос, как компилятор собирается разбираться, какой именно конструктор вызвать, если типы `Integer` и `Byte` совместимы? Проведем эксперимент.

```
  // вызываем конструктор с параметром-переменной
  var
    TExample e1, e2, e3;
    a: Integer;
    b: Byte;
  ...

  e1 := TExample.Create(a); // Разобрался, вызвал второй конструктор
  e2 := TExample.Create(b); // Разобрался, вызвал первый конструктор
  e3 := TExample.Create(1); // Невозможно разобраться. Но компилятор вызвал
                            // первый конструктор
{ ======================================================================== }
```

Если с первыми двумя случаями все ясно, компилятор посмотрел на тип переменной и сопоставил его с типом формального параметра в конструкторе `Create`, то третий случай вызывает некоторое недоумение. Каким образом компилятор решил, что 1 относится к типу `Byte`, а не к типу `Integer`? И тот ли это вариант, который ждал программист?

Вывод: не нужно писать такой код, в котором невозможно разобраться. Хорошие программы – это не те программы, в которых не могут разобраться Ваши коллеги и (или) компилятор. Хорошие программы – это программы, которые работают и работают понятным образом.

Специальные виды методов. Деструкторы

В предыдущем разделе мы научились управлять созданием объектов. А как управлять их уничтожением? Очень просто, для этого предназначен специальный тип метода – *деструктор*. Для объявления деструкторов используется ключевое слово **destructor**. Задачи деструктора – выполнить некоторые завершающие действия и освободить память. Снабдим деструкторами классы из рассмотренных ранее примеров.

```
  TArrDouble = class
  private
    FDim: Integer;
    FValues: array of Double;

    procedure SetDim(const Value: Integer);
    function GetRValue(Index: Integer): Double;
    procedure SetRValue(Index: Integer; const Value: Double);
  public
    { Значение }
    property Dim: Integer read FDim write SetDim;
    property Values[Index: Integer]: Double read GetRValue
```

```
      write SetRValue; default;

    constructor Create; overload;
    constructor Create(aDim: Integer); overload;
    constructor Create(Location: TDBLocation); overload;
    destructor  Destroy;
  end; { TArrDouble }

  destructor  TArrDouble.Destroy;
  begin
    SetLength(FValues, 0);
  end;
```

Как видно из этого примера, память, выделенная в конструкторе, освобождается в деструкторе, логично, не правда ли?

Заметим, что память для "обычных" полей, таких как, например, `FDim`, освобождается автоматически, не требуя написания в деструкторе специальных инструкций.

Теперь посмотрим на проблему завершающих действий несколько в ином ракурсе. Вернемся к классу `TSquare`.

```
  TSquare = class
  private
    {----------  Данные  -------------}
    FVisible: Boolean;  { Признак видимости              }
    FColor:   Cardinal; { Цвет границ                    }
    Fx, Fy:   Real;     { Координаты левого нижнего угла }
    Fa:       Real;     { Сторона квадрата               }

    procedure SetColor(const Value: Cardinal);
  public
    ...
    constructor Create(ax, ay, aa: Real; aColor: Cardinal;
      aVisible: Boolean);
    destructor  Destroy;
  end; { TSquare }

  constructor TSquare.Destroy;
  begin
    Hide;
  end;
```

Посмотрим внимательно на этот несложный пример. Здесь не требуется освобождать в деструкторе выделенную в конструкторе память, но требуется убрать фигуру с координатной плоскости, иначе объект исчезнет, а на экране останется видимым, странно, не правда ли?

В заключение, еще одна рекомендация по стилю именования. Вероятно, вы заметили, что деструкторы в языке Object Pascal принято называть `Destroy`.

Объявление, создание, удаление объектов. Ссылочная модель объекта. Присваивание и копирование

Узнав уже довольно много о создании классов, самое время научиться их использовать. Именно этим мы сейчас и займемся. Для того чтобы понять, как использовать созданные классы,

вспомним, что класс представляет собой тип данных, а, значит, его использование непосредственно связано с созданием переменных. Как мы уже знаем, переменные класса называются объектами. Как же происходит работа с этими переменными?

Фундаментальный принцип программирования в языке Pascal выглядит так: *прежде чем переменная может быть использована, она должна быть объявлена*.

Объявление объектов ничем не отличается от объявления “обычных” переменных.

```
type
  TMyClass = class
  public
    ...
    constructor Create;
    destructor  Destroy;
  end; { TMyClass }
...
var
  temp: TMyClass;
```

Важный для понимания момент – что реально происходит, когда компилятор встречает объявление переменной объектного типа? Какие инструкции он вставляет в объектный код?

Дело в том, что в языке Object Pascal *все переменные объектного типа – ссылки на области оперативной памяти*. Что это значит? Это значит, что в приведенном выше примере переменная `temp` – это не сам объект, а ссылка на объект (проще говоря, адрес некоторой области оперативной памяти). Важно помнить, что изначально эта ссылка равна нулю, и попытка использования объекта непосредственно после объявления в блоке **`var`** приведет к ошибке времени исполнения программы.

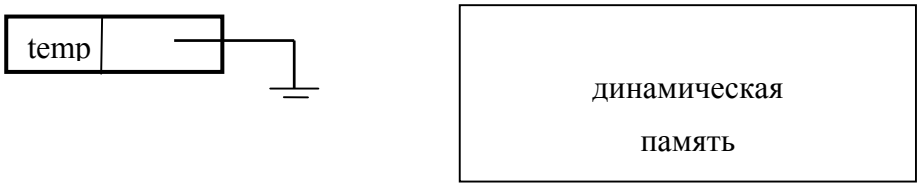


Рис. 10.13. Объявление объекта порождает нулевую ссылку

Сначала объект необходимо создать. Как Вы помните, для создания объектов служит специальный метод, называемый конструктором.

Создадим объект – динамический массив (см. пример – класс `TArrDouble`).

```
var
  arr: TArrDouble;
...
begin
  arr := TArrDouble.Create(10);
end.
```

В представленном коде происходит создание объекта `arr` типа `TArrDouble` при помощи конструктора `Create` с параметром `aDim = 10`. При этом выполняется выделение необходимой памяти. Теперь допускается использование объекта.

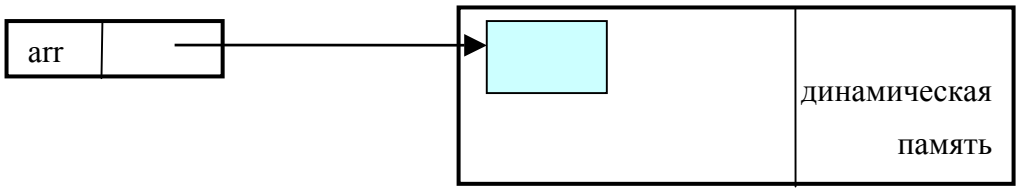


Рис. 10.14. Создание объекта посредством вызова конструктора

Доступ к полям и методам осуществляется через точку после имени объекта.

> Заметим, что создание объекта синтаксически выглядит очень хитро. Метод `Create` вызывается не через переменную `arr` (как известно, это породит ошибку из-за того, что `arr` инициализирована нулевым адресом), а через имя класса `TArrDouble`. В данном случае мы видим пример так называемого *метода класса*, который вызывается не для объекта, а для класса. Подробнее о других случаях разработки и использования методов класса можно прочитать в документации.

Уничтожение созданного объекта производится посредством вызова деструктора. Так, мы можем написать:

```
var
  arr: TArrDouble;
...
begin
  arr := TArrDouble.Create(10);
  ...
  arr.Destroy;
end.
```

Освобождение памяти – важная процедура, о которой нельзя забывать, поэтому вызывать деструкторы, когда объект далее не будет использоваться, необходимо. Вопрос в том, как это делать.

Посмотрим на приведенный выше код “под микроскопом”. В коде нет ошибки. Однако, он небезопасен. Попробуйте в порядке эксперимента (не вздумайте делать это в реальных программах) вызвать деструктор для одного и того же объекта два раза. Или вызвать деструктор для объекта, который не был создан. Как Вы увидите, это приведет к краху Вашей программы. Как этого избежать?

Первый способ – проверять перед вызовом деструктора, инициализирована ли ссылка на объект.

```
var
  arr: TArrDouble;
...
begin
  arr := TArrDouble.Create(10);
  ...
  arr.Destroy;
  if arr <> nil then
    arr.Destroy;
end.
```

Неприятности связаны с тем, что вызов деструктора освобождает память, но не обнуляет ссылку. Так, следующий код приведет к ошибке:

```
var
```

```
    arr: TArrDouble;
...
begin
    arr := TArrDouble.Create(10);
    ...
    arr.Destroy;
    if arr <> nil then
      arr.Destroy;
    // Некоторые действия
    if arr <> nil then
      arr.Destroy; // Ошибка
    // или arr.<Обращение к любому члену класса> – Ошибка
end.
```

С этим можно справиться так:

```
if arr <> nil then
begin
    arr.Destroy;
    arr := nil;
end;
```

Второй способ связан не столько со стандартом языка, сколько непосредственно с реализацией этого стандарта от фирмы Borland в рамках системы программирования Borland Delphi. Эта реализация предоставляет для всех классов метод `Free`, который выполняет проверку на **`nil`** и только потом вызывает деструктор, решая часть проблемы. Так, программируя на Delphi признак плохого стиля вызывать деструктор, вместо этого необходимо писать так:

```
arr.Free;
arr = nil; // Если Вы опасаетесь за дальнейшее
```

Заметим, правда, что для этого вы должны создавать деструкторы виртуальными, используя в классах директиву **`override`** после имени деструктора (подробнее об этом в разделе “Наследование и полиморфизм. Виртуальные методы и позднее связывание”).

Рассмотрим теперь часто встречающуюся ситуацию с присваиванием и копированием объектов. Многие из Вас, начав программировать с использованием ООП на Object Pascal, рано или поздно столкнутся со следующим примером:

```
var
    A, B: TArrDouble;
...
begin
    arr := TArrDouble.Create(10);
    ...
    B := A;
    ...
end.
```

Попробуем откомпилировать этот код – все в порядке, компиляция прошла успешно.

Вопрос: как это работает?

В голову приходят два варианта того, как это в принципе *могло бы* работать. Вариант первый состоит в том, что создается новый объект `B` и все содержимое объекта `A` копируется в `B`. Простая проверка показывает, что это не так.

```
var
    A, B: TArrDouble;
```

```
begin
  A := TArrDouble.Create(10);
  A[1] := 14; A[2] := 15;
  B := A;
  B[1] := 16; B[2] := 17;
  Writeln(A[1]:5:2, ' ', A[2]:5:2, ' ',  B[1]:5:2, ' ', B[2]:5:2);
  Readln;
  A.Free;
end.
```

Запустив эту несложную программу, мы увидим на экране следующее:

```
16.00  17.00  16.00  17.00
```

Тем самым, наше предположение не оправдалось. Оказывается, мы имеем дело со вторым возможным вариантом работы такого присваивания. Компилятор не выполнил создание нового объекта и копирование данных. Он настроил еще одну ссылку на ту же самую область памяти. Здесь необходимо быть очень внимательным. Дело в том, что после удаления объекта A ни в коем случае нельзя удалять объект B, ибо это одно и то же!

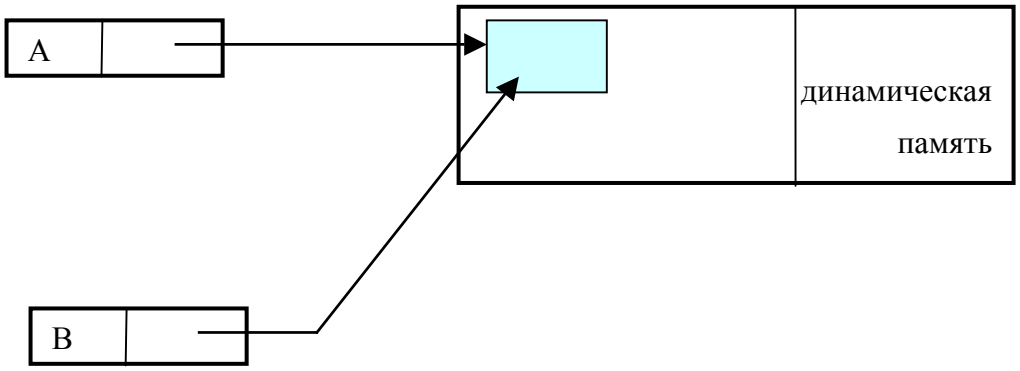


Рис. 10.15. Присваивание для объектов – присваивание адресов

А как же осуществить копирование?

Продемонстрируем на следующем примере возможный вариант организации копирования объектов на уровне класса. Попробуем понять, как это сделать. Очевидно, что для копирования данных для начала необходимо иметь объект, в который будет производиться копирование. Для этого его кто-то должен создать. Мы предлагаем создать в классе специальный конструктор с одним параметром – объектом того же типа. Задача этого конструктора – создать новый объект и скопировать в него все данные из образца, переданного в качестве параметра.

```
{ ========================================================================== }
{ Конструктор копирования }
  TArrDouble = class
  private
    FDim: Integer;
    FValues: array of Double;

    procedure SetDim(const Value: Integer);
    function GetRValue(Index: Integer): Double;
    procedure SetRValue(Index: Integer; const Value: Double);
  public
    { Значение }
    property Dim: Integer read FDim write SetDim;
```

```
    property Values[Index: Integer]: Double read GetRValue
      write SetRValue; default;

    constructor Create; overload;
    constructor Create(aDim: Integer); overload;
    constructor Create(Location: TDBLocation); overload;

    { Конструктор копирования }
    constructor Create(aSource: TArrDouble); overload;
    destructor  Destroy;
  end; { TArrDouble }

  constructor TArrDouble.Create(aSource: TArrDouble); overload;
  var
    i: Integer;
  begin
    SetDim(aSource.Dim);
    for i := 1 to Dim do
      FValues[i] := aSource[i];
  end;
{ ========================================================================== }
```

Теперь напишем в головной программе:

```
  begin
    A := TArrDouble.Create(10);
    A[1] := 14; A[2] := 15;
    C := TArrDouble.Create(A);
    C[1] := 16;
    Writeln(A[1]:5:2, ' ', A[2]:5:2, ' ',  C[1]:5:2, ' ', C[2]:5:2);

    Readln;

    A.Free;
    C.Free;
  end.
```

Запускаем… Работает!

На экране следующий результат:

```
14.00  15.00  16.00  15.00
```

Заметим, что в отличие от C++, Object Pascal не содержит *встроенных* средств для обеспечения операции копирования. В результате мы можем создавать в классах так называемый “конструктор копирования”, действуя указанным выше способом. Однако, в библиотеке VCL (библиотека классов, основа визуального программирования в Delphi) уже предприняты некоторые меры по автоматизации копирования. При рассмотрении наследования мы еще упомянем тот факт, что все классы в языке Object Pascal, в том числе и ваши собственные, для которых не указан предок, будут выведены из класса `TObject`. Сам по себе `TObject` содержит ряд полезных свойств и методов, но никак не помогает в вопросе копирования. А вот один из его потомков `TPersistent` имеет виртуальный метод `Assign`, предназначенный как раз для решения этой проблемы. Таким образом, выводя класс из `TPersistent`, вы можете перекрыть в Вашем классе метод `Assign`, реализовав в нем копирование данных, что обеспечит копирование “в стиле библиотеки VCL“.

Способы коммуникации между объектом и методами. Раннее связывание. Указатель Self

Поговорим о некоторых моментах внутреннего устройства и работы ООП в языке программирования Object Pascal. В данном разделе мы рассмотрим два вопроса – как объект "добирается" до своих методов и как методы класса получают доступ к данным объекта.

Суть первого вопроса состоит в следующем. Как было нами выяснено ранее, в отличие от обычных процедур и функций, методы вызываются не сами по себе, а для конкретных объектов. Так, мы можем написать следующий код:

```
var
  A: TArrDouble;
begin
  A := TArrDouble.Create;
  A.SetDim(20); // Вызов метода SetDim для объекта A
end.
```

В рассмотренном примере производится вызов метода `SetDim` для объекта `A`. Как компилятор разбирается, где этот метод и какой адрес нужно подставить? Для тех методов, которые мы умеем писать на настоящий момент, проблема решается на удивление просто. На этапе сборки программы подставляется адрес той области памяти, в которой находится скомпилированный код метода `SetDim` класса `TArrDouble`. Схема работы выглядит примерно так: объект `A` объявлен как переменная типа `TArrDouble`. Ищем среди методов класса `TArrDouble` метод `SetDim`. Подставляем его адрес.

В данном случае речь идет о так называемом *раннем связывании* (*early binding*), в рамках которого адрес вызываемого метода известен на этапе компиляции и сборки программы. Вы можете задаться вопросом, бывает ли по-другому. Бывает, но об этом при рассмотрении *виртуальных методов*.

Быстро разобравшись с первым вопросом, перейдем теперь ко второму. Как получается, что метод разбирается, какие именно данные использовать в теле метода? Рассмотрим пример.

```
procedure TArrDouble.SetDim(const Value: Integer);
begin
  FDim := Value;
  SetLength(FValues, FDim);
end;
```

Метод `SetDim` использует в своем теле имена `FDim`, `FValues`, `Value`. Если `Value` – параметр метода и в его использовании нет ничего необычного, то кто такие `FValues` и `FDim`?

Ясно, что это поля класса `TArrDouble`. Действительно, `SetDim` – не просто отдельная функция, а метод класса `TArrDouble`, следовательно, использование внутри метода полей класса легально. Все хорошо, но как отрабатывает следующий код?

```
A.SetDim(20); // A – объект типа TArrDouble
```

Интуитивно понятно, что метод `SetDim` должен работать с данными того объекта, который его вызывает. В приведенном примере должны использоваться поля `FDim` и `FValues` объекта `A`.

А как достигают этого результата? На самом деле метод класса всегда получает один "неявный" параметр – ссылку на объект, который вызвал метод. Для приведенного примера метод `SetDim` в реальности получает два параметра – ссылку на объект `A` и значение `Value`. Этот неявный

параметр имеет специальный идентификатор `Self`. Приведем условный код метода `SetDim` (такой, каким видит его компилятор):

```
procedure TArrDouble.SetDim(TArrDouble Self; const Value: Integer);
begin
  Self.FDim := Value;
  SetLength(Self.FValues, Self.FDim);
end;
```

Не правда ли, утомительно многократно писать префикс `Self`? Object Pascal избавляет нас от этой необходимости.

> Бывают методы, которые не получают `Self`. Это так называемые “методы класса”, о которых можно прочитать в разделе “Внутренняя структура объекта. Методы класса. Динамический контроль типов, операторы IS и AS”.

Тем не менее, в некоторых случаях указатель `Self` необходимо использовать явно. Как правило, речь идет о возврате объекта из метода.

```
function TSomeClass.SomeMethod(p: Integer): TSomeClass;
begin
  Fp := Fp + p;
  Result := Self;
end;
```

Пример разработки класса “Рациональная дробь”

Узнав уже довольно много о разработке классов на языке Object Pascal, попробуем написать пример класса для представления и осуществления операций над рациональными дробями.

Посмотрим на последовательность действий, которую необходимо проделать для разработки полноценного класса. Начнем с анализа предметной области.

Как известно, рациональная дробь есть $R = \frac{p}{q}$, где p, q – целые числа. Имеет смысл говорить о представлении в виде класса несократимых дробей, т.е. дробей, для которых НОД(p, q) = 1. Дело в том, что в машинной реализации сокращение дроби выглядит очень важной процедурой, которая страхует нас от элементарного переполнения в результате выполнения операций над дробями. Как автоматически сокращать дробь? Пока оставим этот вопрос и перейдем к проектированию данных нового класса `TRational`.

Проектирование данных

Будем моделировать числитель и знаменатель дроби целыми числами типа `Integer`. Рассмотрим вопрос о помещении данных в открытую или закрытую часть класса. Можно ли открыть данные? Здравый смысл подсказывает, что нет. Действительно, открыв данные, мы утрачиваем контроль за тем, что дробь в любой момент работы программы будет несократимой. Итак, мы скроем данные и предоставим свойства для доступа к ним, кроме того, при изменении p или q будем сокращать дробь.

Проектирование операций

Какие методы нам понадобятся?

- Конструктор: без параметров. Создает “нулевую дробь” – числитель равен 0, знаменатель равен 1.
- Конструктор: параметры метода – p и q. Создает и сокращает дробь.
- Конструктор копирования: параметр – объект типа `TRational`.
- Деструктор.
- Различные арифметические операции: сложение, вычитание, умножение, деление.
- Вывод на консоль.
- Метод сокращения дроби.

Учитывая, что алгоритмы осуществления арифметических операций хорошо известны, остановимся отдельно на алгоритме сокращения дроби.

“*Единица есть то, через что каждое из существующих считается единым. Число же – множество, составленное из единиц*”[1] – так говорил великий Евклид. Будем использовать для сокращения дроби так называемый *алгоритм Евклида* нахождения наибольшего общего делителя.

Пусть a ≤ b. Тогда согласно алгоритму

НОД (a, b) = **НОД** (b – a, a).

Будем считать, что **НОД** (0, 0) = 0, **НОД** (0, b) = b.

Известно, что вычитание в данном алгоритме может быть заменено на деление с остатком, что существенно ускоряет работу алгоритма.

НОД (a, b) = **НОД** (b mod a, a).

Во втором случае сложность алгоритма исчисляется в виде $O(\log B)$ (проверьте этот факт самостоятельно. Указание: показать, что на каждом шаге работы алгоритма размер задачи – число B – уменьшается по крайней мере в 2 раза).

Рассмотрев все необходимые вопросы, приведем программную реализацию. Выполним ее в виде модуля `rational_u`.

> В примере нам встретятся незнакомые слова **`inherited`** и **`override`**. Догадаться об их назначении на самом деле не сложно, подробно мы познакомимся с ними ниже при рассмотрении наследования.

```
{ ========================================================================== }
{ Пример 10.3 }
{ Класс Рациональная дробь }
Unit rational_u;
interface
type
  { Рациональная дробь }
  TRational = class
  private
    { Числитель и знаменатель }
    Fp, Fq: Integer;
```

[1] Евклид. “Начала”.

```
    { Поиск НОД(Fp, Fq) }
    function Euclid: Integer;
    { Сокращение дроби }
    procedure Reduce;

    { Установка Fp }
    procedure SetP(const Value: Integer);
    { Установка Fq }
    procedure SetQ(const Value: Integer);
  public
    { Числитель }
    property p: Integer read Fp write SetP;
    { Знаменатель }
    property q: Integer read Fq write SetQ;

    { Конструктор без параметров }
    constructor Create(); overload;
    { Конструктор копирования }
    constructor Create(R: TRational); overload;
    { Конструктор инициализатор }
    constructor Create(p, q: Integer); overload;
    { Деструктор }
    destructor  Destroy(); override;

    { Группа операций - не создают новый объект }
    { Сложение R0 = R0 + R1 }
    function AddMe(R: TRational): TRational;
    { Вычитание R0 = R0 - R1 }
    function SubtMe(R: TRational): TRational;
    { Умножение R0 = R0 * R1 }
    function MultMe(R: TRational): TRational;
    { Деление R0 = R0 / R1 }
    function DivMe(R: TRational): TRational;

    { Группа операций - создают новый объект }
    { Сложение R3 = R1 + R2 }
    function AddNew(R: TRational): TRational;
    { Вычитание R3 = R1 - R2 }
    function SubtNew(R: TRational): TRational;
    { Умножение R3 = R1 * R2 }
    function MultNew(R: TRational): TRational;
    { Деление R3 = R1 / R2 }
    function DivNew(R: TRational): TRational;

    { Вывод на консоль }
    procedure Print;
  end; { TRational }

implementation

{ TRational }

{ Конструктор инициализатор }
constructor TRational.Create(p, q: Integer);
begin
  inherited Create;
```

```
  Fp := p;
  Fq := q;
  Reduce;
end;

{ Конструктор без параметров }
constructor TRational.Create;
begin
  inherited Create;
  Fp := 0;
  Fq := 1;
end;

{ Конструктор копирования }
constructor TRational.Create(R: TRational);
begin
  inherited Create;
  Fp := R.p;
  Fq := R.q;
end;

{ Деструктор }
destructor TRational.Destroy;
begin
  inherited;
end;

{ Сложение R0 = R0 + R1 }
function TRational.AddMe(R: TRational): TRational;
begin
  Fp := Fp * R.Fq + R.Fp * Fq;
  Fq := Fq * R.Fq;
  Reduce;
  Result := Self;
end;

{ Деление R0 = R0 / R1 }
function TRational.DivMe(R: TRational): TRational;
begin
  Fp := Fp * R.Fq;
  Fq := Fq * R.Fp;
  Reduce;
  Result := Self;
end;

{ Умножение R0 = R0 * R1 }
function TRational.MultMe(R: TRational): TRational;
begin
  Fp := Fp * R.Fp;
  Fq := Fq * R.Fq;
  Reduce;
  Result := Self;
end;

{ Вычитание R0 = R0 - R1 }
function TRational.SubtMe(R: TRational): TRational;
```

```
begin
  Fp := Fp * R.Fq - R.Fp * Fq;
  Fq := Fq * R.Fq;
  Reduce;
  Result := Self;
end;

{ Сложение R3 = R1 + R2 }
function TRational.AddNew(R: TRational): TRational;
var
  RNew: TRational;
begin
  RNew := TRational.Create(Self);
  RNew.Fp := RNew.Fp * R.Fq + R.Fp * RNew.Fq;
  RNew.Fq := RNew.Fq * R.Fq;
  Reduce;
  Result := RNew;
end;

{ Деление R3 = R1 / R2 }
function TRational.DivNew(R: TRational): TRational;
var
  RNew: TRational;
begin
  RNew := TRational.Create(Self);
  RNew.Fp := RNew.Fp * R.Fq;
  RNew.Fq := RNew.Fq * R.Fp;
  Reduce;
  Result := RNew;
end;

{ Умножение R3 = R1 * R2 }
function TRational.MultNew(R: TRational): TRational;
var
  RNew: TRational;
begin
  RNew := TRational.Create(Self);
  RNew.Fp := RNew.Fp * R.Fp;
  RNew.Fq := RNew.Fq * R.Fq;
  Reduce;
  Result := RNew;
end;

{ Вычитание R3 = R1 - R2 }
function TRational.SubtNew(R: TRational): TRational;
var
  RNew: TRational;
begin
  RNew := TRational.Create(Self);
  RNew.Fp := RNew.Fp * R.Fq - R.Fp * RNew.Fq;
  RNew.Fq := RNew.Fq * R.Fq;
  Reduce;
  Result := RNew;
end;

{ Поиск НОД(Fp, Fq) }
```

```
function TRational.Euclid: Integer;
var
  a, b, temp: Integer;
begin
  if Fp = 0 then
  begin
    Result := 0;
    Exit;
  end;
  if Fp > Fq then
  begin
    b := Fp;
    a := Fq;
  end
  else begin
    b := Fq;
    a := Fp;
  end;
  { b > a }
  while (b mod a <> 0) do
  begin
    temp := a;
    a := b mod a;
    b := temp;
  end;
  Result := a;
end;

{ Сокращение дроби }
procedure TRational.Reduce;
var
  gcd: Integer;
begin
  gcd := Euclid;
  if gcd <> 0 then
  begin
    Fp := Fp div gcd;
    Fq := Fq div gcd;
  end;
end;

procedure TRational.SetP(const Value: Integer);
begin
  Fp := Value;
  Reduce;
end;

procedure TRational.SetQ(const Value: Integer);
begin
  Fq := Value;
  Reduce;
end;

{ Вывод на консоль }
procedure TRational.Print;
begin
```

```
  WriteLn('R= ', p, '/', q);
end;

end.
{ ========================================================================== }
```

В примере представлены два варианта арифметических операций. Первый вариант (`AddMe`, `SubtMe`, `MultMe`, `DivMe`) рассчитан на то, что изменяется сам вызывающий операцию объект.

```
  R1 := TRational.Create(x1, y1);
  R2 := TRational.Create(x2, y2);
  R1.AddMe(R2); // Математический аналог: R1 = R1 + R2;
```

Второй вариант (`AddNew`, `SubtNew`, `MultNew`, `DivNew`) создает новый объект внутри метода и возвращает его как результат.

Схема использования этой группы методов выглядит так:

```
  R1 := TRational.Create(x1, y1);
  R2 := TRational.Create(x2, y2);
  // R3 создается внутри метода
  R3 := R1.AddNew(R2); // Математический аналог: R3 = R1 + R2;
```

Агрегация

В разговоре об основных идеях объектного подхода мы упомянули два способа повторного использования уже написанного кода – агрегацию и наследование.

Под агрегацией понимается такой способ создания новых классов, в рамках которого объекты уже созданных классов входят в объявление нового класса в качестве полей.

Проиллюстрируем сказанное на примере. Так, пусть мы имеем класс `TPoint` – точка на координатной плоскости.

```
  TPoint = class
  private
    Fx, Fy: Real;
    ...
  public
     property x: Real read Fx write Fx;
     property y: Real read Fy write Fy;
     constructor Create(_x, _y: Real);
     ...
  end; { TPoint }
```

Попробуем описать Треугольник.

```
  TTriangle = class
  private
    Fx1, Fy1: Real;
    Fx2, Fy2: Real;
    Fx3, Fy3: Real;
    ...
  public
     property x1: Real read Fx1 write Fx1;
     property y1: Real read Fy1 write Fy1;
```

```
    property x2: Real read Fx2 write Fx2;
    property y2: Real read Fy2 write Fy2;
    property x3: Real read Fx3 write Fx3;
    property y3: Real read Fy3 write Fy3;
    constructor Create(_x1, _y1, _x2, _y2, _x3, _y3,: Real);
    ...
end; { TTriangle }
```

Несколько громоздко, не правда ли? Учтем к тому же, что это только фрагмент описания класса.

Попробуем подойти к проблеме с другой стороны. Треугольник – это три точки. Попробуем описать класс “Треугольник” с использованием имеющегося класса “Точка”.

```
TTriangle = class
private
  Fp1, Fp2, Fp3: TPoint;
  ...
public
    property p1: TPoint read Fp1 write Fp1;
    property p2: TPoint read Fp2 write Fp2;
    property p3: TPoint read Fp3 write Fp3;
    ...
end; { TTriangle }
```

Это и есть пример агрегации с использованием объектов класса `TPoint`.

Попробуем понять, как должен выглядеть конструктор и как будет происходить создание объекта. Логично сделать несколько конструкторов. Пусть один из них принимает в качестве параметров 6 координат (3 точки), а второй три объекта класса `TPoint`. В любом случае ясно следующее: именно конструктор класса `TTriangle` должен создавать объекты `Fp1`, `Fp2`, `Fp3`, а деструктор – удалять их. Как правило, это общая ситуация – память выделяется в конструкторе и освобождается в деструкторе. Заметим, что можно было организовать процесс по-другому, осуществляя в конструкторе лишь присваивание ссылок, оставляя вопрос с созданием/удалением объектов головной программе, но по смыслу это неправильно. Треугольник должен быть полноправным хозяином своих точек, что дает ему определенные права (он может делать с ними все, что угодно), но и накладывает некоторые обязательства (он должен следить за своевременным созданием и удалением объектов-точек).

Для начала, дополним фрагмент объявления класса точка конструктором копирования.

```
TPoint = class
private
  Fx, Fy: Real;
  ...
public
    property x: Real read Fx write Fx;
    property y: Real read Fy write Fy;
    constructor Create(_x, _y: Real); overload;
    constructor Create(p: TPoint); overload;
    ...
end; { TPoint }

constructor TPoint.Create(_x, _y: Real);
begin
  Fx := _x;
  Fy := _y;
end;
```

```
constructor TPoint.Create(p: TPoint);
begin
  Fx := p.x;
  Fy := p.y;
end;

TTriangle = class
private
  Fp1, Fp2, Fp3: TPoint;
public
  property p1: TPoint read Fp1 write Fp1;
  property p2: TPoint read Fp2 write Fp2;
  property p3: TPoint read Fp3 write Fp3;

  constructor Create(_p1, _p2, _p3: TPoint); overload;
  constructor Create(_x1, _y1, _x2, _y2, _x3, _y3: Real); overload;
  destructor Destroy; override;
end; { TTriangle }

constructor TTriangle.Create(_p1, _p2, _p3: TPoint);
begin
  Fp1 := TPoint.Create(_p1);
  Fp2 := TPoint.Create(_p2);
  Fp3 := TPoint.Create(_p3);
end;

constructor TTriangle.Create(_x1, _y1, _x2, _y2, _x3, _y3: Real);
  overload;
begin
  Fp1 := TPoint.Create(_x1, _y1);
  Fp2 := TPoint.Create(_x2, _y2);
  Fp3 := TPoint.Create(_x3, _y3);
end;

destructor TTriangle.Destroy;
begin
  Fp1.Free;
  Fp2.Free;
  Fp3.Free;
end;
```

Иногда возникает потребность использовать при объявлении класса (например, при агрегации) другой класс, который будет объявлен далее в тексте программы. При этом компилятор не сможет обнаружить этот класс и выдаст сообщение об ошибке. Существует способ ему помочь – так называемое предварительное объявление класса. Выглядит это следующим образом:

```
TExample = class; { предварительное объявление }
TMyClass = class
public
  e: TExample;
  ...
end; { TMyClass }

TExample = class
  ...
```

```
  end; { TExample }
```

Отметим также, что между предварительным объявлением и полным объявлением не должно быть ничего, кроме других объявлений типов, т.е. они должны находиться в одной секции **type**.

Наследование и полиморфизм. Виртуальные методы и позднее связывание

Рассмотрев пример агрегации, поговорим теперь о наследовании. Для лучшего понимания будем рассматривать материал на знакомом нам примере с координатной плоскостью. Вспомним, как выглядит результат нашего анализа предметной области:

1. *Точка*:
   - данные: (x, y, Visible);
   - операции: Show, Hide, MoveTo, IsVisible.
2. *Квадрат*:
   - данные: (x, y, a, Visible);
   - операции: Show, Hide, S, MoveTo, IsVisible.
3. *Круг*:
   - данные: (x, y, r, Visible);
   - операции: Show, Hide, S, MoveTo, IsVisible.
4. *Треугольник*:
   - данные: (x1, y1, x2, y2, x3, y3, Visible);
   - операции: Show, Hide, S, MoveTo, IsVisible.
5. *Прямоугольник*:
   - данные: (x1, y1, a, b, Visible);
   - операции: Show, Hide, S, MoveTo, IsVisible.
6. *Координатная плоскость*:
   - данные: (набор фигур, Visible);
   - операции: Show, Hide, S, IsVisible.

Сравним “Квадрат” и “Точку”. Очевидно, наблюдается явная преемственность по данным. Так, по сравнению с точкой у квадрата добавляется всего один новый параметр – сторона `a`. А в операциях. Совершенно ясно, что механизм работы с полем `Visible` абсолютно одинаков и у точки, и у квадрата. А вот операции `Show`, `Hide`, `MoveTo`, `S` работают по-разному.

Тем не менее, фигуры имеют достаточно общего, а значит, было бы неплохо унаследовать “Квадрат” от “Точки”, перенимая все, что было у “Точки”, добавляя сторону и механизмы для доступа к ее значению (длине) и заменяя (переписывая) основные операции.

В общем случае в языке Object Pascal это делается следующим образом. При описании нового класса мы можем указать тот класс, от которого производится наследование. В этом случае

новый класс называется *потомком*, а тот класс, от которого производится наследование (выводимость) называется *предком*.

Схема классов в программном комплексе, которая иллюстрирует отношения наследования между различными классами, называется *схемой выводимости* классов.

> Заметим, что в языке Object Pascal предок может быть только один, тогда как, например, в языке C++ их может быть сколько угодно. В этом смысле говорят, что в Pascal принято *одиночное* наследование, а в C++ *множественное* наследование. Несмотря на то, что теоретически множественное наследование обеспечивает большую гибкость при разработке программ, чем одиночное, по мнению многих теоретиков и практиков ООП одиночное наследование делает программы более понятными и защищенными от ошибок. Дело в том, что множественное наследование приводит к возникновению большого количества спорных ситуаций, в которых компилятор принимает то или иное решение, исходя из своих соображений, не всегда очевидных для разработчика, что приводит к возникновению ошибок, найти которые крайне трудно. В этом смысле отсутствие множественного наследования в Object Pascal вряд ли можно причислить к недостаткам. Строго говоря, в Object Pascal существует понятие *интерфейс* (существует оно и в других языках для обеспечения технологии компонентной разработки программ), которое позволяет использовать элементы множественного наследования там, где это необходимо. В данной книге конструкция **interface** не рассматривается.

Синтаксис организации наследования в общем случае выглядит так:

```
TBaseClass = class
...
end; { TBaseClass }

TDerivedClass = class(TBaseClass)
...
end; { TDerivedClass }
```

Таким образом, при наследовании имя класса-предка указывается в скобках. Унаследовав от класса `TBaseClass`, мы становимся обладателями всех его полей, свойств и методов.

Рассмотрим пример класса “Точка” и унаследованного от него класса “Квадрат”. Для начала поработаем с классом “Точка”.

```
{ ========================================================================== }
{ Пример 10.4 }
{ Класс Точка }
Unit figures;
interface
uses Graphics;

type
  { Класс Точка }
  TPoint = class
  private
    Fx, Fy: Word;       { Координаты }
    FVisible: Boolean; { Признак видимости }
    Canvas: TCanvas;    { "Холст" для рисования }

    procedure SetX(const Value: Word);
    procedure SetY(const Value: Word);
```

```
  public
     property x: Word read Fx write SetX;      { Координата X }
     property y: Word read Fy write SetY;      { Координата Y }
     property Visible: Boolean read FVisible; { Признак видимости }

     procedure Show;     { Показать }
     procedure Hide;     { Скрыть   }
     function  S: Word; { Площадь  }

     constructor Create(_x, _y: Word; _Cnv: TCanvas); overload;
     constructor Create(p: TPoint); overload;
     destructor  Destroy; override;
  end; { TPoint }

implementation

{ TPoint }
constructor TPoint.Create(p: TPoint);
begin
  Fx := p.x;
  Fy := p.y;
  FVisible := False;
  Canvas := P.Canvas;
end;

constructor TPoint.Create(_x, _y: Word; _Cnv: TCanvas);
begin
  Fx := _x;
  Fy := _y;
  FVisible := false;
  Canvas := _Cnv;
end;

procedure TPoint.Hide;
begin
  if Visible then
  begin
    Canvas.Pixels[Fx, Fy] := clBtnFace;
    FVisible := false;
  end;
end;

procedure TPoint.Show;
begin
  if not Visible then
  begin
    Canvas.Pixels[Fx, Fy] := clBlack;
    FVisible := true;
  end;
end;

function TPoint.S: Word;
begin
  Result := 0;
end;
```

```
procedure TPoint.SetX(const Value: Word);
begin
  Fx := Value;
  if Visible then
  begin
    Hide;
    Show;
  end;
end;

procedure TPoint.SetY(const Value: Word);
begin
  Fy := Value;
  if Visible then
  begin
    Hide;
    Show;
  end;
end;

destructor TPoint.Destroy;
begin
  Hide;
  inherited;
end;

end.
{ ========================================================================== }
```

Сделаем несколько замечаний по примеру:

1. В классе `TPoint` все данные скрыты.
2. Среди данных присутствует поле `Canvas` – холст для рисования. Это сделано для того, чтобы точка “знала”, где ее нужно рисовать. Тип данных `TCanvas` – класс из библиотеки VCL (для его использования – **uses** `Graphics` в секции **interface** модуля). Этот класс управляет отображением графики в окне. В дальнейшем мы будем пользоваться его методами, их назначение представляется очевидным по названию.
3. Для полей-координат прописаны соответствующие свойства. При установке новых значений координат производится перерисовка точки на холсте.
4. Поле `Visible` сделано доступным только для чтения.
5. Методы `Show` и `Hide` занимаются рисованием/скрытием точки в окне, используя класс `TCanvas` и его индексированное свойство `Pixels`. При этом для рисования используется цвет `clBlack` (черный – константа из модуля `Graphics`), а для скрытия – цвет `clBtnFace` (цвет фона окна – константа из модуля `Graphics`).
6. Деструктор скрывает точку на координатной плоскости.

Теперь напишем тестовую программу. Для этого создадим в Borland Delphi новый проект (оконное приложение) с одним окном, добавим к нему модуль `figures`, разместим на форме две кнопки `Show` и `Hide`. Добавим тестирующий код и получим в результате:

```
{ ========================================================================== }
{ Пример 10.5 }
```

```
{ Точки на форме }
Unit fmain;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TfrmMain = class(TForm)
    btnShow: TButton;
    btnHide: TButton;
    procedure btnShowClick(Sender: TObject);
    procedure btnHideClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmMain: TfrmMain;

implementation
{$R *.dfm}

uses figures;

var
  pt: array [1..1000] of TPoint;

procedure TfrmMain.btnShowClick(Sender: TObject);
var
  i, j, k: Word;
begin
  Randomize;
  for k := 1 to 1000 do
  begin
    i := random(Width);
    j := random(Height);
    pt[k] := TPoint.Create(i, j, Canvas);
    pt[k].Show;
  end;
end;

procedure TfrmMain.btnHideClick(Sender: TObject);
var
  k: Word;
begin
  for k := 1 to 1000 do
  begin
    pt[k].Free;
    pt[k] := nil;
  end;
end;

end.
```

```
{ ========================================================================== }
```

При нажатии на кнопку `Show` будем создавать динамически 1000 точек со случайными координатами (для генерации случайных чисел используем функцию `random`) и отображать их в окне. Процедура `Randomize` используется для инициализации датчика случайных чисел.

Рис. 10.16. Точки в окне – результат нескольких нажатий на кнопку Show

При нажатии на кнопку `Hide` производим удаление объектов-точек, находящихся в массиве точек `pt` (при этом точки скрываются в окне посредством метода `Hide` класса `TPoint`), после чего инициализируем значением **`nil`** элементы массива. В противном случае повторное нажатие на кнопку `Hide` приведет к краху программы.

Убедившись в работоспособности созданного класса `TPoint`, выведем из него класс `TSquare`.

```
TSquare = class(TPoint)
private
  Fa: Word;

  procedure SetA(const Value: Word);
public
  property a: Word read Fa write SetA;

  procedure Show;
  procedure Hide;

  constructor Create(_x, _y, _a: Word; _Cnv: TCanvas); overload;
  constructor Create(s: TSquare); overload;
  destructor  Destroy; override;
end; { TSquare }
```

Первое, что бросается в глаза, – простота и лаконичность объявления класса. Действительно, ведь существенная часть функциональности заимствована из `TPoint`. Проведем анализ объявления класса:

1. Добавлено поле `Fa` и свойство `a` для доступа к нему. Поля `Fx`, `Fy`, `FVisible` и вся связанная с ними функциональность (в том числе и свойства) унаследованы от предка.

2. Переопределены методы `Show` и `Hide`. Действительно, квадрат изображается и скрывается отличным от точки способом. Заметим, что, переписав заголовки методов `Show` и `Hide`, мы осуществили так называемое “перекрытие методов”. Новые реализации `Show` и `Hide` скрывают старые варианты. Заметим, тем не менее, что мы можем вызвать внутри, например, метода `Show` метод `Show` предка, написав **`inherited`** `Show`. Вот мы и выяснили, зачем нужно ключевое слово **`inherited`** – вызов перекрытого метода предка!
3. Для `Show` в данном примере это не актуально, а вот для конструкторов и деструкторов – крайне важно! Создание и уничтожение объектов должно происходить в полном соответствии с их внутренней структурой. Так, конструктор должен вызвать конструктор предка и только потом производить дополнительные действия. Деструктор, напротив, сначала должен совершить завершающие действия для новых данных и только потом вызвать деструктор предка для аналогичных действий над данными предка.

Проиллюстрируем сказанное выше реализацией методов класса `TSquare`.

```
constructor TSquare.Create(_x, _y, _a: Word; _Cnv: TCanvas);
begin
  inherited Create(_x, _y, _Cnv);
  Fa := _a;
end;

constructor TSquare.Create(s: TSquare);
begin
  inherited Create(s.x, s.y, s.Canvas);
  Fa := s.a;
end;

destructor TSquare.Destroy;
begin
  inherited;
end;

procedure TSquare.Show;
begin
  if not Visible then
  begin
    Canvas.Pen.Color := clBlue;
    Canvas.Rectangle(x, y + a, x + a, y);
    FVisible := true;
  end;
end;

procedure TSquare.Hide;
begin
  if Visible then
  begin
    Canvas.Pen.Color := clBtnFace;
    Canvas.Rectangle(x, y + a, x + a, y);
    FVisible := false;
  end;
end;

procedure TSquare.SetA(const Value: Word);
begin
```

```
    Fa := Value;
    if Visible then
    begin
      Hide;
      Show;
    end;
  end;
```

Посмотрим, как все это работает. Дополним наше оконное приложение двумя кнопками на форме, объявим в секции **implementation** модуля формы `fmain` массив “Квадратов” `sq` и реализуем методы реакции на нажатие новых кнопок.

```
  ...
  var
    sq: array [1..200] of TSquare;
  ...
  procedure TvbtnShow1.btnShow1Click(Sender: TObject);
  var
    i, j, a, k: Word;
  begin
    Randomize;
    for k := 1 to 200 do
    begin
      i := random(Width);
      j := random(Height);
      a := random(50);
      sq[k] := TSquare.Create(i, j, a, Canvas);
      sq[k].Show;
    end;
  end;

  procedure TvbtnShow1.btnHide1Click(Sender: TObject);
  var
    k: Word;
  begin
    for k := 1 to 200 do
    begin
      sq[k].Free;
      sq[k] := nil;
    end;
  end;
```

Запустим пример и посмотрим, что получилось, нажав несколько раз кнопку `btnShow1`.

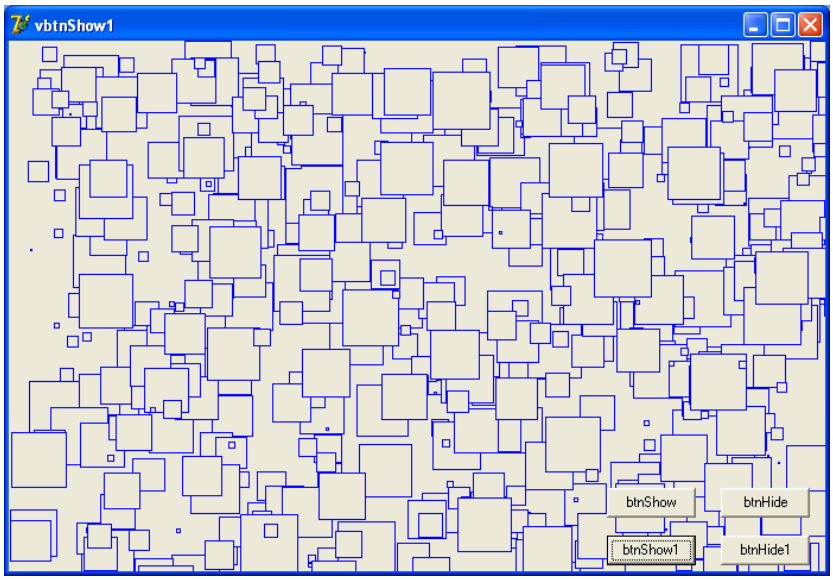


Рис. 10.17. Квадраты в окне – результат нескольких нажатий на кнопку Show1

На рисунке возможный результат нескольких нажатий.

Теперь нажмем на кнопку `btnHide1`. В чем дело? Почему ничего не происходит? Попробуем разобраться с происходящим. В режиме отладки проанализируем последовательность работы. Открывается следующая картина:

1. Попадаем в метод `btnHide1Click`.
2. В цикле вызываем метод `Free`.
3. Попадаем в деструктор класса `TSquare`.
4. Ключевое слово **`inherited`** переносит нас в деструктор класса `TPoint`.
5. Ага! Вызывается метод `Hide` класса `TPoint`, а не метод `Hide` класса `TSquare`, соответственно, у квадрата “стирается” лишь 1 точка – левый нижний угол.

Что делать? Нам нужно, чтобы в окне цветом фона рисовался квадрат, а не точка. Первое возможное решение состоит в том, чтобы переписать деструктор класса `TSquare`, поместив туда код, скопированный из `TPoint`, с заменой рисования точки рисованием квадрата. В результате все будет работать.

Как избежать этого дублирования? Как добиться того, чтобы, попав в деструктор класса `TPoint`, при вызове метода `Hide` компилятор “сообразил”, что вызывающий деструктор объект на самом деле `TSquare` и вызвал метод `Hide` квадрата?

Сложная задача для компилятора и компоновщика. Вернее, не решаемая. Все дело в том, что на этапе компиляции и сборки адрес того метода, который необходимо вызвать в деструкторе класса `TPoint` не известен. Лишь во время работы выяснится, кто именно его вызвал (`TPoint`, `TSquare` или кто-нибудь еще). В результате мы имеем дело с необходимостью “полиморфного” поведения, т.е. определения адреса метода, который необходимо вызвать, на этапе выполнения программы.

Случай, когда адрес вызываемого метода определяется не на этапе компиляции и сборки программы (ранее связывание), а на этапе работы программы, называется *поздним связыванием* (*late binding*). Язык Object Pascal предоставляет средства для использования

этого сложного механизма в программах. Рассмотрим, в чем они состоят, но прежде отметим несколько важных моментов.

1. Позднее связывание возможно тогда и только тогда, когда классы находятся в иерархии наследования.
2. Все классы языка Object Pascal, у которых явно не указан предок, считаются выведенными из класса `TObject`, содержащего некоторую полезную функциональность (разные системные механизмы, которые будут упомянуты в разделе “Внутренняя структура объекта. Методы класса. Динамический контроль типов, операторы IS и AS”). Таким образом, класс `TObject` является общим предком для всех классов. Его данные и методы содержатся во всех объектах и могут быть использованы.
3. Чтобы указать компилятору и сборщику на необходимость использования механизма позднего связывания, требуется в конце заголовка метода при его объявлении поместить ключевое слово **`virtual`**. Далее в потомках необходимо в точности сохранять заголовок метода (реализация, разумеется, может быть изменена), за исключением замены слова **`virtual`** на **`override`** (вот мы и выяснили, что означало таинственное **`override`** после заголовка деструктора).

Методы, объявленные как **`virtual`**, и в дальнейшем переопределяемые в потомках с ключевым словом **`override`**, называются *виртуальными*.

Изменим наш пример с учетом приобретенных знаний.

```
TPoint = class
private
  Fx, Fy: Word;        { Координаты             }
  FVisible: Boolean;   { Признак видимости      }
  Canvas: TCanvas;     { "Холст" для рисования }

  procedure SetX(const Value: Word);
  procedure SetY(const Value: Word);
public
   property x: Word read Fx write SetX;      { Координата X       }
   property y: Word read Fy write SetY;      { Координата Y       }
   property Visible: Boolean read FVisible; { Признак видимости }

   procedure Show; virtual; { Показать }
   procedure Hide; virtual; { Скрыть   }
   function  S: Word;       { Площадь  }

   constructor Create(_x, _y: Word; _Cnv: TCanvas); overload;
   constructor Create(p: TPoint); overload;
   destructor  Destroy; override;
end; { TPoint }

{ Класс Квадрат }
TSquare = class(TPoint)
private
  Fa: Word;
  procedure SetA(const Value: Word);
public
   property a: Word read Fa write SetA;

   procedure Show; override;
```

```
    procedure Hide; override;
    constructor Create(_x, _y, _a: Word; _Cnv: TCanvas); overload;
    constructor Create(s: TSquare); overload;
    destructor  Destroy; override;
  end; { TSquare }
```

Итак, мы сделали методы `Show` и `Hide` виртуальными. Реализация их при этом не изменится (указывать при реализации **`virtual`** и **`override`** не нужно). Компилируем и запускаем пример – теперь все работает!

> Заметим, что деструктор у нас и так был виртуальным. Связано это с тем, что у класса `TObject` – общего предка, деструктор `Destroy` является виртуальным. Сделано это для того, что корректно происходило освобождение памяти по всей иерархии наследования.

Реализуем классы “Круг” (`TCircle`) и “Прямоугольник” (`TRectangle`). Будем выводить `TRectangle` из `TSquare`, добавляя сторону и перекрывая методы рисования/скрытия. Как реализовать `TCircle`? Сравнивая `TCircle` и `TSquare`, обнаруживаем, что они полностью совпадают по данным, за исключением их интерпретации. Выведем `TCircle` из `TSquare`, перекрыв методы рисования.

Отдельно остановимся на классе “Координатная плоскость” (`TPlane`). Учитывая, что `TPlane` – набор фигур, не имеет смысла выводить класс из фигуры. Будем выводить `TPlane` из стандартного `TObject`. Попробуем описать данные `TPlane`. Как было указано ранее, данные этого класса составляет массив фигур. Тут нас, однако, подстерегает некоторая проблема. К сожалению, все фигуры принадлежат к разным типам. Значит, объявить массив нам не удастся? Или все-таки есть возможность?

Вспомним, что нам известно про массив. Массив – средство хранения больших объемов *однотипных* данных. Здесь налицо данные *разнотипные*. Как их объединить вместе? На помощь приходит важнейший элемент синтаксиса языка Object Pascal, специально предназначенный, в том числе, и для решения таких проблем. Вспомним, что, объявляя переменную объектного типа, мы на самом деле объявляем указатель на нее, т.е. адрес. Поскольку все адреса однотипны, объединить их в массив должно быть возможно. Единственно, нужно иметь в виду следующее соглашение: при объявлении таких массивов необходимо использовать базовый класс иерархии. Объявив `FData: array of TPoint`, мы получим возможность на этапе работы программы складывать туда различные фигуры, т.к. в языке Object Pascal допускается присваивание объектной переменной базового типа любого объекта-потомка.

```
var
  p: TPoint;
  q: TCircle;
begin
  ...
  p := q; // Так можно
  q := p; // Так нельзя
  ...
end.
```

Таким образом, объявляя массив из `TPoint`, мы сможем хранить в нем любые фигуры, выведенные из `TPoint`.

Следующий вопрос: как работать с этими фигурами? Ведь у каждого из классов есть свои методы, в том числе `Show` и `Hide`, которые работают по-разному по всей иерархии наследования. Здесь на помощь приходит тот факт, что каждый объект хранит информацию о том, кто он есть на самом деле, то есть информацию о своем типе. Сделав методы `Show` и `Hide` по всей иерархии

виртуальными, мы добьемся того, что `FData[i].Show` на этапе работы программы разберется, кто лежит на самом деле в i-ой ячейке массива `FData`, и вызовет соответствующий метод `Show`. Это еще один пример работы механизма позднего связывания и полиморфного поведения.

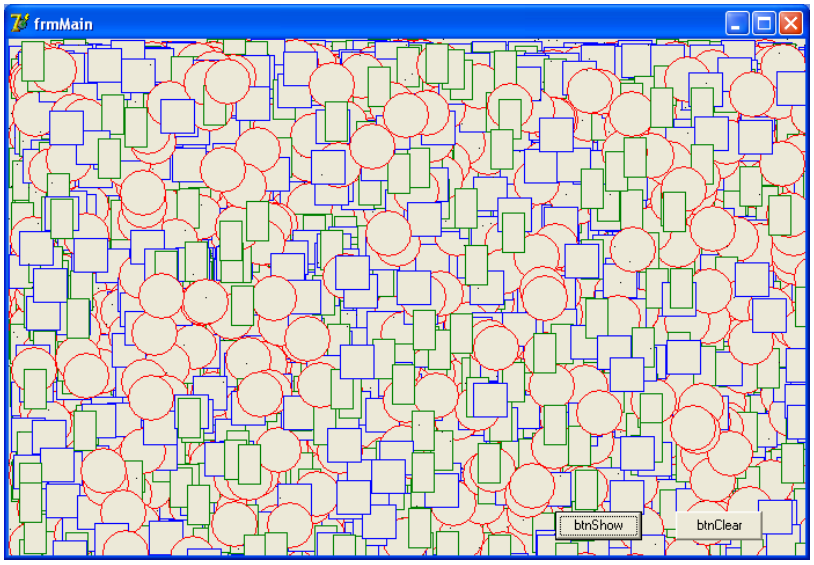


Рис. 10.18. Фигуры на координатной плоскости

Приведем полностью исходный код примера и демонстрационного приложения.

```
{ ========================================================================== }
{ Пример 10.6 }
{ Фигуры на форме }
Unit figures;
interface
uses Graphics;

type
  TPoint = class
  private
    Fx, Fy: Word;
    FVisible: Boolean;
    Canvas: TCanvas;

    procedure SetX(const Value: Word);
    procedure SetY(const Value: Word);
  public
     property x: Word read Fx write SetX;
     property y: Word read Fy write SetY;
     property Visible: Boolean read FVisible;

     procedure Show; virtual;
     procedure Hide; virtual;
     function S: Word;

     constructor Create(_x, _y: Word; _Cnv: TCanvas); overload;
     constructor Create(p: TPoint); overload;
     destructor  Destroy; override;
  end; { TPoint }
```

```
TSquare = class(TPoint)
private
  Fa: Word;
  procedure SetA(const Value: Word);
public
   property a: Word read Fa write SetA;

   procedure Show; override;
   procedure Hide; override;

   constructor Create(_x, _y, _a: Word; _Cnv: TCanvas); overload;
   constructor Create(s: TSquare); overload;
   destructor  Destroy; override;
end; { TSquare }

TCircle = class(TSquare)
public
   procedure Show; override;
   procedure Hide; override;

   constructor Create(_x, _y, _r: Word; _Cnv: TCanvas); overload;
   constructor Create(c: TSquare); overload;
   destructor  Destroy; override;
end; { TCircle }

TRectangle = class(TSquare)
private
  Fb: Word;
  procedure SetB(const Value: Word);
public
   property B: Word read Fb write SetB;

   procedure Show; override;
   procedure Hide; override;

   constructor Create(_x, _y, _a, _b: Word; _Cnv: TCanvas); overload;
   constructor Create(r: TRectangle); overload;
   destructor  Destroy; override;
end; { TRectangle }

TPlane = class(TObject)
private
  FData: array of TPoint;
  FDim: Word;
  FCanvas: TCanvas;

  function GetData(Index: Word): TPoint;
  procedure SetData(Index: Word; const Value: TPoint);
public
  property Dim: Word read FDim;
  property Canvas: TCanvas read FCanvas;
  property Data[Index: Word]:TPoint read GetData write SetData; default;

  procedure Show;
  procedure Clear;
```

```
    procedure Add(Index: Word; f: TPoint);

    constructor Create(_dim: Word; _Cnv: TCanvas);
    destructor  Destroy; override;
  end; { TPlane }

implementation

{ TPoint }

constructor TPoint.Create(p: TPoint);
begin
  Fx := p.x;
  Fy := p.y;
  Canvas := P.Canvas;
  FVisible := false;
end;

constructor TPoint.Create(_x, _y: Word; _Cnv: TCanvas);
begin
  Fx := _x;
  Fy := _y;
  FVisible := false;
  Canvas := _Cnv;
end;

procedure TPoint.Hide;
begin
  if Visible then
  begin
    Canvas.Pixels[Fx, Fy] := clBtnFace;
    FVisible := false;
  end;
end;

procedure TPoint.Show;
begin
  if not Visible then
  begin
    Canvas.Pixels[Fx, Fy] := clBlack;
    FVisible := true;
  end;
end;

function TPoint.S: Word;
begin
  Result := 0;
end;

procedure TPoint.SetX(const Value: Word);
begin
  Fx := Value;
  if Visible then
  begin
    Hide;
    Show;
```

```
  end;
end;

procedure TPoint.SetY(const Value: Word);
begin
  Fy := Value;
  if Visible then
  begin
    Hide;
    Show;
  end;
end;

destructor TPoint.Destroy;
begin
  Hide;
  inherited;
end;

{ TSquare }

constructor TSquare.Create(_x, _y, _a: Word; _Cnv: TCanvas);
begin
  inherited Create(_x, _y, _Cnv);
  Fa := _a;
end;

constructor TSquare.Create(s: TSquare);
begin
  inherited Create(s.x, s.y, s.Canvas);
  Fa := s.a;
end;

destructor TSquare.Destroy;
begin
  inherited;
end;

procedure TSquare.Show;
begin
  if not Visible then
  begin
    Canvas.Pen.Color := clBlue;
    Canvas.Rectangle(x, y + a, x + a, y);
    FVisible := true;
  end;
end;

procedure TSquare.Hide;
begin
  if Visible then
  begin
    Canvas.Pen.Color := clBtnFace;
    Canvas.Rectangle(x, y + a, x + a, y);
    FVisible := false;
  end;
```

```
end;

procedure TSquare.SetA(const Value: Word);
begin
  Fa := Value;
  if Visible then
  begin
    Hide;
    Show;
  end;
end;

{ TCircle }

constructor TCircle.Create(_x, _y, _r: Word; _Cnv: TCanvas);
begin
  inherited Create(_x, _y, _r, _Cnv);
end;

constructor TCircle.Create(c: TSquare);
begin
  inherited Create(c.x, c.y, c.a, c.Canvas);
end;

destructor TCircle.Destroy;
begin
  inherited;
end;

procedure TCircle.Hide;
begin
  if Visible then
  begin
    Canvas.Pen.Color := clBtnFace;
    Canvas.Ellipse(x, y + a, x + a, y);
    FVisible := false;
  end;
end;

procedure TCircle.Show;
begin
  if not Visible then
  begin
    Canvas.Pen.Color := clRed;
    Canvas.Ellipse(x, y + a, x + a, y);
    FVisible := true;
  end;
end;

{ TRectangle }

constructor TRectangle.Create(_x, _y, _a, _b: Word; _Cnv: TCanvas);
begin
  inherited Create(_x, _y, _a, _Cnv);
  Fb := _b;
end;
```

```
constructor TRectangle.Create(r: TRectangle);
begin
  inherited Create(r.x, r.y, r.a, r.Canvas);
  Fb := r.b;
end;

destructor TRectangle.Destroy;
begin
  inherited;
end;

procedure TRectangle.Hide;
begin
  if Visible then
  begin
    Canvas.Pen.Color := clBtnFace;
    Canvas.Rectangle(x, y + b, x + a, y);
    FVisible := false;
  end;
end;

procedure TRectangle.Show;
begin
  if not Visible then
  begin
    Canvas.Pen.Color := clGreen;
    Canvas.Rectangle(x, y + b, x + a, y);
    FVisible := true;
  end;
end;

procedure TRectangle.SetB(const Value: Word);
begin
  Fb := Value;
  if Visible then
  begin
    Hide;
    Show;
  end;
end;

{ TPlane }

constructor TPlane.Create(_dim: Word; _Cnv: TCanvas);
begin
  inherited Create;
  FDim := _dim;
  FCanvas := _Cnv;
  SetLength(FData, FDim);
end;


destructor TPlane.Destroy;
var
  i: Word;
```

```
begin
  for i := 0 to Dim - 1 do
    FData[i].Free;
  SetLength(FData, 0);
  inherited;
end;

function TPlane.GetData(Index: Word): TPoint;
begin
  Result := FData[Index];
end;

procedure TPlane.SetData(Index: Word; const Value: TPoint);
begin
  FData[Index] := Value;
end;

procedure TPlane.Clear;
var
  i: Word;
begin
  for i := 0 to Dim - 1 do
  begin
    FData[i].Free;
    FData[i] := nil;
  end;
end;

procedure TPlane.Show;
var
  i: Word;
begin
  for i := 0 to Dim - 1 do
    FData[i].Show;
end;

procedure TPlane.Add(Index: Word; f: TPoint);
begin
  FData[Index] := f;
end;

end.

{ Демонстрационное приложение }
Unit fmain;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Figures;

type
  TfrmMain = class(TForm)
    btnShow: TButton;
    btnClear: TButton;
    procedure btnShowClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
```

```
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
    Plane: TPlane;
  public
    { Public declarations }
  end;

var
  frmMain: TfrmMain;

implementation
{$R *.dfm}

procedure TfrmMain.FormCreate(Sender: TObject);
begin
  Plane := TPlane.Create(10000, Canvas);
end;

procedure TfrmMain.FormDestroy(Sender: TObject);
begin
  Plane.Free;
end;

procedure TfrmMain.btnShowClick(Sender: TObject);
var
  i1, j1, i2, j2, k: Word;
  f: 0..3;
  p: TPoint;
begin
  Randomize;
  for k := 1 to Plane.Dim do
  begin
    i1 := random(Width);
    j1 := random(Height);
    f := random(4);
    case f of
      0: p := TPoint.Create(i1, j1, Plane.Canvas);
      1: p := TSquare.Create(i1, j1, 30, Plane.Canvas);
      2: p := TCircle.Create(i1, j1, 40, Plane.Canvas);
      3: p := TRectangle.Create(i1, j1, 20, 35, Plane.Canvas);
    end;
    Plane.Add(k - 1, p);
  end;
  Plane.Show;
end;

procedure TfrmMain.btnClearClick(Sender: TObject);
begin
  Plane.Clear;
end;

end.
{ ============================================================================ }
```

Нашу цепочку классов можно продолжать сколь угодно долго, выводя все новые и новые фигуры. Понятно, что у каждого из нас могут получиться несколько отличные схемы наследования. Построение удачной схемы наследования – важный шаг для того, чтобы проект оказался успешным. Искусство создания подобных схем приходит с годами и опытом. Экспериментируйте, обращайте внимание на закономерности, и у Вас тоже будет получаться!

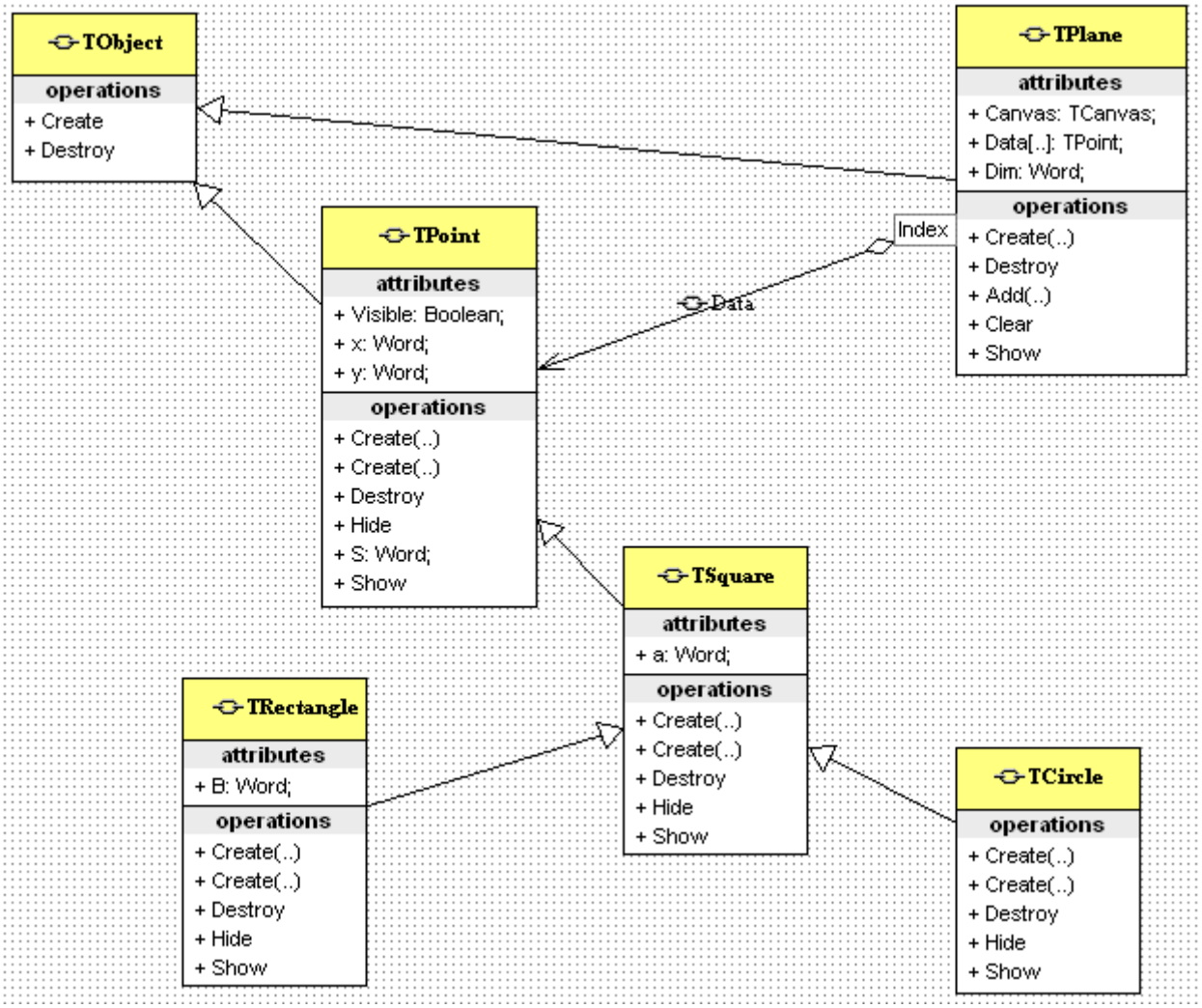


Рис. 10.19. Фигуры на координатной плоскости – схема выводимости

Абстрактные методы

Быть может, Вам уже попадались в литературе по программированию “умные слова”: концепция, парадигма, абстракция, иерархия? Надеемся, Вашей первой реакцией не было желание закрыть книгу и сложить ее куда-нибудь подальше или предложить знакомым как безвредное лекарство от бессонницы. Обещаем, в этом разделе не будет долгого и нудного философствования. Вместо этого, мы вкратце познакомимся с одним из элементов объектно-ориентированного программирования – *абстрактными методами*.

Как стало ясно из предыдущего раздела, построение цепочек наследования – мощный инструмент проектирования и разработки программ. Иногда встречаются ситуации, когда мы хотим обеспечить *полиморфное поведение* – присваивание объектов производных типов элементу базового типа с последующим вызовом “правильных” (часто говорят “релевантных”)

методов в момент работы программы. Это обеспечивается технологией позднего связывания и механизмом виртуальных методов. В связи с этим во многих случаях является оправданным включение большого числа виртуальных методов в базовый класс. Но как быть, если для базового класса эти методы не имеют смысла и содержанием будут наполнены лишь в потомках?

В предыдущем примере мы не стали реализовывать класс “Треугольник”. Подумаем, как он будет выглядеть. Очевидно, что мы хотели бы сохранить полиморфные `Show` и `Hide`. Получается, что для этого мы должны вывести его из `TPoint`. Это возможно, но несколько странно, а ведь объектный подход подразумевает естественность представления. Отсюда вывод – нужен базовый для всех класс “Фигура”, из которого можно вывести с одной стороны “Точку”, а с другой стороны “Треугольник”, сохранив дальше иерархию наследования.

Итак, мы хотим использовать класс “Фигура” в качестве базового для образования иерархии классов. При этом мы хотим предусмотреть виртуальный метод `Show`, который будет заниматься отображением фигур на координатной плоскости. Зададимся вопросом, как должна выглядеть реализация этого метода? Для точки, круга, квадрата и даже для самой координатной плоскости все ясно, всем известно, как рисовать стандартные геометрические фигуры. А вот как быть с рисованием объектов класса “Фигура”? И вообще, должны ли быть в программе экземпляры этого типа, ведь в предметной области нет “Фигур”, есть точки, круги, квадраты…

Возможный вариант повесить у себя перед глазами плакат следующего содержания: “Никогда не создавать экземпляры класса “Фигура” и написать в этом классе пустую реализацию метода `Show`.

```
TShape = class
  private
    ...
  public
    procedure Show; virtual;
    ...
end;

procedure TShape.Show;
begin
  { пустая реализация }
end;
```

Ясно, что это не лучший способ. Кто же будет страховать нас от ошибок? Для этого Object Pascal имеет специальный механизм – *абстрактные методы*.

Для начал дадим определение. *Абстрактные методы – виртуальные методы, для которых в классе отсутствует реализация*. Должны быть переопределены в потомках с указанной реализацией.

Так, для рассмотренного примера мы можем написать:

```
TShape = class
  private
    ...
  public
    procedure Show(); virtual; abstract;
    ...
end;
```

Заметим, что в этом случае мы не можем указать реализацию для метода `Show`.

Как быть с классом `TShape`? Обзаведясь хотя бы одним абстрактным методом, класс `TShape` становится *абстрактным классом*. В языке программирования C++ это означало бы, что

мы не можем создавать экземпляры класса `TShape`. В языке Object Pascal требования несколько понижены, мы можем создавать экземпляры, но компилятор выдает предупреждение о том, что мы создали экземпляр абстрактного класса (`constructing instance of 'TShape' containing abstract method TShape.Show`). А вот попытка вызова метода `Show` для объекта типа `TShape` инициирует ошибку во время работы программы (исключение `EAbstractError`), что совершенно логично. На то он и абстрактный метод, чтобы его нельзя было вызвать, компилятор контролирует это, страхуя нас от неприятностей.

В потомках мы должны переопределить метод `Show` обычным образом, используя директиву **`override`**.

```
TPoint = class(TShape)
private
  ...
public
  procedure Show(); override;
  ...
end;

procedure TPoint.Show;
begin
  { реализация }
  ...
end;
```

Внутренняя структура объекта. Методы класса. Динамический контроль типов, операторы IS и AS

Давно подмечено, что многие из нас, с самого юного возраста обожают разбирать на запасные части все, что попадает в руки. Вы за собой такого не помните? Присмотритесь внимательней за поведением ребенка, постепенно, с большим старанием отдирающим нос у какой-нибудь мягкой игрушки, или пытающегося проникнуть внутрь игрушечного автомобиля, или с довольным видом и выражением полного счастья на лице вынимающего аккумулятор из сотового телефона родителей. Тяга к познанию окружающего мира незримо присутствует внутри каждого из нас.

К сожалению, формат данной книги не позволяет нам подробно байт за байтом проанализировать содержимое переменных объектного типа и ассемблерный код текста программы, получающегося после того, как над ним на славу поработал компилятор. Справедливости ради надо сказать, что эта информация является полезной для общего развития, но обычно не более того. Нам было бы довольно трудно привести понятный пример обозримого размера, иллюстрирующий необходимость знания того, по какому смещению находится ссылка на таблицу виртуальных методов и т.д. В конце концов, эти данные обычно не документированы, меняются от версии к версии компилятора, что делает их использование при программировании недопустимым (остается еще взлом программ, но мы надеемся, что Вы не собираетесь этим заниматься). Поэтому мы ограничимся общим изложением, иллюстрирующим, как все устроено изнутри, и вполне, как мы считаем, достаточным для понимания принципов работы.

Итак, что такое объект? Во-первых, еще раз вспомним тот факт, что любой класс в языке Object Pascal является выведенным из класса `TObject`, если не указан другой предок. Класс `TObject` выполняет следующие основные функции:

1. поддерживает унифицированный механизм для создания, обработки и уничтожения экземпляров объектов производных типов;
2. осуществляет выделение, инициализацию и освобождение памяти для хранения экземпляров объектов;
3. предоставляет информацию о типе объекта на этапе работы программы (RTTI – runtime type information);
4. поддерживает механизм обработки сообщений (подробнее можно посмотреть в описании библиотеки VCL) и некоторые другие функции.

Из приведенного списка мы уже знакомы с пунктами 1 и 2, последний 4-ый пункт выходит за рамки данной книги, поскольку относится не столько к языку программирования Object Pascal, сколько к реализации библиотеки визуальных компонентов VCL. В данном разделе мы будем говорить о 3-ем пункте – работа с информацией о типе объекта на этапе исполнения программы.

Итак, объект “изнутри” это:

1. Указатель на специальную структуру RTTI, содержащую информацию о том, к какому классу принадлежит объект, и как этот класс устроен. Основными полями этой структуры являются:
   a. указатель на таблицу виртуальных методов с адресами методов;
   b. указатель на таблицу динамических методов с адресами методов;
   c. имя класса (в виде строки) и его размер;
   d. указатель на аналогичную RTTI-структуру предка;
   e. размер экземпляра (объекта).
2. Экземпляры предков со своими данными.
3. Данные объекта.

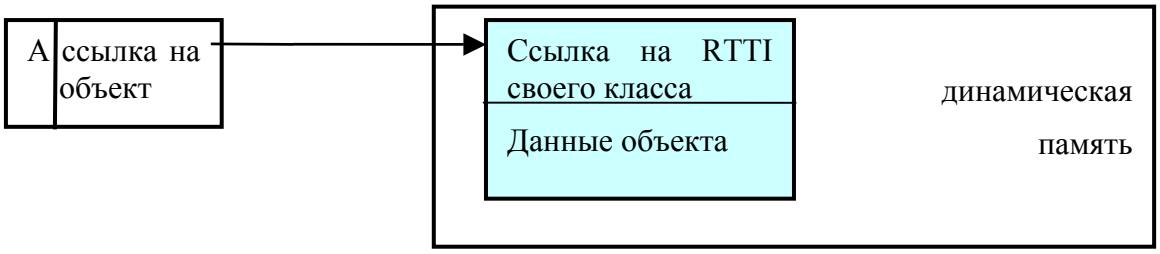


Рис. 10.20. Внутренняя структура объекта

Как видим, объект представляет собой совокупность данных и информации о своем классе. Как использовать данные объекта, мы уже знаем. А как использовать информацию о классе?

Во-первых, эта информация сплошь и рядом используется автоматически. Так, таблицы виртуальных методов применяются в рамках работы механизма позднего связывания, таблицы динамических методов – при функционировании механизма обработки сообщений (см. описание библиотеки VCL) и т.д. Остальная информация тоже находит применение в программах. Мы рассмотрим один, самый важный из аспектов, связанный с использованием этой информации – динамический контроль типов во время работы программы.

Заметим, что к части информации из RTTI можно получить доступ посредством вызова методов класса `TObject` (`ClassType`, `ClassInfo`, `ClassParent`). При этом, вызов

метода `ClassInfo` позволяет добраться до самой структуры RTTI, получив указатель на нее. Однако в документации не рекомендуется использовать непосредственно саму структуру, поскольку она может быть изменена при переходе к следующей версии Object Pascal. Отметим также тот факт, что указанные выше методы принадлежат к числу так называемых методов класса (в терминологии C++ – статических методов). Эти методы описываются с ключевым словом **`class`** перед **`procedure`** или **`function`**. Основной их особенностью является то, что они не получают указатель `Self` и, соответственно, не могут работать с полями объекта. Это методы, которые работают с данными класса, например с RTTI, как сделано в классе `TObject`.

Рассмотрим распространенную ситуацию.

Пусть мы имеем некоторую подпрограмму `DoSomething`, которая получает параметр объектного типа данных. Пусть эта подпрограмма такова, что мы не можем написать для этого параметра один конкретный тип данных, поскольку она должна осуществлять разные действия для разных типов данных. Посмотрим на следующий код:

```
procedure DoSomething(Obj: TObject);
var
  P: TPoint;
  S: TSquare;
begin
  if Obj IS TPoint then
  begin
    P := Obj AS TPoint;
    P.Hide;
  end
  else
    if Obj IS TSquare then
    begin
      S := Obj AS TSquare;
      S.Show;
    end;
end;
```

Код демонстрирует использование двух операторов языка – **`IS`** и **`AS`**.

Оператор **`IS`** осуществляет динамический контроль типов. Он проверяет, принадлежит ли пришедший в процедуру объект `Obj` тому или иному объектному типу данных. В момент срабатывания (не в момент компиляции, а в момент работы программы) у объекта `Obj` запрашивается информация о типе из RTTI, после чего производится сравнение с искомым типом данных (`TPoint`, `TSquare` в рассматриваемом примере). В результате **`IS`** возвращает значение **`true`** или **`false`**.

Оператор **`AS`** осуществляет безопасное преобразование типа на этапе работы программы. После того, как **`IS`** подтвердил, что `Obj` – действительно объект типа `TPoint`, оператор **`AS`** производит приведение типа от `TObject` к `TPoint`, что позволяет вызывать для переменной `P` методы класса `TPoint` (напомним, что для `Obj` доступны лишь методы `TObject`).

Использование операторов **`IS`** и **`AS`** – важный способ обеспечения корректной работы программы.

В случае попытки некорректного преобразования оператор **`AS`** сгенерирует ошибочную ситуацию – исключение, которое может быть обработано в программе. В результате пользователь не увидит сообщений типа “`Access violation`” или предложений обратиться к разработчику.

Выводы

Итак, мы закончили изложение материала по сложной и интересной теме – объектно-ориентированному программированию. К настоящему времени ООП из новой прогрессивной технологии превратилось в основной подход к разработке больших программных систем. Каждый программист сегодня должен владеть основными элементами этого подхода. К сожалению, нельзя объять необъятное, и наша книга, конечно, не претендует на абсолютную полноту изложения материала. Мы ставили своей целью рассмотреть основные элементы объектного подхода и их выражение в языке программирования Object Pascal. Некоторые частные возможности остались за кадром и могут быть изучены самостоятельно. К их числу относятся:

- ссылки на класс и виртуальные конструкторы;
- модификатор **`published`**, обработка событий (`events`);
- директива **`reintroduce`**.

Заключение

Всякое наше знание ограничено, лишь незнание бесконечно.

Пьер Симон Лаплас

Что ж! Пришла пора прощаться, уважаемый Читатель. С сожалением вынуждены констатировать: наше с Вами совместное путешествие подошло к концу.

В бурном море современных информационных технологий, как ни в какой другой области человеческой деятельности, справедлив принцип “все течет, все изменяется”. Причем изменяется настолько быстро, что любой, кто хочет считаться специалистом в данной сфере, должен постоянно идти вверх, только чтобы оставаться на месте, ну а чтобы действительно двигаться, приходится бежать со всей скоростью. Другими словами, современный IT-специалист это человек, постоянно пребывающей в состоянии изучения нового. Нам кажется, в этом состоит большой плюс нашей профессии!

Однако, несмотря на всю изменчивость и молодость компьютерного мира, уже существуют островки знания, признанные сообществом, как классические, без овладения которыми стать профессионалом невозможно. Мы с Вами на протяжении книги посетили и составили подробные карты нескольких таких островов: острова, где царят три алгоритмические конструкции; острова, где правят бал процедуры и функции, предпочитающие объединяться в библиотеки; и, наконец, острова, где в большом почете находится “его величество” Объект.

При этом главной нашей задачей было отнюдь не составление “географического” справочника, напротив, мы пытались показать, как на основе данных, извлекаемых путем анализа из постановки задачи, можно перекинуть через эти острова мост до самого главного – острова Работающей Программы. Насколько нам это удалось, судить Вам.

В качестве прощального слова, хотим выразить искреннюю надежду, что Вы продолжите знакомство с компьютерными науками, и что почерпнутые Вами из данной книги знания помогут Вам в этом процессе, а также пригодятся в дальнейшей профессиональной деятельности.

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.