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;

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

- 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.

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!

ГЛАВА 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 рекламировать себя 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 зависимости 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 стремится 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].

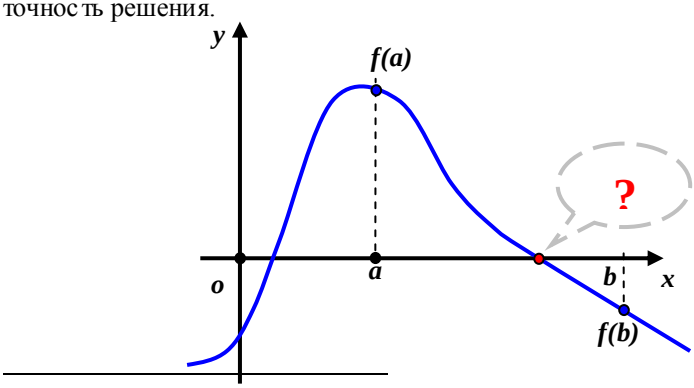
Из курса математики известно, что если функция $y = f(x)$ является *непрерывной* на $[a, b]$ и $f(a) \cdot f(b) < 0$, то эта функция имеет *корень* на $[a, b]$, т.е. такое $x = x_0$, что $x_0 \in [a, b]$, $f(x_0) = 0$.

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 $\varepsilon > 0$, 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.



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

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:

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

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.

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

Формула, задающая метод, выглядит следующим образом:

$$x_{n+1} = \frac{a_n + b_n}{2}, \tag{1.3}$$

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

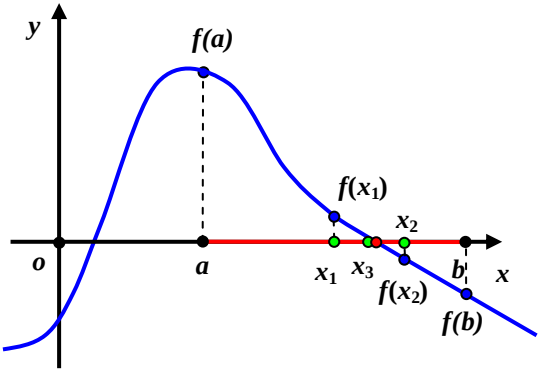


Рис. 1.2. Метод половинного деления

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 $[a_n, x_{n+1}]$ или $[x_{n+1}, b_n]$, 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.

1. 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 хотя 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, чт.∴ 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 перечислили 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 – *построению модели*.

2. 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.

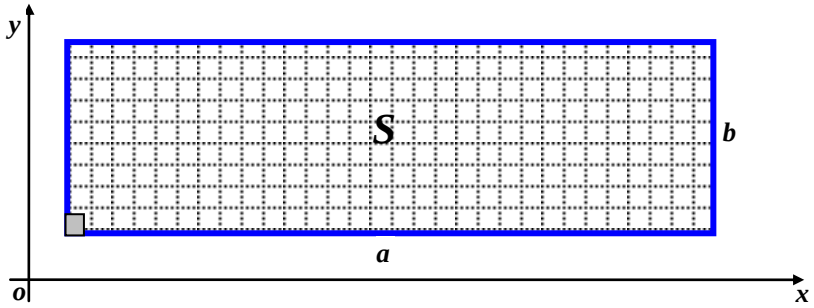


Рис. 1.3. Модель земельного участка прямоугольной формы

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

$$S = a \times b. \tag{1.4}$$

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

$$Rent = S \times Price. \tag{1.5}$$

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.

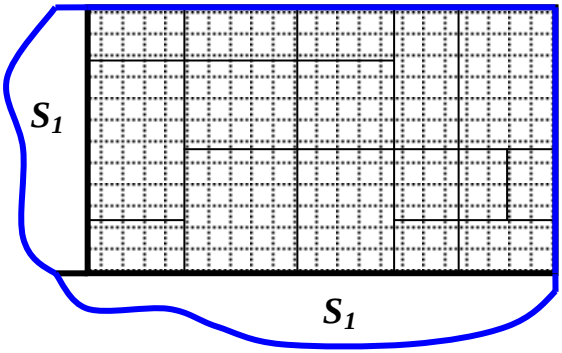


Рис. 1.4. Модель земельного участка общего вида

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

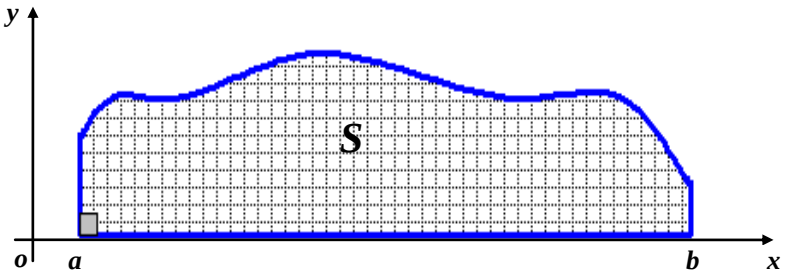


Рис. 1.5. Криволинейная трапеция

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):

$$S = \int_a^b f(x)dx \,. \tag{1.6}$$

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*.

3. 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.

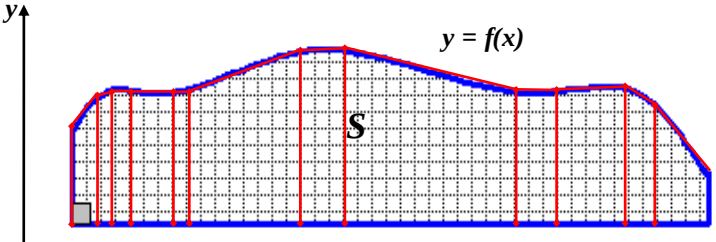
Со вторым вариантом несколько сложнее. Он тоже задается двумя формулами: (1.5) и (1.6), однако расчет площади по формуле (1.6) уже не так прост. Если для входящей в *исходные данные* функции $f(x)$

известно аналитическое (формульное) выражение, то можно попытаться проинтегрировать $f(x)$ 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.



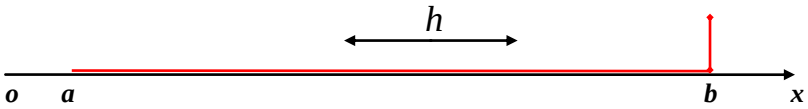


Рис. 1.6. Приближенное вычисление площади криволинейной трапеции

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*.

4. 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 о представлении 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 пойдет 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 использу-

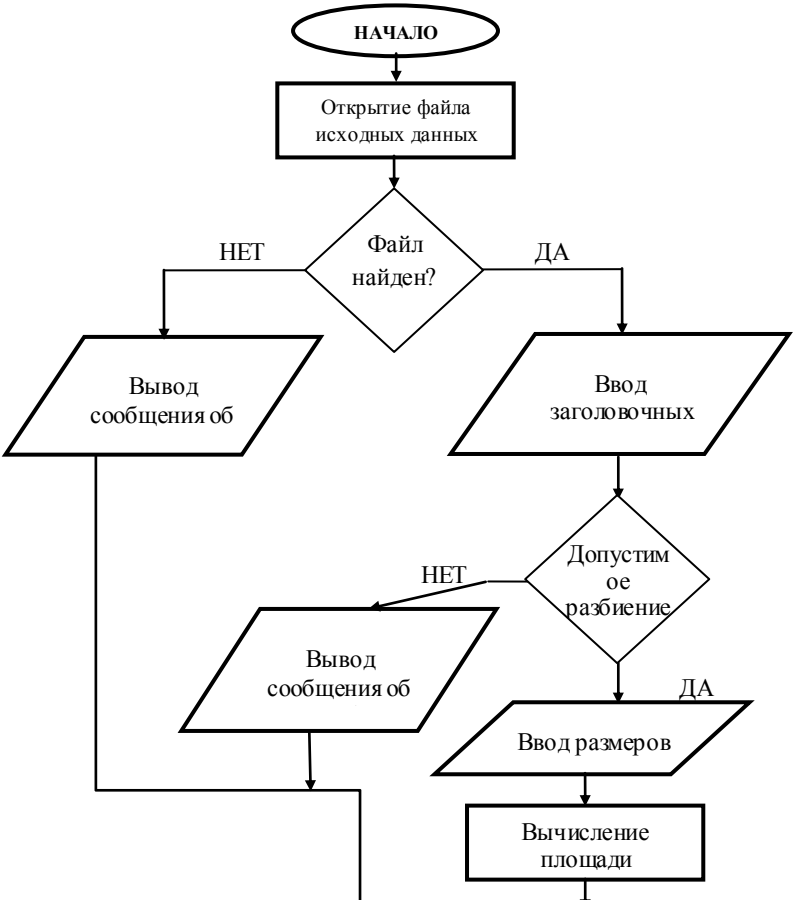


Рис. 1.7. Блок-схема алгоритма расчета арендной платы

ется *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. Пример такого укрупненного изображения алгоритма для нашей задачи приведен на рис. 1.7.

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 приемов 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 к 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.

5. 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 разобранной 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, заключенных 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 стоимость                }
   Cost: Real;        { (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.

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, успешная разработка 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*.

6. Отладка

Ошибки, возникающие в процессе создания программы, помимо упомянутой выше причины могут быть вызваны и неадекватным

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

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 как получить 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 безнадежное. Как определить, что программа имеет смысловую ошибку? В лучшем случае программа не работает, то есть ее работа прерывается в некоторый момент и система выдает какое-нибудь туманное сообщение типа «исчезновение порядка числа с плавающей точкой». В худшем случае программа успешно завершает свою работу и выдает результаты, отвечающие интуитивным представлениям о характере решения задачи, а о наличии ошибки в программе мы узнаем только после практического внедрения результатов, например когда по нашим прочностным расчетам 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 расчета для которых известен заранее, и выполнить программу с этими данными. Если полученный результат совпадает с известным, то, как говорят, «тест прошел». 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. Возьмем 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 задается 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 трудоемкой, следовательно, надо написать программу. Но ее тоже надо отладить! Замкнутый круг! Наконец, никто не может гарантировать, что мы не ошибемся и при аналитическом интегрировании.

Проблема получения тестовых данных настолько серьезна, 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 обновленные версии программы.

Принципиально другой подход к выяснению корректности программы состоит в методе формального доказательства ее 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».

7. 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 меняются 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.

8. 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 возвращение 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».

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.

ГЛАВА 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*.

1. О средствах разработки

Взглянем под другим углом зрения на рассмотренную в предыдущей главе схему. Весьма важен 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. 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 в виду – 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.

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*[3].

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

[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.

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

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.

2. 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.

2.1. 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 знают 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 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, существует 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:

[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.

- *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 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*.

2.2. 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.

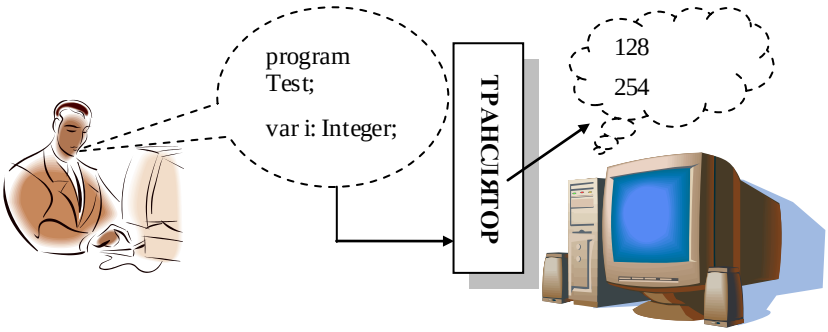


Рис. 2.1. Роль транслятора в создании программ

Основные достоинства компилируемых языков по сравнению с интерпретируемыми:

- 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).

2.3. Редактор связей

Уже в самом начале развития методов программирования стал применяться простой и эффективный приём выделения часто используемых алгоритмов в самостоятельные программы, получившие 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.

2.4. 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 и т.д.). Более того, теория в данной области не стоит на месте. Разработано несколько разных подходов к решению рассмотренных проблем. Здесь мы не будем подробно останавливаться на этих подходах, это предмет для серьезного разговора, заслуживающий отдельной книги.

2.5. 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.

3. Дополнительные средства разработки

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

3.1. Средства автоматизированной генерации кода

Помните, как, сидя в школе на уроке, вы мечтали о том, чтобы какой-нибудь джинн 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 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 следующее: в некоторых случаях современные системы

программирования способны автоматически создавать фрагменты текста программы.

3.2. 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.

3.3. 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].

3.4. 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 создания 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 на 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 – систему генерации проектной документации Rational SoDA [15].

4. 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. То есть 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 он 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

[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.

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.

5. 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 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.

6. 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.

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.

ГЛАВА 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 разработчиков 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-инструкций 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 инструментом «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, хотя 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.

1. 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 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,

---

[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.

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, чт.∴ 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 (например, 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 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).

2. 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 «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

[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.

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 небольшие 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 4 Гб.

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.

3. 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

[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

системы. Неторопли в (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 так же 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 важного объекта – *каталога* (папки). Каталог, так же 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

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».

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.

4. 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 *драйверы* – 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*.

5. Операционная система

Удобное взаимодействие 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 в

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 и объема 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.

6. 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 хотя разработка пользовательского интерфейса и обработка событий 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.

7. 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

[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.

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.

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.