ФГОС

11

К. Ю. Поляков
Е. А. Еремин

ИНФОРМАТИКА

2

УГЛУБЛЕННЫЙ УРОВЕНЬ

ФГОС

К. Ю. Поляков, Е. А. Еремин

ИНФОРМАТИКА

УГЛУБЛЕННЫЙ УРОВЕНЬ

Учебник для 11 класса

в 2-х частях

Часть 2

Рекомендовано
Министерством образования и науки
Российской Федерации
к использованию в образовательном процессе
в имеющих государственную аккредитацию
и реализующих образовательные программы
общего образования образовательных учреждениях

Москва
БИНОМ. Лаборатория знаний
2013

УДК 004.9
ББК 32.97
П54

Поляков К. Ю.

П54 Информатика. Углубленный уровень : учебник для 11 класса : в 2 ч. Ч. 2 / К. Ю. Поляков, Е. А. Еремин. — М. : БИНОМ. Лаборатория знаний, 2013. — 304 с. : ил.

ISBN 978-5-9963-1419-5 (Ч. 2)
ISBN 978-5-9963-1153-8

Учебник предназначен для изучения курса информатики на углубленном уровне в 11 классах общеобразовательных учреждений. Содержание учебника является продолжением курса 10 класса и опирается на изученный в 7–9 классах курс информатики для основной школы.

Рассматриваются вопросы передачи информации, информационные системы и базы данных, разработка веб-сайтов, компьютерное моделирование, методы объектно-ориентированного программирования, компьютерная графика и анимация.

Учебник входит в учебно-методический комплект (УМК), включающий в себя также учебник для 10 класса и компьютерный практикум.

Предполагается широкое использование ресурсов портала Федерального центра электронных образовательных ресурсов (http://fcior.edu.ru/).

Соответствует Федеральному государственному образовательному стандарту среднего (полного) общего образования (2012 г.).

УДК 004.9
ББК 32.97

Учебное издание

Поляков Константин Юрьевич
Еремин Евгений Александрович

ИНФОРМАТИКА.
УГЛУБЛЕННЫЙ УРОВЕНЬ
Учебник для 11 класса
В двух частях
Часть вторая

Ведущий редактор *О. Полежаева*
Ведущие методисты *И. Сретенская, И. Хлобыстова*
Обложка: *И. Марев*. Художественный редактор *Н. Новак*
Иллюстрации: *Я. Соловцова, Ю. Белаш*
Технический редактор *Е. Денюкова*. Корректор *Е. Клитина*
Компьютерная верстка: *В. Носенко*

Подписано в печать 28.03.13. Формат 70×100/16.
Усл. печ. л. 24,70. Тираж 10 000 экз. Заказ № 34085.
Издательство «БИНОМ. Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-5272, e-mail: binom@Lbz.ru
http://www.Lbz.ru, http://e-umk.Lbz.ru, http://metodist.Lbz.ru
При участии ООО Агентство печати «Столица»
www.apstolica.ru; e-mail: apstolica@bk.ru
Отпечатано в соответствии с качеством предоставленных издательством электронных носителей в ОАО «Саратовский полиграфкомбинат».
410004, г. Саратов, ул. Чернышевского, 59. www.sarpk.ru

ISBN 978-5-9963-1419-5 (Ч. 2)
ISBN 978-5-9963-1153-8
© БИНОМ. Лаборатория знаний, 2013

Оглавление

Глава 5. Элементы теории алгоритмов 5

§ 34. Уточнение понятия алгоритма 5

§ 35. Алгоритмически неразрешимые задачи 20

§ 36. Сложность вычислений 26

§ 37. Доказательство правильности программ 36

Глава 6. Алгоритмизация и программирование 49

§ 38. Целочисленные алгоритмы 49

§ 39. Структуры (записи) 57

§ 40. Динамические массивы 66

§ 41. Списки 73

§ 42. Стек, очередь, дек 82

§ 43. Деревья 95

§ 44. Графы 107

§ 45. Динамическое программирование 119

Глава 7. Объектно-ориентированное программирование 132

§ 46. Что такое ООП? 132

§ 47. Объекты и классы 135

§ 48. Создание объектов в программе 141

§ 49. Скрытие внутреннего устройства 147

§ 50. Иерархия классов 153

§ 51. Программы с графическим интерфейсом 167
§ 52. Основы программирования в RAD-средах 171
§ 53. Использование компонентов 178
§ 54. Совершенствование компонентов 187
§ 55. Модель и представление 192

Глава 8. Компьютерная графика и анимация 201
§ 56. Основы растровой графики 201
§ 57. Ввод изображений 205
§ 58. Коррекция фотографий 209
§ 59. Работа с областями 216
§ 60. Фильтры 220
§ 61. Многослойные изображения 222
§ 62. Каналы 227
§ 63. Иллюстрации для веб-сайтов 230
§ 64. Анимация 233
§ 65. Контуры 236

Глава 9. Трёхмерная графика 241
§ 66. Введение 241
§ 67. Работа с объектами 246
§ 68. Сеточные модели 251
§ 69. Модификаторы 257
§ 70. Кривые 262
§ 71. Материалы и текстуры 266
§ 72. Рендеринг 273
§ 73. Анимация 282
§ 74. Язык VRML 292

Глава 5
Элементы теории алгоритмов

§ 34
Уточнение понятия алгоритма

Зачем нужно определение алгоритма?

Как вы знаете, **алгоритмом** называют точный набор инструкций для исполнителя, который приводит к решению задачи за конечное время.

Особый интерес проявляли к алгоритмам математики. Один из древнейших известных алгоритмов — алгоритм Евклида для вычисления наибольшего общего делителя (НОД) двух натуральных чисел. Само слово «алгоритм» (от имени математика IX века аль-Хорезми, которого считают основателем алгебры) ввёл в науку в XVII веке немецкий математик Г. В. Лейбниц.

Долгое время считалось, что для любой математической задачи можно найти метод (алгоритм) решения, просто для ряда задач такие алгоритмы ещё не найдены. Эту идею высказал аль-Хорезми, такой же точки зрения придерживались и другие математики вплоть до начала XX века.

Однако, несмотря на все усилия, решить некоторые задачи не удавалось в течение столетий. Например, безуспешно закончились многочисленные попытки найти алгоритм доказательства правильности любой теоремы на основе заданной системы аксиом.

В 1931 г. австрийский математик К. Гёдель доказал теорему о неполноте, смысл которой состоит в том, что в любой достаточно сложной формальной системе, основанной на аксиомах (например, в арифметике, где введены натуральные числа и операции сложения и умножения), есть утверждение, которое невозможно ни доказать, ни опровергнуть в рамках этой системы. Поэтому было высказано предположение о том, что некоторые задачи **алгоритмически неразрешимы**, т. е. для них в принципе не существует алгоритма решения, и поэтому искать его бессмысленно. Чтобы строго доказать или опровергнуть эту гипотезу, нужно было ввести математическое понятие алгоритма.

«Определение», которое мы привели в начале главы, часто называют *интуитивным*, потому что оно содержит такие «немате-

матические» понятия, как «точный набор», «инструкция», «исполнитель», «решение задачи». Эти термины невозможно записать строго, используя язык математики и логики, поэтому для математического доказательства такое определение не подходит.

Исследования в этой области, которые начали активно проводиться в 30-х годах XX века, привели к возникновению **теории алгоритмов**, которая занимается:

- доказательством алгоритмической неразрешимости задач;
- анализом сложности алгоритмов;
- сравнительной оценкой качества алгоритмов.

Значительный вклад в развитие теории алгоритмов внесли математики А. Тьюринг (Великобритания), Э. Пост (США), А. Чёрч (Великобритания), С. Клини (США) и А. А. Марков (СССР).

Что такое алгоритм?

Первые известные алгоритмы — это правила выполнения арифметических действий с числами. В них чётко определены объекты (числа в десятичной записи) и элементарные шаги (сложить, вычесть, перемножить два однозначных числа — вспомните таблицы сложения и умножения). Постепенно сложность задач, которые решались с помощью алгоритмов, увеличивалась, и понятие «шаг алгоритма» оказалось нечётким, размытым. Например, можно ли считать элементарным шагом разложение числа на простые множители или сложение многозначных чисел?

Со временем понятие алгоритма расширилось — сейчас мы говорим об алгоритмах для исполнителей, которые работают с текстами и другими объектами реального мира. Однако оказалось, что все эти объекты можно тем или иным способом закодировать в виде цепочек символов, так что любой алгоритм сводится к преобразованию одной символьной строки в другую. Таким способом можно представить и классические вычислительные алгоритмы — операции с цифрами. В алгоритме шахматной игры объекты — это фигуры на доске, но их расположение легко закодировать в символьной форме (вспомните запись шахматных партий).

Поэтому можно рассматривать только алгоритмы обработки символьных строк, а полученные результаты будут применимы к любым алгоритмам. Как вы знаете, текст, записанный с помощью любого алфавита, всегда можно перевести в двоичный код, поэтому, вообще говоря, достаточно рассматривать только алгоритмы, работающие с двоичными последовательностями.

Про любой алгоритм можно сказать следующее:

- алгоритм получает на вход дискретный объект (например, слово);
- алгоритм обрабатывает входной объект по шагам (дискретно), строя на каждом шаге промежуточные дискретные объекты; этот процесс может закончиться или не закончиться;
- если выполнение алгоритма заканчивается, его результат — это объект, построенный на последнем шаге;
- если выполнение алгоритма не заканчивается (алгоритм зацикливается) или заканчивается аварийно (например, в результате деления на 0), то результат его работы при данном входе не определён.

Любой алгоритм рассчитан на определённого исполнителя: он должен использовать только понятные этому исполнителю команды. Задание для исполнителя — это текст на специальном (формальном) языке, который обычно называют программой. Поэтому можно определить алгоритм как программу для некоторого исполнителя.

Напомним, что, с точки зрения теории алгоритмов, достаточно рассматривать только алгоритмы, работающие с цепочками символов, которые называют **словами** (рис. 5.1).

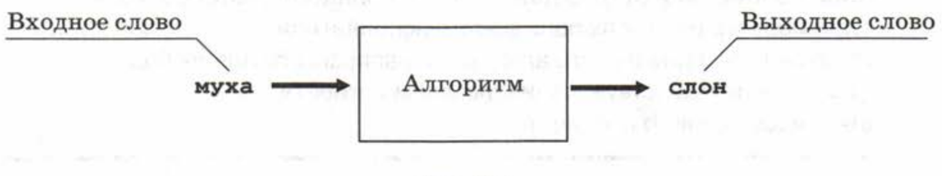


Рис. 5.1

Каждый алгоритм **задаёт (вычисляет) функцию**, которая преобразует входное слово в результат (выходное слово). Такая функция может быть не определена для некоторых входных слов, если алгоритм зацикливается.

Функция, заданная алгоритмом, может быть нигде не определена. Например, алгоритм

```
нц пока да
кц
```

зацикливается при любом входном слове.

Алгоритмы называются **эквивалентными**, если они задают одну и ту же функцию. То есть при любом входном слове оба алгоритма должны приводить к одному и тому же результату или зацикливаться (оба алгоритма не выдают никакого результата). Например, следующие алгоритмы для выбора минимального из значений переменных a и b эквивалентны:

```
если a<b то
  M:=a
иначе
  M:=b
все
```

```
M:=b
если a<b то
  M:=a
все
```

Универсальные исполнители

Как мы уже видели, понятие алгоритма оказывается «привязанным» к его исполнителю и некоторому языку программирования. Это не позволяет определить алгоритм как математический объект. Поэтому возникла идея попытаться построить **универсальный исполнитель.**

Универсальный исполнитель — это исполнитель, который может моделировать работу любого другого исполнителя.
Это значит, что для любого алгоритма, написанного для любого исполнителя, существует эквивалентный алгоритм для универсального исполнителя.

Такой исполнитель можно было бы использовать для доказательства разрешимости или неразрешимости задач. Если удаётся построить алгоритм решения задачи для универсального исполнителя, то задача разрешима. Если доказано, что алгоритм не существует, то задача неразрешима. Система команд такого исполнителя должна быть как можно проще — так его будет легче использовать в доказательствах.

В середине XX века разными учёными независимо друг от друга были предложены несколько исполнителей, претендующих на роль универсальных (они будут рассмотрены далее), причём в теории алгоритмов доказано, что все они эквивалентны друг другу. Это означает, что для любого алгоритма для одного уни-

версального исполнителя можно построить эквивалентный алгоритм для другого универсального исполнителя.

Как же связан универсальный исполнитель с проблемой строгого определения алгоритма?

Любой алгоритм может быть представлен как программа для универсального исполнителя.

Это основная идея теории алгоритмов. Строго доказать это утверждение невозможно, потому что здесь используется интуитивное понятие «алгоритм».

Как мы увидим, каждый универсальный исполнитель описывается с помощью математических терминов, поэтому на его основе можно дать строгое определение алгоритма:

Алгоритм — это программа для универсального исполнителя.

Универсальный исполнитель — это некоторая **модель вычислений**, которая задаёт способ описания алгоритмов и их выполнения. Модель вычислений должна содержать:

- «процессор», задающий систему команд и способ их выполнения;
- «память», определяющую способ хранения данных;
- язык программирования (способ записи программ);
- способ ввода данных (чтения входного слова);
- способ вывода слова-результата.

Все универсальные исполнители эквивалентны, поэтому последнее приведённое определение алгоритма не зависит от конкретного исполнителя.

Машина Тьюринга

Первым предложил универсальный исполнитель английский математик Алан Тьюринг. Придуманное им воображаемое устройство состоит из трёх частей:

- *бесконечной ленты*, разделённой на ячейки;
- *каретки* (читающей и записывающей головки);
- *программируемого автомата*.

А. Тьюринг (1912–1956)

Программируемый автомат управляет кареткой, посылая ей команды в соответствии с заложенной в него сменяемой программой. Лента выполняет роль памяти компьютера, автомат — роль процессора, а каретка служит для ввода и вывода данных. Такое устройство называют **машиной Тьюринга**.

Теоретически лента в машине Тьюринга бесконечна, однако в каждый момент времени работы машины используется лишь конечная её часть.

Каретка в любой момент времени находится над одной ячейкой, автомат может читать и изменять содержимое этой ячейки, которая называется **текущей (рабочей) ячейкой** (рис. 5.2).

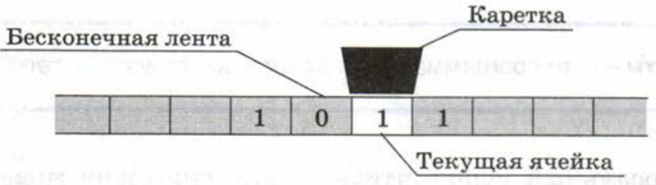


Рис. 5.2

В каждую ячейку ленты можно записать один любой символ, принадлежащий выбранному алфавиту. Любой алфавит обязательно содержит пробел (пустой символ, соответствующий «чистым» участкам ленты), который мы будем обозначать знаком □. Алфавит обычно обозначается буквой A, а его элементы — строчными буквами a с индексами: $A = \{a_1, a_2, \ldots, a_N\}$. Например, алфавит машины Тьюринга, работающей с двоичными числами, задаётся в виде $A = \{0, 1, \square\}$.

Непрерывную цепочку символов на ленте называют **словом**. На рисунке 5.2 лента содержит слово «1011», которое можно воспринимать как двоичное число.

Автоматом называют устройство, работающее без участия человека. Автомат в машине Тьюринга имеет несколько **состояний** и при определённых условиях переходит из одного состояния в другое. Состояние автомата определяет ту промежуточную задачу,

которую решает автомат в данный момент. Это напоминает состояния человека: ночью он спит (состояние 1), утром встаёт и умывается (состояние 2), завтракает (состояние 3), идёт на работу (состояние 4) и т. д.

Множество всех состояний автомата обозначается буквой Q, а его элементы — строчными буквами q с индексами: $Q=\{q_1, q_2, ..., q_M\}$. Принято, что в начальный момент машина Тьюринга находится в состоянии q_1.

Особое состояние q_0 — это **состояние останова**. Если машина переходит в это состояние, выполнение программы сразу останавливается.

Автомат управляется **программой**. Во время каждого шага программы автомат выполняет последовательно три действия:

1) изменяет символ в рабочей ячейке на другой (или оставляет без изменений);
2) перемещает каретку влево или вправо (или оставляет на месте);
3) переходит в другое состояние (или остаётся в прежнем состоянии).

Поэтому при составлении программы для каждой пары (*символ, состояние*) нужно определить три параметра: символ a_i из выбранного алфавита A, направление перемещения каретки (← — влево, → — вправо, точка — нет перемещения) и новое состояние автомата q_k. Например, команда $1 \leftarrow q_2$ обозначает «заменить символ на 1, переместить каретку влево на 1 ячейку и перейти в состояние q_2».

Пример 1. На ленте записано число в двоичной системе счисления. Каретка находится где-то над числом. Требуется увеличить число на единицу.

Для того чтобы построить машину Тьюринга, нужно:

- определить алфавит машины Тьюринга A;
- выделить простейшие подзадачи и определить набор возможных состояний Q; задать начальное состояние q_1 и конечное состояние q_0 (в котором машина останавливается);
- составить программу, т. е. для каждой пары (a_i, q_k) определить команду, которую должен выполнить автомат.

Как мы уже выяснили, алфавит машины Тьюринга, работающей с двоичными числами, включает символы 0, 1 и пробел: $A=\{0, 1, \square\}$. Определим возможные состояния (разобьём задачу на элементарные подзадачи):

1) q_1 — автомат ищет правый конец слова (числа) на ленте;
2) q_2 — автомат увеличивает число на 1, проходя его справа налево, и останавливается, закончив работу.

Теперь займёмся программой. На первом этапе, когда автомат ищет конец слова, его работа может быть описана так:

1) если в рабочей ячейке записана цифра 0, переместиться вправо;
2) если в рабочей ячейке записана цифра 1, переместиться вправо;
3) если в рабочей ячейке пробел, переместить каретку влево и перейти в состояние q_2.

Тогда действия автомата в состоянии q_1 можно представить в виде таблицы, где в заголовках строк записываются символы алфавита, а в заголовках столбцов — состояния (рис. 5.3).

	q_1
0	$0 \to q_1$
1	$1 \to q_1$
□	$\square \leftarrow q_2$

Рис. 5.3

Заметим, что во всех случаях символ под кареткой не меняется. Кроме того, состояние меняется только в последней ячейке. Поэтому для упрощения записи не будем указывать в таблице то, что остаётся без изменений. Так, на наш взгляд, более кратко и понятно (рис. 5.4).

	q_1
0	$\to$
1	$\to$
□	$\leftarrow q_2$

Рис. 5.4

Второй этап — увеличение двоичного числа на единицу. Это можно сделать следующим способом (вспомните тему «Системы счисления»):

1) если в рабочей ячейке записана цифра 0, записать в неё 1 и стоп;
2) если в рабочей ячейке записана цифра 1, выполнить перенос в старший разряд — записать в ячейку 0 и переместиться влево;
3) если в рабочей ячейке пробел, записать в неё 1 и стоп.

Для того чтобы остановить работу машины Тьюринга, нужно перевести её в состояние останова q_0. Теперь можно добавить в таблицу столбец, соответствующий состоянию q_2 (рис. 5.5).

	q_1	q_2
0	$\rightarrow$	$1 \, . \, q_0$
1	$\rightarrow$	$0 \leftarrow$
$\square$	$\leftarrow q_2$	$1 \, . \, q_0$

Рис. 5.5

Построенная полная таблица (см. рис. 5.5) — это и есть **программа для машины Тьюринга**. Обратите внимание, что мы разбили исходную задачу на подзадачи, для каждой из них составили программу, а потом их соединили. Две подзадачи связаны через ячейку ($\square$, q_1), в которой состояние автомата изменяется на q_2. В данном простейшем случае в каждом из двух алгоритмов было использовано только одно состояние, но это не обязательно — можно таким же способом соединять и более сложные алгоритмы. Если алгоритмы *А* и *Б* можно запрограммировать на машине Тьюринга, то и любую их комбинацию тоже можно запрограммировать.

Тьюринг предположил, что:

Любой алгоритм (в интуитивном смысле этого слова) может быть представлен как программа для машины Тьюринга.

Это утверждение в теории алгоритмов известно как **тезис Чёрча–Тьюринга**.

Машина Тьюринга может быть строго задана с точки зрения математики. Алфавит A и набор возможных состояний Q могут быть записаны в виде множеств, а программа — в виде пятёрок вида (a_i, q_k, a_j, d_{ik}, q_m), задающих команду «если машина находится в состоянии q_k и в рабочей ячейке записан символ a_i, то записать в рабочую ячейку символ a_j, сместиться в направлении d_{ik} и перейти в состояние q_m». Например, приведённая выше программа увеличения двоичного числа на 1, записанная в виде таких пятёрок, выглядит так:

$(0, q_1, 0, \rightarrow, q_1), (1, q_1, 1, \rightarrow, q_1), (\square, q_1, \square, \leftarrow, q_2),$
$(0, q_2, 1, ., q_0), (1, q_2, 0, \leftarrow, q_2), (\square, q_2, 1, ., q_0).$

Эта машина — математический объект, и данное на её основе определение алгоритма может использоваться для доказательств. Едва ли можно применить машину Тьюринга для решения практических задач, но эта простая модель алгоритма очень удобна для проведения теоретических исследований. В отличие от интуитивного определения алгоритма новое определение не содержит таких неопределённых понятий, как «инструкция», «исполнитель», «решение задачи». Таким образом, формальное определение слова «алгоритм» (по Тьюрингу) выглядит так: алгоритм — это программа для машины Тьюринга.

Машина Поста

Практически одновременно с Тьюрингом (в том же 1936 г.) и независимо от него американский математик Э. Л. Пост предложил ещё более простую систему обработки данных, на основе которой позднее была построена так называемая **машина Поста.**

Э. Л. Пост (1897–1954)

Лента в машине Поста (так же как и в машине Тьюринга) бесконечна и разбита на ячейки. Каждая ячейка может содержать метку (быть отмечена) или не содержать её (пустая ячейка) (рис. 5.6).

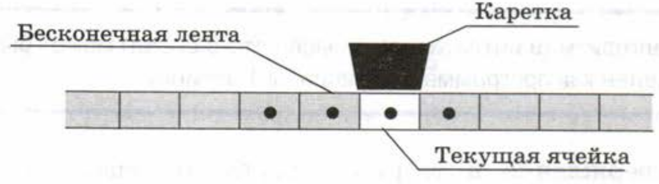


Рис. 5.6

Таким образом, Пост сократил алфавит всего до двух цифр. Это допустимо, потому что любые данные можно перекодировать в двоичный код, сопоставив каждой букве исходного алфавита уникальную последовательность нулей и единиц.

Кроме того, алгоритм работы машины Поста задаётся не в виде таблицы, а как программа, состоящая из отдельных команд. Система команд машины Поста содержит только 6 команд:

← — переместить каретку на 1 ячейку влево;
→ — переместить каретку на 1 ячейку вправо;
0 — стереть метку в рабочей ячейке (записать 0);

1 — поставить метку в рабочей ячейке (записать 1);
? n_0, n_1 — если в рабочей ячейке нет метки, перейти к строке n_0, иначе перейти к строке n_1;
стоп — остановить машину.

Попытка стереть метку там, где её нет, или поставить метку повторно считается ошибкой, и машина аварийно останавливается.

Все строки в программе нумеруются по порядку, это необходимо для работы команды ветвления (? n_0, n_1). С помощью этой команды можно также строить циклы как с предусловием, так и с постусловием. Например, следующая программа перемещает каретку влево до первой отмеченной ячейки:

1. ←
2. ? 1, 3
3. стоп

Если после выполнения команды ←, →, 0 или 1 требуется перейти не на следующую строку, а на какую-то другую, то номер этой строки можно записать в конце команды. Например, команда

← 3

означает «переместить каретку влево и перейти на строку 3».

При работе с машиной Поста числа обычно записывают в унарной (единичной) системе счисления, в виде непрерывной цепочки меток нужной длины (вспомните счётные палочки в младшей школе). Например, на ленте, показанной на рис. 5.6, записано число 4.

Пост предположил, что любой алгоритм может быть записан как программа для предложенного им исполнителя. В теории алгоритмов доказано, что машины Поста и Тьюринга одинаковы по своим возможностям. Это значит, что круг задач, который они решают, тоже одинаков.

Нормальные алгорифмы Маркова

Советский математик А. А. Марков, который в середине XX века изучал разрешимость некоторых задач алгебры, предложил новую модель вычислений, которую он назвал нормальными алгорифмами.

А. А. Марков (младший) (1903–1979)

Нормальные алгорифмы Маркова (НАМ) — это строгая математическая форма записи алгорит-

мов обработки символьных строк, которую можно использовать для доказательства разрешимости или неразрешимости различных задач. Марков предположил, что любой алгоритм можно записать как НАМ. В отличие от машин Тьюринга и Поста НАМ — это «чистый» алгоритм, который не связан ни с каким «аппаратным обеспечением» (лентой, кареткой и т. п.).

НАМ преобразует одно слово (цепочку символов некоторого алфавита) в другое и задаётся алфавитом и системой подстановок. Заметьте, что в жизни мы нередко применяем такие замены. Например, при умножении в столбик мы не вычисляем каждый раз произведение $7 \cdot 8$, а просто помним, что оно равно 56.

Пример 1. Пусть алфавит НАМ — это русские буквы и задана система подстановок:

а → н
ух → ло
м → с

Применим эту систему подстановок к начальному слову «муха». Подстановки нужно просматривать по порядку, начиная с первой. Первая подстановка означает: «если в слове есть буквы "а", заменить первую букву "а" на букву "н"». В слове «муха» есть буква «а», поэтому заменяем её на «н». Получается «мухн».

Начинаем просмотр подстановок сначала. Букв «а» больше нет, поэтому переходим ко второй подстановке. Сочетание «ух» есть в слове «мухн», поэтому вторая подстановка срабатывает, и мы заменяем «ух» на «ло»: получается «млон».

Теперь ни первая, ни вторая подстановки не применимы, а использование третьей даёт в результате слово «слон». Больше ни одну подстановку сделать нельзя, и НАМ заканчивает работу. Таким образом, приведённая система подстановок преобразует слово «муха» в слово «слон».

При поиске образца рабочая цепочка символов просматривается с начала. Если в строке слово-образец встречается несколько раз, то за один шаг заменяется только первое из них. Так как на следующем шаге просмотр опять начинается с начала цепочки, после первой выполненной замены может «сработать» совсем другая подстановка.

В записи подстановок слово-образец может быть пустым, в этом случае слово-замена приписывается в начало рабочей строки:

→ 0

Такая подстановка всегда должна быть последней в списке, иначе программа зациклится: в начало слова будут постоянно дописываться всё новые и новые нули.

Если после слова-замены стоит точка, после выполнения такой подстановки работа программы заканчивается. Например, если применить НАМ

a → o.

к слову «карова», то в результате получим «корова», потому что после первого же действия работа программы закончится, и последняя буква не будет заменена.

Пример 2. Построим НАМ для следующей задачи: удалить из строки, состоящей из букв «a» и «b», первый символ. Например, строка «abba» должна быть преобразована в «bba». Казалось бы, здесь нужно использовать систему подстановок:

a → .
b → .

Однако такой НАМ будет неправильно работать для слов, начинающихся с буквы «b», например для слова «bba», в котором будет удалена последняя буква, потому что первая подстановка выполнится раньше, чем вторая. Перестановка двух строк также не даёт решения — теперь алгоритм неправильно работает для слов, начинающихся с буквы «a». Чтобы решить эту задачу, в алфавит НАМ добавляют еще один специальный символ, например символ «*». Этим символом помечают начало слова, используя подстановку.

→ *

Полный алгоритм выглядит так:

*a → .
*b → .
→ *

Сначала срабатывает третья подстановка (ставим «*» в начало строки), затем, в зависимости от первой буквы исходного слова, работает первая или вторая подстановка, и алгоритм заканчивает работу. Дополнительный символ похож на маркер в текстовом редакторе — он отмечает место в тексте, с которым потом будут выполняться какие-то действия.

Как показано в теории алгоритмов, любой алгоритм для машин Тьюринга и Поста можно записать как НАМ и наоборот.

Поэтому все три рассмотренных подхода к строгому определению понятия «алгоритм» эквивалентны (равносильны).

Вопросы и задания

1. Зачем понадобилось уточнять понятие «алгоритм»?
2. Какие задачи рассматриваются в теории алгоритмов?
3. Почему можно ограничиться алгоритмами обработки символьных строк? Можно ли рассматривать только алгоритмы для преобразования двоичных кодов?
4. Как вы понимаете утверждение «Алгоритм задаёт некоторую функцию»?
5. Как связаны понятия «алгоритм» и «исполнитель»?
6. Что такое программа?
7. В каком случае говорят, что два алгоритма эквивалентны?
8. Что такое универсальный исполнитель?
9. Сравните интуитивное и строгое понятия алгоритма.
10. Опишите устройство и систему программирования машины Тьюринга.
11. Что такое состояние машины Тьюринга?
12. Сопоставьте устройство машины Тьюринга с устройством компьютера. Какие устройства машины Тьюринга выполняют те же функции, что и аналогичные устройства компьютера?
13. В чем особенность состояний q_0 и q_1 машины Тьюринга?
14. По какому принципу можно построить программу для машины Тьюринга, которая последовательно выполняет операции А и Б?
15. Сформулируйте тезис Чёрча–Тьюринга.
16. Сравните машины Тьюринга и Поста.
17. Зачем нумеруются строки в программе для машины Поста?
18. Что такое нормальный алгорифм Маркова?
19. Зачем используют специальные символы в НАМ?
20. Что означает эквивалентность различных универсальных исполнителей?

Подготовьте сообщение

а) «Какие бывают машины Тьюринга?»
б) «Эзотерические языки программирования»
в) «Рекурсивные функции»

Задачи

1. Что делают следующие программы для машины Тьюринга?

а)

	q_1
0	←
1	←
□	→ q_0

б)

	q_1
0	→ q_0
1	→ q_0
□	←

в)

	q_1	q_2
а	q_2	□ ←
б	q_2	□ ←
□	←	q_0

В каких случаях эти программы зацикливаются?

2. Предложите программу для машины Тьюринга и начальное состояние ленты, при котором эта программа зацикливается.
3. Составьте программу для машины Тьюринга, которая уменьшает двоичное число на 1.
4. Составьте программы для машины Тьюринга, которые увеличивают и уменьшают на единицу число, записанное в десятичной системе счисления.
5. Составьте программу для машины Тьюринга, которая складывает два числа в двоичной системе, разделенные на ленте знаком «+».
6. Составьте программы для машины Тьюринга, которые выполняют сложение и вычитание двух чисел в десятичной системе счисления.
7. Что делают следующие программы для машины Поста?

а)
1. 1
2. →
3. → 1

б)
1. →
2. ? 3,4
3. 1 1
4. стоп

в)
1. ? 2,3
2. 1 4
3. → 1
4. стоп

Как будет работать каждая из программ при различных начальных состояниях ленты?

8. Напишите программу для машины Поста, которая увеличивает (уменьшает) число в единичной системе счисления на единицу. Каретка расположена слева от числа.
9. Напишите программу для машины Поста, которая складывает два числа в единичной системе счисления. Каретка расположена над пробелом, разделяющим эти числа на ленте.
10. Что делают следующие НАМ, если применить их к символьной цепочке, состоящей из нулей и единиц?

а)
0 → 00
1 → 11

б)
0 → 0
1 → 1
* → =.
→ *

в)
0 → 00
1 → 11
* → .
→ *

Как будет работать каждая из программ при различных начальных состояниях ленты?

11. Напишите НАМ, который сортирует цифры двоичного числа так, чтобы сначала стояли все нули, а потом — все единицы.
12. Дополните приведённый в параграфе НАМ для удаления первого символа строки так, чтобы он не зацикливался на пустом слове.
13. Напишите НАМ, который умножает двоичное число на 2, добавляя 0 в конец записи числа.

§ 35
Алгоритмически неразрешимые задачи

Вычислимые и невычислимые функции

Мы уже говорили, что любой алгоритм задаёт некоторую функцию, которая для каждого входного слова, к которому применим алгоритм, однозначно задаёт результат — выходное слово. Такие функции называются вычислимыми.

Вычислимая функция — это функция, для вычисления которой существует алгоритм.

Любая вычислимая функция может задаваться разными алгоритмами (разными программами для выбранного универсального исполнителя). Например, следующие два нормальных алгорифма Маркова заменяют во входном двоичном слове все буквы «а» на нули и все буквы «б» на единицы:

а → 0	б → 1
б → 1	а → 0

Любая вычислимая функция может быть вычислена с помощью любого универсального исполнителя: машин Тьюринга и Поста, нормальных алгорифмов Маркова и др.

Рассмотрим, например, такую функцию, определённую для всех натуральных чисел:

$$f(n) = \begin{cases} 1, \text{ если } n \text{ — чётное;} \\ 0, \text{ если } n \text{ — нечётное.} \end{cases}$$

Попробуем составить программу для машины Тьюринга, которая вычисляет эту функцию. Будем считать, что число записано в единичной системе счисления (в виде цепочки единиц[1]), и каретка в начальный момент стоит над самой левой единицей. Оказывается, такая программа действительно существует (рис. 5.7).

	q_1	q_2	q_3	q_4
1	$\rightarrow q_2$	$\rightarrow q_1$	$\leftarrow q_4$	$\square \leftarrow$
$\square$	$\leftarrow q_3$	$\leftarrow q_4$		q_0

Рис. 5.7

Как принято, в начальный момент машина находится в состоянии q_1. Затем она движется вправо вдоль числа, поочередно переходя из состояния q_1 (пройдено чётное число единиц) в состояние q_2 (пройдено нечётное число единиц) и обратно. Таким образом, если встречен пробел и машина находится в состоянии q_2, то число нечётное и нужно просто стереть все единицы (состояние q_4). Если машина закончила просмотр в состоянии q_1, то число чётное; при этом нужно оставить одну единицу (состояние q_3) и перейти в состояние q_4 (стереть все остальные единицы). Обратите внимание, что ячейка ($\square$, q_3) в таблице пустая – это невозможное состояние (покажите это самостоятельно).

Таким образом, рассмотренная функция вычислима, т. е. её можно вычислять с помощью машины Тьюринга, а значит, и с помощью любого универсального исполнителя. Например, нормальный алгорифм Маркова для алфавита $A = \{1\}$ выглядит так:

$11 \rightarrow$ ""
$1 \rightarrow .$
$\rightarrow 1.$

В первой подстановке две соседние единицы удаляются (слово-замена здесь пустое, для ясности оно взято в кавычки, которыми можно ограничивать слова в НАМ). Это происходит до тех пор, пока не будут удалены все пары, поскольку эта подстановка стоит первой. Если остаётся одна единица, она удаляется с помощью второй подстановки, и работа программы заканчивается. Если все единицы удалены (число чётное), то с помощью третьей подстановки мы ставим одну единицу и останавливаем автомат.

[1] Пустая лента соответствует числу 0.

Существуют и **невычислимые функции**. Рассмотрим простой пример, предложенный В. А. Успенским в книге «Машина Поста». Известно, что математическая постоянная π — иррациональное число, его десятичная запись бесконечна и непериодична. Введем функцию $h(n)$, которая для любого натурального числа n равна 1, если в десятичной записи числа π есть n стоящих подряд девяток, окружённых другими цифрами, и равна нулю, если такой цепочки девяток нет. Как вычислить значение этой функции при некотором заданном n? Конечно, можно вычислять друг за другом десятичные знаки числа π (такие алгоритмы математикам известны!) и проверять, не нашлась ли в полученной последовательности цифр цепочка из n девяток. С помощью такого «наивного» алгоритма можно найти такие значения n, при которых $h(n) = 1$: обнаружив требуемую цепочку, алгоритм закончит работу. Например, анализ первых 800 знаков показывает, что $h(n) = 1$ при $n = 0, 1, 2, 6$. Но если для какого-то n функция $h(n)$ равна нулю, то «наивный» алгоритм никогда не остановится. Более того, для этой функции вообще не существует алгоритма, который при любом n останавливается и выдает значение $h(n)$ в качестве результата. Поэтому такая функция невычислима.

Когда задача алгоритмически неразрешима?

Как вы знаете, невозможно создать вечный двигатель, потому что это противоречит универсальным физическим законам сохранения. Точно так же в математике и информатике существуют задачи, для которых решение в общем виде отсутствует.

Поскольку алгоритм работает только с дискретными объектами, любая алгоритмическая задача — это функция, заданная на множестве дискретных объектов (входных слов).

Пусть, например, требуется по шахматной позиции определить, кто выигрывает при правильной игре — белые, чёрные или будет ничья. Построим функцию, соответствующую этому алгоритму. Для этого выберем способ кодирования, при котором каждая позиция может быть закодирована словом (символьной строкой) v в подходящем алфавите. Тогда приведённой задаче может соответствовать функция $f(v)$, заданная на множестве таких слов:

$$f(v) = \begin{cases} \text{'Б'}, & \text{если } v \text{ — код позиции, в которой выигрывают белые,} \\ \text{'Ч'}, & \text{если } v \text{ — код позиции, в которой выигрывают чёрные,} \\ \text{'0'}, & \text{если } v \text{ — код позиции, в которой будет ничья,} \\ \text{'?'}, & \text{если } v \text{ — ошибочный код позиции.} \end{cases}$$

Если функция, соответствующая задаче, вычислима, то задача называется **алгоритмически разрешимой** — для её вычисления можно построить алгоритм. Если определённая в задаче функция невычислима, то алгоритма для её решения не существует.

Алгоритмически неразрешимая задача — это задача, соответствующая невычислимой функции.

В 1900 г. на Международном математическом конгрессе в Париже известный математик Давид Гильберт сформулировал 23 нерешённые математические проблемы[1]. В знаменитой «десятой проблеме Гильберта» требуется найти метод, который позволяет определить, имеет ли заданное алгебраическое уравнение с целыми коэффициентами решение в целых числах. Например, уравнение

$$x^2 + y^3 + 2 = 0$$

Ю. В. Матиясевич (род. в 1947)

имеет два целочисленных решения, (5; –3) и (–5; –3). Сложность состояла в том, что требовалось найти единый метод (алгоритм), позволяющий решить задачу для *любого* такого уравнения со многими неизвестными.

В начале XX века была уверенность, что такой алгоритм есть, и поэтому его упорно искали. Однако в 1970 г. советскому математику Ю. В. Матиясевичу удалось доказать, что общего алгоритма решения этой задачи не существует.

Немецкий математик Г. В. Лейбниц в XVII веке безуспешно пытался найти метод проверки правильности любых математических утверждений. Как вы знаете, почти все математические теории основаны на использовании *аксиом* (положений, принимаемых без доказательства), из которых выводятся все остальные утверждения (*теоремы*). Задача заключалась в том, чтобы разработать алгоритм, позволяющий установить, можно ли вывести формулу Б из формулы А в рамках заданной системы аксиом

[1] Сейчас большинство из них решено полностью или частично.

А. Чёрч
(1903–1995)

(*«проблема распознавания выводимости»*). В 1936 г. американский математик А. Чёрч доказал, что эта задача *в общем виде* алгоритмически неразрешима, поэтому нельзя сформулировать универсальный алгоритм, пригодный для доказательства любой теоремы[1].

Таким образом, уточнённые определения алгоритма, основанные на понятии универсальных исполнителей, сыграли в науке очень важную роль — позволили получить *отрицательные результаты*, т. е. доказать, что алгоритмов решения некоторых задач в общем виде не существует.

Для того чтобы доказать неразрешимость какой-то новой задачи, пытаются свести её к уже известным алгоритмически неразрешимым задачам. Если это удаётся, значит, и новая задача алгоритмически неразрешима[2].

Существуют также задачи, про которые неизвестно, алгоритмически разрешимы они или нет — решение не найдено, но алгоритмическая неразрешимость не доказана.

Алгоритмически неразрешимые задачи встречаются не только в математике, но и в информатике, например при разработке программ. Оказывается, невозможно написать программу для машины Тьюринга (алгоритм), которая по тексту любой программы P и её входным данным X определяет, завершается ли программа P при входе X за конечное число шагов или зацикливается. Это так называемая **проблема останова**. Её неразрешимость означает, в частности, что нельзя полностью автоматизировать тестирование любых программ, поручив это компьютеру. Однако для некоторых классов алгоритмов проблему останова решить можно. Например, линейная программа, не содержащая ветвлений и циклов, всегда завершится.

Было доказано, что алгоритмически неразрешима **проблема эквивалентности:** по двум заданным алгоритмам определить, будут ли они выдавать одинаковые результаты для любых допустимых исходных данных. Следовательно, невозможно полностью

---

[1] Тем не менее отдельные классы теорем можно доказывать на компьютере.

[2] Допустим, что (1) задача *А* неразрешима и (2) если мы можем построить алгоритм для решения задачи *Б*, то с его помощью можно построить алгоритм решения задачи *А*. Тогда задача *Б* тоже неразрешима.

автоматизировать решение многих важных задач, связанных с разработкой программ, например:

- по заданному тексту программы определить, что она «делает»;
- определить, правильно ли работает программа при любых допустимых исходных данных;
- найти ошибку в программе, работающей неправильно.

Поэтому при отладке программы большую роль играет интуиция. Помогают (но не решают проблему полностью!) стандартные приёмы, позволяющие найти ошибку:

- сравнение результатов работы программы с результатами ручного счёта;
- эксперименты с программой при различных исходных данных для того, чтобы выявить закономерность появления ошибок;
- временное отключение (комментирование) частей программы и др.

Поскольку многие этапы разработки программного обеспечения в принципе невозможно представить в виде алгоритмов, программирование остаётся работой человека. Полностью поручить его компьютеру не удаётся, хотя решение некоторых задач всё же можно автоматизировать.

Вопросы и задания

1. Что такое вычислимая функция?
2. Приведите пример невычислимой функции.
3. Что такое алгоритмически неразрешимые задачи? Приведите известные вам примеры.
4. Что такое проблема останова? Каковы её следствия?
5. Что такое проблема эквивалентности?
6. Как можно доказать алгоритмическую неразрешимость новой задачи?

Задачи

1. В качестве доказательства того, что следующая функция вычислима, напишите программу для машины Поста.

$$f(n) = \begin{cases} 1, \text{ если } n \text{ — чётное;} \\ 0, \text{ если } n \text{ — нечётное.} \end{cases}$$

2. Докажите, что следующая функция вычислима:

$$f(n) = \begin{cases} 1, \text{ если } n \text{ делится на 3;} \\ 0, \text{ если } n \text{ не делится на 3.} \end{cases}$$

В качестве доказательства напишите программы для машин Тьюринга и Поста, а также НАМ.

*3. Первой задачей, неразрешимость которой была доказана, была **проблема самоприменимости:** по заданному тексту программы P определить, останавливается ли программа P, если ей на вход подать текст этой же программы. Докажите, что проблема останова сводится к проблеме самоприменимости (именно так и была доказана неразрешимость проблемы останова).

§ 36 Сложность вычислений

Что такое сложность вычислений?

Центральная задача теории алгоритмов — выяснить, существует ли алгоритм решения той или иной задачи. Если существует, то возникает следующий вопрос: а можно ли им воспользоваться на практике, при современном уровне развития вычислительной техники? То есть способен ли компьютер за приемлемое время получить результат? Например, в игре в шахматы возможно лишь конечное количество позиций и, значит, только конечное количество различных партий. Следовательно, теоретически можно перебрать все возможные партии и выяснить, кто побеждает при правильной игре — белые или чёрные. Однако количество вариантов настолько велико, что современные компьютеры не могут выполнить такой перебор за приемлемое время.

Что мы хотим от алгоритма? Во-первых, чтобы он работал как можно быстрее. Во-вторых, чтобы объём необходимой памяти был как можно меньше. В-третьих, чтобы он был как можно более прост и понятен, что позволяет легче отлаживать программу. К сожалению, эти требования противоречивы, и в серьёзных задачах редко удаётся найти алгоритм, который был бы лучше остальных по всем показателям.

Часто говорят о *временно́й сложности* алгоритма (*быстродействии*) и *пространственной сложности*, которая определяется объёмом необходимой памяти. Поскольку память постоянно дешевеет, а быстродействие компьютеров растёт медленно, мы будем рассматривать главным образом временну́ю сложность — время выполнения программы, работающей по данному алгоритму.

В общем случае, говоря о сложности алгоритма, нужно уточнить, о каком исполнителе идёт речь, какие элементарные операции мы используем. Как правило, это один из универсальных исполнителей (во многих случаях универсальным исполнителем можно считать компьютер). Временем работы алгоритма называется количество выполненных им элементарных операций T. Такой подход позволяет оценивать именно качество алгоритма, а не свойства исполнителя (например, быстродействие компьютера, на котором выполняется алгоритм).

Как правило, величина T будет существенно зависеть от объёма исходных данных: поиск в списке из 10 элементов завершится гораздо быстрее, чем в списке из 10 000 элементов. Поэтому сложность алгоритма обычно связывают с размером входных данных n и определяют как функцию $T(n)$. Например, для алгоритмов обработки массивов в качестве размера n используют длину массива. Функция $T(n)$ называется **временно́й сложностью алгоритма**.

Примеры

Рассмотрим алгоритмы выполнения различных операций с массивом `A` длины n, который может быть объявлен в программе на школьном алгоритмическом языке как

```
целтаб A[1:n]
```

Пример 1. Требуется вычислить сумму первых трёх элементов массива (при $n \geq 3$).

Решение этой задачи содержит всего один оператор:

```
Sum:=A[1]+A[2]+A[3]
```

Этот алгоритм включает две операции сложения и одну операцию записи значения в память, поэтому его сложность $T(n) = 3$ не зависит от размера массива вообще.

Пример 2. Требуется вычислить сумму всех элементов массива. В этой задаче уже не обойтись без цикла:

```
Sum:=0
нц для i от 1 до n
  Sum:=Sum+A[i]
кц
```

Здесь выполняется *n* операций сложения и *n* + 1 операций записи в память[1], поэтому его сложность $T(n) = 2n + 1$ возрастает линейно с увеличением длины массива[2].

Пример 3. Требуется отсортировать все элементы массива по возрастанию методом выбора.

Напомним, что метод выбора предполагает поиск на каждом шаге минимального из оставшихся неупорядоченных значений (здесь *i*, *j*, *nMin* и *c* — целочисленные переменные):

```
нц для i от 1 до n-1
  nMin:=i;
  нц для j от i+1 до n
    если A[j]<A[nMin] то nMin:=j все
  кц
  если nMin<>i то
    c:=A[i]; A[i]:=A[nMin]; A[nMin]:=c
  все
кц
```

Подсчитаем отдельно количество сравнений $T_c(n)$ и количество перестановок $T_p(n)$. Количество сравнений не зависит от данных и определяется числом шагов внутреннего цикла:

$$T_c(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n.$$

Число перестановок зависит от данных. Например, если массив уже отсортирован в нужном порядке, перестановок не будет вообще. В худшем случае на каждом шаге основного цикла происходит перестановка, всего их будет $T_p(n) = n - 1$.

---

[1] Здесь и далее для упрощения выводов мы не учитываем команды, необходимые для организации цикла, потому что при больших *n* время их выполнения очень мало в сравнении со временем выполнения остальных операторов.

[2] Предполагается, что сложение любых двух чисел выполняется одинаковое время.

Что такое асимптотическая сложность?

Допустим, что нужно выбрать между несколькими алгоритмами, которые имеют разную сложность. Какой из них лучше (работает быстрее)? Оказывается, для этого необходимо знать размер массива данных, которые нужно обрабатывать. Сравним, например, три алгоритма, сложность которых

$$T_1(n) = 10\,000 \cdot n, \quad T_2(n) = 100 \cdot n^2, \quad T_3(n) = n^3.$$

Построим эти зависимости на графике (рис. 5.8). При $n \le 100$ получаем $T_3(n) < T_2(n) < T_1(n)$, при $n = 100$ количество операций для всех трёх алгоритмов совпадает, а при больших n имеем $T_3(n) > T_2(n) > T_1(n)$.

Обычно в теоретической информатике при сравнении алгоритмов используется их **асимптотическая сложность**, т. е. скорость роста количества операций при больших значениях n. При этом запись $O(n)$ (читается «О большое от n») обозначает, что, начиная с некоторого значения $n = n_0$, количество операций ограничено функцией $c \cdot n$, где c — некоторая константа:

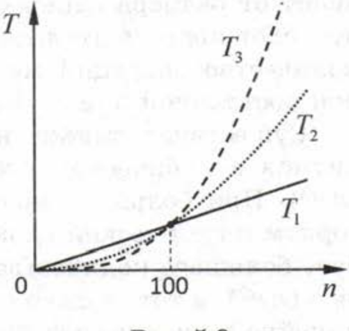


Рис. 5.8

$$T(n) \le c \cdot n \text{ для } n \ge n_0.$$

Такие алгоритмы имеют **линейную сложность**, т. е. при увеличении размера данных в 10 раз объём вычислений увеличивается тоже примерно в 10 раз.

Пусть, например, $T(n) = 2n - 1$, как в алгоритме поиска суммы элементов массива. Очевидно, что при этом $T(n) \le 2n$ для всех $n \ge 1$, поэтому алгоритм имеет линейную сложность.

Многие известные алгоритмы имеют **квадратичную сложность** $O(n^2)$. Это значит, что сложность алгоритма ограничена функцией $c \cdot n^2$:

$$T(n) \le c \cdot n^2 \text{ для } n \ge n_0.$$

При этом если размер данных увеличивается в 10 раз, то количество операций (и время выполнения) увеличивается пример-

но в 100 раз. Пример такого алгоритма — сортировка методом прямого выбора, для которой число сравнений

$$T_c(n) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{для всех } n \ge 0.$$

> Алгоритм имеет **асимптотическую сложность** $O(f(n))$, если найдётся такая постоянная c, что для всех $n \ge n_0$ выполняется условие $T(n) \le c \cdot f(n)$.

Это значит, что при $n \ge n_0$ график функции $c \cdot f(n)$ идёт выше, чем график функции $T(n)$ (рис. 5.9).

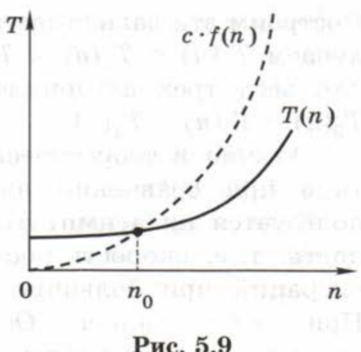


Рис. 5.9

Если количество операций не зависит от размера данных, то говорят, что сложность алгоритма $O(1)$, т. е. количество операций меньше некоторой постоянной при любых n.

Существует также немало алгоритмов с кубической сложностью — $O(n^3)$. При больших значениях n алгоритм с кубической сложностью требует большего количества вычислений, чем алгоритм со сложностью $O(n^2)$, а тот, в свою очередь, работает дольше, чем алгоритм с линейной сложностью. Заметьте, что при малых значениях n всё может быть наоборот; это зависит от постоянной c для каждого из алгоритмов.

Известны и алгоритмы, для которых количество операций растёт быстрее, чем любой полином, например как $O(2^n)$ или $O(n!)$. Они встречаются чаще всего в задачах оптимизации, которые решаются только методом полного перебора. Самая известная задача такого типа — это **задача коммивояжёра** (бродячего торговца), который должен посетить по одному разу каждый из указанных городов и вернуться в начальную точку. Для него нужно выбрать оптимальный маршрут, при котором стоимость поездки (или общая длина пути) будет минимальной.

Ещё один пример сложной задачи, которая решается только полным перебором всех вариантов, — **задача выполнимости**. Дано логическое выражение, которое содержит только имена логических переменных, скобки, а также операции «И», «ИЛИ» и «НЕ». Требуется определить, существует ли набор значений логических переменных, при котором заданное выражение истинно.

Алгоритмы поиска

Сравним вычислительную сложность двух наиболее известных алгоритмов поиска.

Пример 4 (линейный поиск). Дан массив, в котором элементы расположены в произвольном порядке. Требуется найти в нём заданное значение X или сообщить, что его нет.

Решение этой задачи сводится к последовательному просмотру всех элементов массива:

```
nX:=0
нц для i от 1 до n
  если A[i]=X то
    nX:=i
    выход
  все
кц
если nX>0 то
  вывод "A[", nX, "]=", X
иначе
  вывод "Элемент не найден"
все
```

В этом алгоритме число сравнений (в худшем случае) равно $T(n) = n$, поэтому он имеет линейную сложность.

Пример 5 (двоичный поиск). Дан массив, в котором элементы упорядочены по возрастанию. Требуется найти в нём заданное значение X или сообщить, что его нет.

По сравнению с предыдущей задачей, элементы массива отсортированы, и это ускоряет решение, потому что можно применить метод двоичного поиска (дихотомии):

```
L:=1; R:=n+1
нц пока L<R-1
  c:=div(L+R, 2)     | или c:=L+div(R-L,2)
  если X<A[c] то
    R:=c
  иначе
    L:=c
  все
кц
```

```
если A[L]=X то
  вывод "A[", L, "]=", X
иначе
  вывод "Элемент не найден"
все
```

Попробуем определить, сколько раз выполняется основной цикл при двоичном поиске. Сначала ширина интервала поиска — все n элементов массива. На каждом шаге этот интервал делится на 2, процесс завершается, когда левая и правая границы интервала совпадут. Предположим, что число элементов — это целая степень двойки, т. е. $n = 2^m$. Тогда за m шагов ширина интервала сужается до 1, а на следующем шаге его границы совпадут, и цикл закончится. Таким образом, количество шагов цикла равно $m + 1$. Из равенства $n = 2^m$ получаем $m = \log_2 n$, так что

$$T(n) = \log_2 n + 1.$$

Например, при $n = 2^{16}$ линейный поиск потребует в худшем случае $2^{16} = 65\,536$ сравнений, а двоичный — всего $16 + 1 = 17$ сравнений.

Таким образом, алгоритм двоичного поиска имеет асимптотическую сложность $O(\log n)$. Основание логарифма обычно не указывают, потому что выражения $\log_a n$ и $\log_b n$ различаются на постоянный множитель (который можно включить в постоянную c):

$$\log_a n = \frac{1}{\log_b a} \cdot \log_b n.$$

Можно ли сказать, что алгоритм двоичного поиска лучше алгоритма линейного поиска? Нет! Ведь алгоритм линейного поиска применим к любым массивам данных, а алгоритм двоичного поиска — только к упорядоченным (отсортированным). А если мы сначала отсортируем массив, а потом применим к нему двоичный поиск, то общее время работы будет больше, чем при линейном поиске.

Ситуация меняется, если нам нужно многократно выполнять операцию поиска для одних и тех же данных (так, как правило, бывает при работе с базами данных). Тогда имеет смысл заранее отсортировать массив (применить предварительную обработку данных), а затем, используя двоичный поиск, экономить время при каждом новом поисковом запросе.

Алгоритмы сортировки

Ранее мы проанализировали один из простых методов сортировки массивов — метод прямого выбора и выяснили, что его асимптотическая сложность $O(n^2)$. Повторим анализ для метода пузырька, который также изучался в 10 классе:

```
нц для i от 1 до n-1
  нц для j от n-1 до i шаг -1
    если A[j]>A[j+1] то
      c:= A[j]; A[j]:= A[j+1]; A[j+1]:= c;
    все
  кц
кц
```

На первом шаге основного цикла выполняется $n - 1$ шагов внутреннего цикла, т. е. $n - 1$ сравнений. Далее количество сравнений уменьшается до 1, так что общее количество сравнений равно:

$$T_c(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n,$$

так же как и у алгоритма прямого выбора. В то же время в худшем случае при каждом сравнении выполняется перестановка, что требует

$$T_p(n) = 3 \cdot \frac{n(n-1)}{2} = \frac{3}{2}n^2 - \frac{3}{2}n$$

операций присваивания. Таким образом, этот алгоритм имеет асимптотическую сложность $O(n^2)$ как по числу сравнений, так и по числу присваиваний.

Существуют ли более эффективные сортировки, имеющие, например, линейную сложность? Да, для некоторых особых случаев существуют. Например, если известно, что все значения исходного массива находятся в интервале от 1 до некоторого значения MAX, можно использовать **сортировку подсчётом**. Для этого выделяется дополнительный массив счётчиков:

```
целтаб C[1:MAX]
```

который предварительно обнуляется:

```
нц для i от 1 до MAX
  C[i]:= 0
кц
```

Затем в цикле проходим весь массив с данными и для каждого элемента $A[i]$ увеличиваем счётчик $C[A[i]]$. Например, если $A[i]$=20, счётчик $C[20]$ увеличивается на 1. После окончания цикла в каждом счётчике $C[i]$ находится количество значений исходного массива, равных i.

```
нц для i от 1 до n
  C[A[i]]:=C[A[i]]+1
кц
```

Теперь остаётся расставить числа в массиве A в нужном количестве. Например, если C[20] = 5, в массив A записываются последовательно 5 значений, равных 20:

```
k:= 1
нц для i от 1 до MAX
  нц для j от 1 до C[i]
    A[k]:=i
    k:=k+1
  кц
кц
```

Попробуем подсчитать количество операций для этого алгоритма. Заполнение массива C нулями требует MAX присваиваний. Цикл подсчёта элементов содержит n сложений и присваиваний, т. е. его сложность — линейная, $O(n)$. Наконец, последний вложенный цикл выполняет также n сложений и присваиваний (по числу элементов массива A), поэтому алгоритм в целом имеет линейную сложность по n.

Однако нужно учитывать, что принципиальное ускорение алгоритма в сравнении с предыдущими получено за счёт того, что:

- все значения — целые числа в ограниченном диапазоне;
- есть возможность использовать дополнительный массив размером MAX, который может значительно превышать размер исходного массива.

Здесь проявляется компромисс «скорость — память», который присутствует во многих задачах: ускорение алгоритма возможно за счёт использования дополнительной памяти и наоборот, экономия памяти приводит к замедлению работы алгоритма.

Доказано, что в общем случае вычислительная сложность сортировки, основанной только на использовании операций «сравнить» и «переставить», не может быть меньше, чем $O(n\log n)$. Именно такую сложность имеют, например, **сортировка слиянием** (англ. *merge sort*) и **пирамидальная сортировка** (англ. *heap sort*), которые применяются при работе с большими наборами данных. **Быстрая сортировка** (англ. *quick sort*), которая изучалась в 10 классе, в среднем тоже имеет сложность $O(n\log n)$, однако в худшем случае (когда на каждом шаге массив делится на две части, одна из которых состоит из одного элемента) требуется $O(n^2)$ обменов.

Вопросы и задания

1. Какие критерии используются для оценки качества алгоритмов?
2. Почему скорость работы алгоритма оценивается не временем выполнения, а количеством элементарных операций?
3. Как учитывается размер данных при оценке скорости алгоритма?
4. Что означают записи $O(1)$, $O(n)$, $O(n^2)$ и $O(2^n)$?
5. В каких случаях алгоритм, имеющий асимптотическую сложность $O(n^2)$, может работать быстрее, чем алгоритм с асимптотической сложностью $O(n)$?

Задачи

1. Оцените количество операций для алгоритмов:
   а) поиска всех делителей числа;
   б) нахождения минимального и максимального элементов массива;
   в) определения количества положительных элементов массива;
   г) проверки числа на простоту.

   В каждом случае опишите набор используемых элементарных операций. Определите асимптотическую сложность этих алгоритмов.

*2. Предложите алгоритм, позволяющий найти и вывести на экран те символы, которые встречаются в строке более одного раза. Оцените его асимптотическую сложность.

*3. Алфавит языка племени «тумба-юмба» содержит k символов. Предложите алгоритм построения всех возможных слов этого языка, имеющих длину n символов, и оцените его асимптотическую сложность.

§ 37
Доказательство правильности программ

Как доказать правильность программы?

Как правило, программист разрабатывает программу на заказ, и от него требуется не только написать код, но и убедиться, что код работает правильно, т. е. в соответствии с требованиями заказчика.

Очевидно, что если программа выдаёт неверный результат хотя бы для одного варианта входных данных, можно сразу сказать, что она некорректна, т. е. содержит ошибки.

Сложнее доказать правильность программы — убедиться, что она выдает верные результаты при любых допустимых входных данных. Программисты-практики для решения этой задачи используют **тестирование:** проверяют работу программы с помощью набора тестовых данных, для которых известен правильный результат. Если полученный результат не совпадает с заданным, выполняется **отладка** программы, т. е. поиск и исправление ошибок.

Однако, как писал нидерландский учёный, один из создателей современного программирования Эдсгер Дейкстра, «отладка может показать лишь наличие ошибок и никогда — их отсутствие». В результате можно гарантировать верную работу программы только при тех данных, которые использовались в контрольных тестах. Кроме того, неясно, как определить, что все ошибки выявлены и нужно завершить отладку.

Э. В. Дейкстра (1930–2002)

Пример 1. Рассмотрим следующую программу для выбора максимального из трёх значений, записанных в переменных a, b и c:

```
если a>b то M:=a иначе M:=b все
если b>c то M:=b иначе M:=c все
```

Проверяя её на тестах

(a,b,c) = (1,2,3), (1,3,2), (2,1,3) и (2,3,1),

мы во всех этих случаях получаем в переменной M верный ответ 3. Однако это не означает, что программа правильная, так как

существует контрпример (3,2,1): для этого набора входных данных в переменной M в результате оказывается число 2.

Чтобы быть уверенными в том, что программа работает правильно при *любых* допустимых исходных данных, применяют методы **доказательного программирования**: для каждого блока программы составляют требования к входным и выходным данным и строго доказывают, что программа всегда работает верно.

К сожалению, доказывать правильность программ не так просто, и в таких доказательствах тоже возможны ошибки. Однако при этом автор должен глубоко разобраться в алгоритме и его «подводных камнях», и часто при этом обнаруживаются ошибки, которые могли бы проявиться уже после выпуска программы в свет.

На практике редко доказывают правильность всей программы в целом. В то же время очень полезно доказывать правильность отдельных блоков (циклов, процедур и функций) для уменьшения количества «необъяснимых» ошибок и сокращения времени отладки.

Покажем метод доказательства правильности программы на простом примере.

Пример 2. Требуется доказать, что после выполнения следующей программы значения переменных a и b меняются местами:

```
b:=a+b          | 1
a:=b-a          | 2
b:=b-a          | 3
```

Предполагается, что сумма исходных чисел не приводит к переполнению разрядной сетки. Для удобства операторы программы пронумерованы.

Обозначим начальные значения переменных a и b через a_0 и b_0. После выполнения оператора 1 в переменной b будет записано значение $a_0 + b_0$. Оператор 2 записывает в переменную a значение

$$b - a = a_0 + b_0 - a_0 = b_0.$$

В результате выполнения оператора 3 получаем новое значение переменной b, равное

$$b - a = a_0 + b_0 - b_0 = a_0.$$

Таким образом, в результате выполнения программы переменные a и b будут равны b_0 и a_0 соответственно, что и требовалось доказать. Поэтому приведённая программа правильная.

Пример 3. Попробуем доказать или опровергнуть правильность уже встречавшейся ранее программы для выбора максимального из трёх значений, записанных в переменных a, b и c:

```
если a>b то M:=a иначе M:=b все   | 1
если b>c то M:=b иначе M:=c все   | 2
```

Анализируя строку 2, выясняем, что в ней значение переменной M *всегда* будет изменено, т. е. результат работы первой строки программы стирается, и

$$M = \begin{cases} b, \text{ если } b > c, \\ c, \text{ если } c \geq b. \end{cases}$$

Конечно, эта величина не совпадает с определением максимального значения из a, b и c. Таким образом, программа неправильная: она выдает неверное значение, если максимальное из трёх чисел хранилось в переменной a. Контрпример мы уже приводили: (3,2,1).

Алгоритм Евклида

Теперь докажем, что один из древнейших известных алгоритмов — алгоритм Евклида — действительно вычисляет наибольший общий делитель (НОД) двух натуральных чисел (мы рассматривали его в 10 классе).

Алгоритм Евклида. Пусть заданы два натуральных числа m и n, причём $m > n$. Для вычисления НОД(m, n) следует многократно заменять большее число остатком от деления большего на меньшее до тех пор, пока меньшее число не станет равным нулю. Тогда оставшееся ненулевое число и есть НОД(m, n).

Программа, основанная на алгоритме Евклида, может выглядеть, например, так (здесь a, b и r — целочисленные переменные):

```
a:=m; b:=n          | 1   НОД(a,b)=НОД(m,n)
нц пока b<>0        | 2
  r:=mod(a, b)      | 3
  a:=b; b:=r        | 4   НОД(a,b)=НОД(m,n)
кц                  | 5
вывод a             | 6   НОД(a,b)=НОД(m,n), b=0
```

Докажем, что в результате этого алгоритма в переменной a находится НОД(m, n).

В строке 1 исходные значения копируются из переменных m и n соответственно в переменные a и b. Очевидно, что при этом выполнено условие НОД(a, b) = НОД(m, n).

На каждом шаге цикла (в строках 3–4) вычисляется остаток r от деления a на b и пара (a, b) заменяется на пару (b, r). Какими свойствами обладают полученные значения b и r?

Поскольку r — это остаток от деления a на b, до выполнения строк 3–4 было справедливо равенство $a = bp + r$, где p — некоторое целое число. Тогда, если a и b имеют общий делитель, то такой же делитель имеет и r. Следовательно, НОД(b, r) = НОД(a, b) = = НОД(m, n). Это значит, что условие НОД(a, b) = НОД(m, n) по-прежнему выполняется после каждого шага цикла.

Поскольку остаток r с каждым шагом строго уменьшается, в конце концов он станет равным нулю и запишется в переменную b при выполнении строки 4. Цикл сразу же закончится, поскольку нарушится условие его выполнения. После завершения работы цикла условие НОД(a, b) = НОД(m, n) по-прежнему выполняется, но, кроме того, $b = 0$. Отсюда следует, что a = НОД(m, n).

Инвариант цикла

Таким образом, для алгоритма Евклида существует условие НОД(a, b) = НОД(m, n), которое остаётся справедливым на протяжении всего выполнения алгоритма: перед началом цикла, после каждого шага цикла и после окончания работы цикла. Такое условие называется инвариантом цикла (англ. *invariant* — неизменный).

Инвариант цикла — это соотношение между значениями переменных, которое остаётся справедливым после завершения любого шага цикла.

Выделив в явном виде инвариант каждого цикла, мы избегаем многих возможных ошибок на начальной стадии и делаем первый шаг к доказательству правильности всей программы. Как писал академик Андрей Петрович Ершов, один из первых теоретиков программирования в СССР, «программиста бьют по рукам, если он посмеет написать оператор цикла, не найдя перед этим его инварианта».

А. П. Ершов
(1931–1988)

Рассмотрим несколько примеров.

Пример 1. Двое играют в следующую игру: перед ними лежат в ряд $N + 1$ камней, сначала N белых, и в конце цепочки — один чёрный. За один ход каждый может взять от 1 до 3 камней. Проигрывает тот, кто берет чёрный («несчастливый») камень.

Начнём анализ с простейших случаев. Если $N = 0$, то первый игрок проиграл, он может взять только чёрный камень. Если $N = 1, 2, 3$, то, наоборот, при правильной игре проигрывает второй игрок, потому что первый может забрать все камни, кроме чёрного. Вариант $N = 4$ снова приводит к проигрышу первого игрока, потому что забрать все белые камни он не может, а после его хода второй оставит только чёрный камень. Также проигрышными будут позиции при $N = 8, 12, 16, \ldots$, т. е. при любых значениях N, которые делятся на 4.

Таким образом, для своего выигрыша игрок должен каждым своим ходом восстанавливать *инвариант*: число оставшихся белых камней должно быть кратно 4. Если инвариант выполнен в начальной позиции, положение проигрышное и первый игрок может надеяться только на ошибку соперника.

Пример 2. Пусть задан массив A длины n. Найдём инвариант цикла в программе суммирования элементов массива:

```
Sum:=0
нц для i от 1 до n
  Sum:=Sum+A[i]
кц
```

Здесь на каждом шаге к переменной Sum добавляется элемент массива $A[i]$, так что при любом i после окончания очередного шага цикла в Sum накоплена сумма всех элементов массива с номерами от 1 до i. Это и есть инвариант цикла. Поэтому сразу можно сделать вывод о том, что после завершения цикла в переменной Sum будет записана сумма всех элементов массива.

Аналогично можно показать, что в алгоритме поиска наименьшего значения в массиве:

```
Min:=A[1]
нц для i от 2 до n
  если A[i]<Min то
    Min:=A[i]
  все
кц
```

инвариант формулируется так: после выполнения каждого шага цикла в переменной Min записан минимальный из первых i элементов. Отсюда сразу следует, что после завершения цикла (при $i = n$) в этой переменной будет минимальный из всех элементов.

Пример 3. Для того же массива найдем инвариант цикла в программе сортировки элементов массива методом пузырька:

```
нц для i от 1 до n-1
  нц для j от n-1 до i шаг -1
    если A[j]>A[j+1] то
      c:= A[j]; A[j]:= A[j+1]; A[j+1]:= c;
    все
  кц
кц
```

До начала алгоритма элементы расположены произвольно. На каждом шаге внешнего цикла на свое место «всплывает» один элемент массива. Поэтому инвариант этого цикла можно сформулировать так: «После выполнения i-го шага цикла первые i элементов массива отсортированы и установлены на свои места».

Теперь построим инвариант внутреннего цикла. В этом цикле очередной «лёгкий» элемент поднимается вверх к началу массива. Перед первым шагом внутреннего цикла элемент, который будет стоять на i-м месте в отсортированном массиве, может находиться в любой ячейке от $A[i]$ до $A[n]$. После каждого шага его «зона нахождения» сужается на одну позицию, так что инвариант внутреннего цикла можно сформулировать так: «Элемент, который будет стоять на i-м месте в отсортированном массиве, может находиться в любой ячейке от $A[i]$ до $A[j]$». Очевидно, что когда в конце этого цикла $j = i$, элемент $A[i]$ встаёт на своё место.

В предыдущих примерах мы определяли инвариант готового цикла. Теперь покажем, как можно строить цикл с помощью заранее выбранного инварианта.

Пример 4. Рассмотрим алгоритм **быстрого возведения в степень,** основанный на использовании операций возведения в квадрат и умножения. Он использует две очевидные формулы:

(1) $a^k = a^{k-1} \cdot a$ при нечётной степени k и
(2) $a^k = (a^2)^{k/2}$ при чётной степени k.

Покажем, как работает алгоритм, на примере возведения числа a в степень 7:

$$a^7 = a^6 \cdot [a] = (a^2)^3 \cdot [a] = (a^2)^2 \cdot [a^2 \cdot a] = (a^4)^1 \cdot [a^2 \cdot a] =$$
$$= (a^4)^0 \cdot [a^4 \cdot a^2 \cdot a] = [a^4 \cdot a^2 \cdot a].$$

Здесь поочерёдно применяются первая и вторая формулы. Заметим, что на каждом этапе выражение a^n можно представить в виде $a^n = b^k \cdot p$, где через p обозначена часть, взятая выше в квадратные скобки. Если нам каким-то образом удастся уменьшить k до нуля, сохранив это равенство, то мы получим $a^n = p$, т. е. задача будет решена, а результат будет находиться в переменной p.

Таким образом, равенство $a^n = b^k \cdot p$ можно использовать как инвариант цикла. Для того чтобы обеспечить выполнение этого равенства в начальный момент, можно принять, например, $b = a$, $k = n$ и $p = 1$. Далее в цикле применяются формулы (1) и (2) (в зависимости от чётности k на данном шаге). Цикл заканчивается, когда $k = 0$. В результате получаем следующее решение:

```
b:=a; k:=n; p:=1
нц пока k<>0
  если mod(k,2)=0 то
    k:=div(k,2)
    b:=b*b
  иначе
    k:=k-1
    p:=b*p
  все
кц
вывод p
```

Заметим, что инвариант цикла $a^n = b^k \cdot p$ выполняется до начала цикла, после каждого шага, а также после завершения цикла. Таким образом, мы написали код программы и одновременно доказали правильность этого блока.

Спецификация

Для доказательства правильности программы необходимо иметь спецификацию — точное описание того, что должно быть сделано в результате работы программы.

Спецификация — точная и полная формулировка задачи, содержащая информацию, необходимую для построения алгоритма её решения.

На практике спецификации программ обычно формулируют на естественном языке, в котором слова могут иметь несколько разных значений. Для строгого доказательства желательно, чтобы спецификация была задана в формальном виде, с помощью формул или соотношений между величинами.

По предложению английского ученого Ч. Хоара, спецификация записывается в форме $\{Q\}S\{R\}$, где Q — начальное условие, S — программа и R — утверждения, описывающие конечный результат. Запись $\{Q\}S\{R\}$ означает следующее: «Если выполнение программы S началось в состоянии, удовлетворяющем Q, то гарантируется, что оно завершится через конечное время в состоянии, удовлетворяющем R».

Корректная программа — это программа, соответствующая спецификации.

Если для исходных данных не удовлетворяется условие Q, программа должна сообщать об этом пользователю и закончить работу. Это говорит о **надёжности** программы.

Например, для алгоритма Евклида условия Q и R могут выглядеть так:

$$Q\colon m > n > 0, \quad R\colon a = \text{НОД}(m,n),$$

а для программы суммирования элементов массива $A[1{:}n]$ (см. пример 2 на стр. 40) — так:

$$Q\colon n > 0, \qquad R\colon Sum = \sum_{i=1}^{n} A[i] = A[1] + A[2] + \ldots + A[n].$$

Спецификации могут (и должны) быть составлены не только для программы в целом, но и для её отдельных блоков (процедур, функций, циклов и т. д.). Полезно вносить утверждения Q и R прямо в текст программы. Построенная таким образом *аннотированная программа* — это ещё один шаг к доказательному программированию.

Ч. Хоар разработал специальный аппарат, позволяющий доказывать правильность программы на основе спецификаций отдельных блоков. Приведём простейшие правила преобразования:

- если $\{Q\}S\{P\}$ и $P \Rightarrow R$ (из истинности P следует истинность R), то $\{Q\}S\{R\}$;
- если $\{Q\}S\{P\}$ и $R \Rightarrow Q$, то $\{R\}S\{P\}$;
- если программа S — это последовательное выполнение блоков S_1 и S_2, для которых выполняются спецификации $\{Q\}S_1\{P\}$ и $\{P\}S_2\{R\}$, то выполняется спецификация $\{Q\}S\{R\}$.

Доказательство правильности программ используют в двух ситуациях:

- доказывают правильность готовых программ (**верификация** программ);
- строят программы одновременно с доказательством их правильности (синтез программ).

Как правило, верификация — это очень трудоёмкий и сложный процесс, и оказывается значительно проще использовать доказательства правильности во время разработки программы. При этом программы получаются проще, эффективнее и значительно надёжнее.

Вопросы и задания

1. Зачем нужно доказывать правильность программ?
2. Расскажите о двух подходах к проверке правильности программ.
3. Почему с помощью тестирования сложно доказать правильность программы? В каких случаях это всё же можно сделать? Приведите примеры.
4. Что изменится в доказательстве алгоритма Евклида, если m и n — это произвольные натуральные числа (неравенство $m > n$ может не выполняться)?
5. Что такое инвариант цикла?
6. Зачем нужно определять инвариант цикла?
7. Что такое спецификация? Почему желательно формулировать её в виде формальных утверждений, а не на естественном языке?
8. Объясните запись $\{Q\}S\{R\}$.
9. Какая программа называется корректной?
10. Как вы думаете, можно ли назвать корректной программу, которая «зависает» при неверных входных данных? Обсудите этот вопрос в классе.
11. Что такое верификация программы?

12. Как вы думаете, что сложнее — доказывать правильность готовой программы или сразу писать программу, доказывая правильность отдельных блоков? Почему? Обсудите этот вопрос в классе.

Задачи

1. Докажите, что следующие операторы дают одинаковый результат при любых значениях L и R (рассмотрите чётные и нечётные значения обеих переменных):

```
c:=div(L+R,2)                c:=L+div(R-L,2)
```

Какие достоинства и недостатки есть у каждого метода вычисления этой величины?

2. Докажите, что в результате выполнения следующего фрагмента программы в переменной M не всегда будет записано максимальное из трёх чисел (a, b и c):

```
M:=a
если b>a то M:=b все
если c>b то M:=c все
```

Приведите контрпример, т. е. такие значения a, b и c, при которых значение M будет отличаться от $\max(a, b, c)$. Как можно исправить эту программу, заменив в ней всего один символ?

3. Докажите или опровергните правильность программы для выбора максимального из трёх значений, записанных в переменных a, b и c:

```
если a>b то M:=a
иначе если b>c то M:=b
иначе если c>a то M:=c
все; все; все
```

Если эта программа некорректная, приведите контрпример. Может ли быть, что при каких-то входных данных значение переменной M будет неопределённым?

4. Докажите, что следующий фрагмент программы правильно сортирует значения в переменных a, b и c по возрастанию, т. е. всегда получается $a \le b \le c$:

```
если a>b то поменять(a, b) все
если b>c то поменять(b, c) все
если a>b то поменять(a, b) все
```

Алгоритм поменять меняет местами значения переменных-параметров.

5. В игре «ним» двое игроков по очереди берут камни из двух кучек. За один ход можно взять любое ненулевое количество камней, но только из одной кучки. Тот, кому не осталось камней, проигрывает. Как определить, кто выиграет при правильной игре? Какой инвариант обеспечивает выигрыш?

6. Определите инвариант цикла для следующего алгоритма двоичного поиска (предполагается, что элементы массива A отсортированы по неубыванию):

```
L:=1; R:=n+1
нц пока L<R-1
  c:=div(L+R,2)
  если X<A[c] то
    R:=c
  иначе
    L:=c
  все
кц
```

Используя найденный инвариант, определите, какой именно элемент массива будет найден, если в массиве есть несколько элементов, равных X. Как нужно изменить инвариант (и цикл), чтобы найти *первый* элемент, равный X?

7. Определите инварианты для следующих циклов. Что будет вычислено в переменной b?

а)

```
k:=0; b:=1;
нц пока k<n
  k:=k+1;
  b:=b*a;
кц
```

б)

```
k:=n; b:=1;
нц пока k>0
  k:=k-1;
  b:=b*a;
кц
```

8. Определите условия Q и R для алгоритмов:
   а) нахождения суммы всех делителей числа;
   б) проверки числа на простоту;
   в) определения количества слов в символьной строке;
   г) двоичного поиска элемента в отсортированном массиве;
   д) перестановки элементов массива в обратном порядке;
   е) преобразования числа из символьной записи в значение целого типа.

9. Предложите другие начальные значения переменных b, k и p в алгоритме быстрого возведения в степень. Инвариант цикла должен сохраниться.
10. Оцените сложность алгоритма быстрого возведения в степень при $n = 2^m$.

Практические работы к главе 5

Работа № 36 «Машина Тьюринга»
Работа № 37 «Машина Поста»
Работа № 38 «Нормальные алгоритмы Маркова»
Работа № 39 «Вычислимые функции»
Работа № 40 «Инвариант цикла»

ЭОР к главе 5 на сайте ФЦИОР (http://fcior.edu.ru)

- Алгоритмически неразрешимые задачи

Самое важное в главе 5

- Интуитивное понятие алгоритма, которое мы использовали ранее, непригодно для математического доказательства неразрешимости задач.
- Входные и выходные данные любого алгоритма можно закодировать в виде последовательностей символов некоторого алфавита (и даже двоичного алфавита).
- Про любой алгоритм можно сказать следующее:
  - алгоритм получает на вход дискретный объект (например, слово);
  - алгоритм обрабатывает входной объект по шагам (дискретно), строя на каждом шаге промежуточные дискретные объекты; этот процесс может закончиться или не закончиться;
  - если выполнение алгоритма заканчивается, его результат — это объект, построенный на последнем шаге;
  - если выполнение алгоритма не заканчивается (алгоритм зацикливается) или заканчивается аварийно, то результат его работы при данном входе не определён.

- Алгоритм — это программа для некоторого исполнителя.
- Универсальный исполнитель — это исполнитель, который может моделировать работу любого другого исполнителя. Это значит, что для любого алгоритма, написанного для любого исполнителя, существует эквивалентный алгоритм для универсального исполнителя.
- Все универсальные исполнители эквивалентны между собой.
- Каждый алгоритм задаёт (вычисляет) функцию, которая преобразует входное слово в результат (выходное слово). Алгоритмы называются эквивалентными, если они задают одну и ту же функцию.
- Вычислимая функция — это функция, для вычисления которой существует алгоритм. Любая вычислимая функция может задаваться разными алгоритмами.
- Алгоритмически неразрешимая задача — это задача, соответствующая невычислимой функции.
- Говорят, что алгоритм имеет асимптотическую сложность $O(f(n))$, если найдётся такая постоянная c, что, начиная с некоторого $n = n_0$, выполняется условие $T(n) \leq c \cdot f(n)$.
- Задачи оптимизации, которые решаются только полным перебором вариантов, могут иметь, например, асимптотическую сложность $O(2^n)$ или $O(n!)$. Эти функции при больших n возрастают быстрее, чем многочлен любой степени.
- Чтобы обеспечить надёжность программы, используют методы доказательного программирования: разработка программы ведётся одновременно с доказательством её правильности.
- Инвариант цикла — это соотношение между величинами, которое остаётся справедливым после завершения любого шага цикла.

Глава 6
Алгоритмизация и программирование

§ 38
Целочисленные алгоритмы

Во многих задачах все исходные данные и необходимые результаты — целые числа. При этом всегда желательно, чтобы все промежуточные вычисления тоже проводились только с целыми числами. На это есть, по крайней мере, две причины:

- процессор, как правило, выполняет операции с целыми числами значительно быстрее, чем с вещественными;
- целые числа всегда точно представляются в памяти компьютера, и вычисления с ними также выполняются без погрешностей (если, конечно, не происходит переполнение разрядной сетки).

Решето Эратосфена

Простые числа широко используются во многих прикладных задачах, например при шифровании с помощью алгоритма RSA (вспомните материал учебника для 10 класса). Основные задачи при работе с простыми числами — это проверка числа на простоту и нахождение всех простых чисел в заданном диапазоне.

Пусть задано некоторое натуральное число N и требуется найти все простые числа в диапазоне от 1 до N. Самое простое (но неэффективное) решение этой задачи состоит в том, что в цикле перебираются все числа от 1 до N, и каждое из них отдельно проверяется на простоту. Например, можно проверить, есть ли у числа k делители в диапазоне от 2 до $\sqrt{k}$. Если ни одного такого делителя нет, то число k простое.

Описанный метод при больших N работает очень медленно. Греческий математик Эратосфен Киренский (275–194 гг. до н. э.) предложил другой алгоритм, который работает намного быстрее:

1) выписать все числа от 2 до N;
2) начать с $k = 2$;
3) вычеркнуть все числа, кратные k ($2k$, $3k$, $4k$ и т. д.);
4) найти следующее невычеркнутое число и присвоить его переменной k;
5) повторять шаги 3 и 4, пока $k < N$.

Покажем работу алгоритма при $N = 16$:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Первое невычеркнутое число — это 2, поэтому вычёркиваем все чётные числа:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~

Далее вычёркиваем все числа, кратные 3:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~

Все числа, кратные 5 и 7, уже вычеркнуты. Таким образом, получены простые числа 2, 3, 5, 7, 11 и 13.

Классический алгоритм можно улучшить, уменьшив количество операций. Заметьте, что при вычёркивании чисел, кратных трём, нам не пришлось вычёркивать число 6, так как оно уже было вычеркнуто. Кроме того, все числа, кратные 5 и 7, к последнему шагу тоже оказались вычеркнуты.

Предположим, что мы хотим вычеркнуть все числа, кратные некоторому k, например $k = 5$. При этом числа $2k$, $3k$ и $4k$ уже были вычеркнуты на предыдущих шагах, поэтому нужно начать не с $2k$, а с k^2. Тогда получается, что при $k^2 > N$ вычёркивать уже будет нечего, что мы и увидели в примере. Поэтому можно использовать улучшенный алгоритм:

1) выписать все числа от 2 до N;
2) начать с $k = 2$;
3) вычеркнуть все числа, кратные k, начиная с k^2;
4) найти следующее невычеркнутое число и присвоить его переменной k;
5) повторять шаги 3 и 4, пока $k^2 \le N$.

Чтобы составить программу, нужно определить, что значит «выписать все числа» и «вычеркнуть число». Один из возможных вариантов хранения данных — массив логических величин с индексами от 2 до N. Как и в учебнике 10 класса, слева будем писать программу на школьном алгоритмическом языке, а справа — на языке Паскаль.

Объявление переменных в программе будет выглядеть так (для $N = 100$):

```
цел i, k, N=100          const N=100;
логтаб A[2:N]            var i, k: integer;
                             A: array[2..N] of boolean;
```

Если число i не вычеркнуто, будем хранить в элементе массива $A[i]$ истинное значение, если вычеркнуто — ложное. В самом начале нужно заполнить массив истинными значениями:

```
нц для i от 2 до N                for i:=2 to N do
  A[i]:=да                          A[i]:=True;
кц
```

В основном цикле выполняется описанный выше алгоритм:

```
k:=2                              k:=2;
нц пока k*k<=N                    while k*k<=N do begin
  если A[k] то                      if A[k] then begin
    i:=k*k                            i:=k*k;
    нц пока i<=N                      while i<=N do begin
      A[i]:=нет                         A[i]:=False;
      i:=i+k                            i:=i+k
    кц                                end
  все                               end;
  k:=k+1                            k:=k+1
кц                                end;
```

Обратите внимание, что для того, чтобы вообще не применять вещественную арифметику, мы заменили условие $k \le \sqrt{N}$ на равносильное условие $k^2 \le N$, в котором используются только целые числа.

После завершения этого цикла невычеркнутыми остались только простые числа, для них соответствующие элементы массива содержат истинные значения. Эти числа нужно вывести на экран:

```
нц для i от 2 до N                for i:=2 to N do
  если A[i] то                      if A[i] then
    вывод i,нс                        writeln(i);
  все
кц
```

«Длинные» числа

Современные алгоритмы шифрования используют достаточно длинные ключи, которые представляют собой числа длиной 256 битов и больше. С ними нужно выполнять разные операции: складывать, умножать, находить остаток от деления. Вопрос состоит в том, как хранить такие числа в памяти, где для целых чисел отводится память значительно меньших размеров (обычно до 64 битов). Ответ достаточно очевиден: нужно разбить длинное число на части так, чтобы оно занимало несколько ячеек памяти.

«Длинное» число — это число, которое не помещается в переменную стандартных типов данных языка программирования. Алгоритмы работы с длинными числами называют «**длинной арифметикой**».

Для хранения «длинного» числа удобно использовать массив целых чисел. Например, число 12345678 можно записать в массив с индексами от 0 до 9 таким образом:

	0	1	2	3	4	5	6	7	8	9
A	1	2	3	4	5	6	7	8	0	0

Такой способ имеет ряд недостатков:

1) нужно где-то хранить длину числа, иначе числа 12345678, 123456780 и 1234567800 будет невозможно различить;
2) неудобно выполнять арифметические операции, которые начинаются с младшего разряда;
3) память расходуется неэкономно, потому что в одном элементе массива хранится только один разряд — число от 0 до 9.

Чтобы избавиться от первых двух проблем, достаточно «развернуть» массив наоборот, так чтобы младший разряд находился в $A[0]$. В этом случае на рисунках удобно применять обратный порядок элементов:

	9	8	7	6	5	4	3	2	1	0
A	0	0	1	2	3	4	5	6	7	8

Теперь нужно найти более экономичный способ хранения длинного числа. Например, разместим в одном элементе массива три разряда числа, начиная справа:

	9	8	7	6	5	4	3	2	1	0
A	0	0	0	0	0	0	0	12	345	678

Здесь использовано равенство

$$12345678 = 12 \cdot 1000^2 + 345 \cdot 1000^1 + 678 \cdot 1000^0.$$

Фактически мы представили исходное число в системе счисления с основанием 1000.

Сколько разрядов можно хранить в одном элементе массива? Это зависит от размера элемента. Например, если переменная занимает 4 байта и число хранится со знаком, допустимый диапазон его значений:

$$\text{от } -2^{32} = -4\,294\,967\,296 \quad \text{до} \quad 2^{32} - 1 = 4\,294\,967\,295.$$

В таком элементе можно хранить до 9 разрядов десятичного числа, т. е. использовать систему счисления с основанием 1 000 000 000. Однако нужно учитывать, что с такими числами будут выполняться арифметические операции, результат которых должен помещаться в переменную некоторого типа. Например, если надо умножать разряды этого числа на число $k < 100$ и в языке программирования нет 64-битных целочисленных типов данных, то в элементе массива можно хранить не более 7 разрядов.

Покажем на примере, как можно использовать систему счисления с основанием 1 000 000 для выполнения операций с «длинными» числами.

Задача. Вычислить точно значение факториала $100! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot 99 \cdot 100$ и вывести его на экран в десятичной системе счисления (это число состоит более чем из сотни цифр и явно не помещается в одну переменную).

Для хранения «длинного» числа будем использовать целочисленный массив A. Определим необходимую длину массива. Заметим, что

$$1 \cdot 2 \cdot 3 \cdot \ldots \cdot 99 \cdot 100 < 100^{100}.$$

Число 100^{100} содержит 201 цифру, поэтому число 100! содержит не более 200 цифр. Если в каждом элементе массива записано 6 цифр, для хранения всего числа требуется не более 34 элементов:

```
цел N=33                  const N=33;
целтаб A[0:N]             var A: array[0..N] of integer;
```

Чтобы найти 100!, нужно сначала присвоить «длинному» числу значение 1, а затем последовательно умножать его на все числа от 2 до 100. Запишем эту идею на псевдокоде, обозначив через $[A]$ «длинное» число, находящееся в массиве A:

```
[A]:=1
нц для k от 2 до 100
  [A]:=[A]*k
кц
```

Записать в «длинное» число единицу — значит присвоить элементу $A[0]$ значение 1, а в остальные переменные записать нули:

```
A[0]:=1                      A[0]:=1;
нц для i от 1 до N           for i:=1 to N do
  A[i]:=0                      A[i]:=0;
кц
```

Таким образом, остаётся научиться умножать «длинное» число на «короткое» ($k \le 100$). «Короткими» обычно называют числа, которые помещаются в переменную одного из стандартных типов данных.

Попробуем сначала выполнить такое умножение на примере. Предположим, что в каждом элементе массива хранятся 6 цифр «длинного» числа, т. е. используется система счисления с основанием $d = 1\,000\,000$. Тогда число $[A]$=12345678901734567 хранится в трёх элементах:

	2	1	0
A	12345	678901	734567

Пусть $k = 3$. Начинаем умножать с младшего разряда: $734567 \cdot 3 = 2203701$. В нулевом разряде могут находиться только 6 цифр, значит, старшая двойка перейдёт в перенос в следующий разряд. В программе для выделения переноса r можно использовать целочисленное деление на основание системы счисления d. Остаток от деления — это то, что остаётся в текущем разряде. Поэтому получаем:

```
s:=A[0]*k                    s:=A[0]*k;
A[0]:=mod(s,d)               A[0]:=s mod d;
r:=div(s,d)                  r:=s div d;
```

Для следующего разряда будет всё то же самое, только в первой операции к произведению нужно добавить перенос из предыдущего разряда, который был записан в переменную r. Приняв в самом начале $r = 0$, запишем умножение «длинного» числа на «короткое» в виде цикла по всем элементам массива, от $A[0]$ до $A[N]$:

```
r:=0                         r:=0;
нц для i от 0 до N           for i:=0 to N do begin
   s:=A[i]*k+r                  s:=A[i]*k+r;
   A[i]:=mod(s,d)               A[i]:=s mod d;
   r:=div(s,d)                  r:=s div d
кц                           end;
```

В свою очередь, эти действия нужно выполнить в другом (внешнем) цикле для всех *k* от 2 до 100:

```
нц для k от 2 до 100        for k:=2 to 100 do begin
  . . .                        ...
кц                          end;
```

После этого в массиве *A* будет находиться искомое значение 100!, остаётся вывести его на экран. Нужно учесть, что в каждой ячейке хранятся 6 цифр, поэтому в массиве

	2	1	0
A	1	2	3

хранится значение 1000002000003, а не 123. Кроме того, старшие нулевые разряды выводить на экран не надо. Поэтому при выводе требуется:

1) найти первый (старший) ненулевой разряд числа;
2) вывести это значение без лидирующих нулей;
3) вывести все следующие разряды, добавляя лидирующие нули до 6 цифр.

Поскольку мы знаем, что число не равно нулю, старший ненулевой разряд можно найти в таком цикле[1]:

```
i:=N                        i:=N;
нц пока A[i]=0              while A[i]=0 do
  i:=i-1                      i:=i-1;
кц
```

Старший разряд выводим обычным образом (без лидирующих нулей):

```
вывод A[i]                  write(A[i]);
```

Для остальных разрядов будем использовать специальную процедуру Write6:

```
нц для k от i-1 до 0 шаг -1   for k:=i-1 downto 0 do
  Write6(A[k])                   Write6(A[k]);
кц
```

[1] Подумайте, что изменится, если выводимое число может быть нулевым.

Эта процедура последовательно выводит цифры десятичного числа, начиная с сотен тысяч и кончая единицами:

```
алг Write6 (цел x)
нач
  цел M, xx
  xx:=x
  M:=100000
  нц пока M>0
    вывод div(xx,M)
    xx:=mod(xx,M)
    M:=div(M, 10)
  кц
кон
```

```
procedure Write6(x: integer);
var M: integer;
begin
  M:=100000;
  while M>0 do begin
    write(x div M);
    x:=x mod M;
    M:=M div 10
  end
end;
```

Для того чтобы разобраться, как она работает, выполните «ручную прокрутку» при различных значениях x (например, возьмите $x = 1$, $x = 123$ и $x = 123456$).

Вопросы и задания

1. Какие преимущества и недостатки имеет алгоритм «решето Эратосфена» по сравнению с проверкой каждого числа на простоту?
2. Что такое «длинные» числа?
3. В каких случаях необходимо применять «длинную арифметику»?
4. Какое максимальное число можно записать в ячейку размером 64 бита? Рассмотрите варианты хранения чисел со знаком и без знака.
5. Можно ли использовать для хранения «длинного» числа символьную строку? Какие проблемы при этом могут возникнуть?
6. Почему неудобно хранить «длинное» число, записывая первую значащую цифру в начало массива?
7. Почему неэкономично хранить по одной цифре в каждом элементе массива?
8. Сколько разрядов числа можно хранить в 16-битном элементе массива?
9. Объясните, какие проблемы возникают при выводе длинного числа. Как их можно решать?
10. Объясните работу процедуры Write6.

Задачи

1. Докажите, что если у числа k нет ни одного делителя в диапазоне от 2 до $\sqrt{k}$, то оно простое.

2. Напишите две программы, которые находят все простые числа от 1 до N двумя разными способами:

   а) проверкой каждого числа из этого интервала на простоту;

   б) используя решето Эратосфена.

   Сравните число шагов цикла (время работы) этих программ для разных значений N. Постройте для каждого варианта зависимость количества шагов от N, сделайте выводы о сложности алгоритмов.
3. Докажите, что в приведённой в параграфе программе вычисления 100! не будет переполнения при использовании 32-битных целых переменных.
4. Можно ли в программе вычисления 100! в одной ячейке массива хранить 9 цифр «длинного» числа?
5. Без использования программы определите, сколько нулей стоит в конце числа 100!
6. Соберите всю программу и вычислите 100!. Сколько цифр входит в это число?
7. Оформите вывод «длинного» числа на экран в виде отдельной процедуры. Учтите, что число может быть нулевым.

*8. Придумайте другой способ вывода «длинного» числа, использующий символьные строки.

9. Напишите процедуру для ввода «длинных» чисел из файла.
10. Напишите процедуры для сложения и вычитания длинных чисел.

*11. Напишите процедуры для умножения и деления «длинных» чисел.

*12. Напишите процедуру для извлечения квадратного корня из «длинного» числа.

§ 39
Структуры (записи)

Зачем нужны структуры?

Представим себе базу данных библиотеки, в которой хранится информация о книгах. Для каждой из них нужно запомнить автора, название, год издания, количество страниц, число экземпляров и т. д. Как хранить эти данные?

Поскольку книг много, нужен массив. Но в массиве используются элементы одного типа, тогда как информация о книгах разнородна, она содержит целые числа и символьные строки разной длины. Конечно, можно разбить эти данные на несколько массивов (массив авторов, массив названий и т. д.) так, чтобы *i*-й элемент каждого массива относился к книге с номером *i*. Но такой подход оказывается слишком неудобным и ненадёжным. Например, при сортировке нужно переставлять элементы всех массивов (отдельно!) и можно легко ошибиться и нарушить связь данных.

Возникает естественная идея — объединить все данные, относящиеся к книге, в единый блок памяти, который в программировании называется структурой.

Структура — это тип данных, который может включать несколько **полей — элементов разных типов (в том числе и другие структуры).**

В Паскале структуры по традиции называют **записями** (англ. *record* — запись). Далее в этой главе мы будем использовать возможности свободно распространяемого компилятора *FreePascal*.

Объявление структур

Как и любые переменные в Паскале, структуры необходимо объявлять. До этого мы работали с простыми типами данных (целыми, вещественными, логическими и символьными), а также с массивами этих типов. Вы знаете, что при объявлении переменных и массивов указывается их тип, поэтому для того, чтобы работать со структурами, нужно ввести новый тип данных.

Построим структуру, с помощью которой можно описать книгу в базе данных библиотеки. Будем хранить в структуре только[1]:

- фамилию автора (строка не более 40 символов);
- название книги (строка не более 80 символов);
- имеющееся в библиотеке количество экземпляров (целое число).

---

[1] Конечно, в реальной ситуации данных больше, но принцип не меняется.

Объявление такого **составного типа** имеет вид:

```
type
  TBook = record
    author: string[40];    {автор, строка}
    title: string[80];     {название, строка}
    count: integer;        {количество, целое}
  end;
```

Объявления типов данных начинаются с ключевого слова **type** (в переводе с англ. — тип) и располагаются выше блока объявления переменных. Имя нового типа — TBook — это удобное сокращение от английских слов Type Book (*тип книга*), хотя можно было использовать и любое другое имя, составленное по правилам языка программирования. Слово **record** означает, что этот тип данных — **структура (запись)**; далее перечисляются поля и указываются их типы. Объявление структуры заканчивается ключевым словом **end**.

Обратите внимание, что для строк *author* и *title* указан максимальный размер. Это сделано для того, чтобы точно определить, сколько места нужно выделить на них в памяти.

В результате такого объявления никаких структур в памяти не создаётся: мы просто описали новый тип данных, чтобы транслятор знал, что делать, если мы захотим его использовать.

Теперь можно использовать тип TBook так же, как и простые типы, для объявления переменных и массивов:

```
const N = 100;
var B: TBook;
    Books: array[1..N] of TBook;
```

Здесь введена переменная *B* типа TBook и массив *Books*, состоящий из элементов того же типа.

Иногда бывает нужно определить размер одной структуры. Для этого используется стандартная функция sizeof, которой можно передать имя типа, а также переменную или массив:

```
Writeln(sizeof(TBook));
Writeln(sizeof(B));
Writeln(sizeof(Books));
```

Первые две команды выведут на экран размер одной структуры (124 байта), а последняя — размер выделенного массива из 100 структур. Размер структуры вызывает некоторые вопросы:

каждый элемент строки занимает 1 байт, а целое число — 2 байта, поэтому простой подсчёт дает значение 40 + 80 + 2 = 122. Откуда появились ещё 2 байта? Дело в том, что строка *author* из 40 символов фактически занимает 41 байт, а строковое поле *title* — 81 байт: 1 дополнительный байт расходуется на хранение фактического размера строки.

Обращение к полю структуры

Для того чтобы работать не со всей структурой, а с отдельными полями, используют так называемую **точечную нотацию**, разделяя точкой имя структуры и имя поля. Например, B.author обозначает «поле author структуры B», а Books[5].count — «поле count элемента массива Books[5]». Например, для определения размера полей в байтах, можно снова использовать функцию sizeof:

```
writeln(sizeof(B.author));
writeln(sizeof(B.title));
writeln(sizeof(B.count));
```

и мы увидим на экране числа 41, 81 и 2.

С полями структуры можно обращаться так же, как и с обычными переменными соответствующего типа. Можно вводить их с клавиатуры (или из файла):

```
readln(B.author);
readln(B.title);
readln(B.count);
```

присваивать новые значения:

```
B.author:='Пушкин А.С.';
B.title:='Полтава';
B.count:=1;
```

использовать при обработке данных:

```
p:=Pos(' ', B.author);
fam:=Copy(B.author, 1, p-1); { только фамилия }
B.count:=B.count - 1;         { взяли одну книгу}
if B.count=0 then
  writeln('Этих книг больше нет!');
```

и выводить на экран:

```
writeln(B.author, ' ',B.title, '. ', B.count,' шт.');
```

Работа с файлами

В программах, работающих с базами данных, необходимо читать массивы структур из файла и записывать в файл. Конечно, можно хранить структуры в текстовых файлах, например, записывая все поля одной структуры в одну строку и разделяя их каким-то символом-разделителем, который не встречается внутри самих полей.

Но есть более грамотный способ, который позволяет выполнять файловые операции проще и надёжнее (с меньшей вероятностью ошибки). Для этого нужно использовать файлы специального типа, которые называются **типизированными**. Все записываемые в них данные должны иметь одинаковый тип. В отличие от текстовых файлов данные в типизированных файлах хранятся во внутреннем формате, т. е. так, как они представлены в памяти компьютера во время работы программы. Например, можно сделать файл целых чисел или логических величин.

В данном случае нас интересует файл структур типа `TBook`, так что файловая переменная *F* для работы с типизированным файлом должна быть объявлена так:

```
var F: file of TBook;
```

Запись структуры в файл выполняется стандартным способом:

```
Assign(F,'books.dat');
Rewrite(F);
B.author:='Тургенев И.С.';
B.title:='Муму';
B.count:=2;
write(F,B);
Close(F);
```

Напомним, что процедура `Assign` связывает файловую переменную с файлом на диске, процедура `Rewrite` открывает файл на запись, а процедура `Close` — завершает запись на диск и закрывает файл.

Процедура `Write`, определив, что файловая переменная *F* связана с типизированным файлом структур, записывает в файл одну структуру во внутреннем формате. При попытке передать этой процедуре переменную другого типа произойдёт ошибка и аварийный останов программы.

С помощью цикла можно записать в файл весь массив структур:

```
for i:=1 to N do
  write(F,Books[i]);
```

Прочитать из файла одну структуру и вывести её поля на экран можно следующим образом:

```
Assign(F,'books.dat');
Reset(F);
Read(F,B);
Writeln(B.author,', ',B.title,', ',B.count);
Close(F);
```

Процедура `read`, получив ссылку *F* на типизированный файл, может принимать в качестве следующих параметров только структуры типа `TBook`.

Если заранее известно, сколько структур записано в файле, при чтении их в массив можно применить цикл с переменной:

```
for i:=1 to N do
  read(F,Books[i]);
```

Если же число структур неизвестно, нужно выполнять чтение до тех пор, пока файл не закончится, т. е. функция `Eof` не вернёт истинное значение:

```
i:=0;
while not eof(F) do begin
  i:=i+1;
  Read(F,Books[i]);
end;
```

Здесь целая переменная *i* играет роль счётчика: в ней на каждом шаге записано количество фактически прочитанных структур.

Сортировка

Для сортировки массива структур применяют те же методы, что и для сортировки массива простых переменных. Структуры обычно сортируют по возрастанию или убыванию одного из полей, которое называют **ключевым полем** или **ключом**, хотя можно, конечно, использовать и сложные условия, зависящие от нескольких полей **(составной ключ)**.

Отсортируем массив *Books* (типа `TBook`) по фамилиям авторов в алфавитном порядке. В данном случае ключом будет поле *author*. Предположим, что фамилия состоит из одного слова, а за ней через пробел следуют инициалы. Тогда сортировка *методом пузырька* выглядит так:

```
for i:=1 to N-1 do
  for j:=N-1 downto i do
    if Books[j].author>Books[j+1].author then
      begin
        B:=Books[j]; Books[j]:=Books[j+1];
        Books[j+1]:=B;
      end;
```

Здесь *i* и *j* — целочисленные переменные, а *B* — вспомогательная структура типа TBook.

Как вы знаете из курса 10 класса, при сравнении двух символьных строк они рассматриваются посимвольно до тех пор, пока не будут найдены первые различающиеся символы. Далее сравниваются коды этих символов по кодовой таблице. Так как код пробела меньше, чем код любой русской (и латинской) буквы, строка с фамилией «Волк» окажется выше в отсортированном списке, чем строка с более длинной фамилией «Волков», даже с учётом того, что после фамилии есть инициалы. Если фамилии одинаковы, сортировка происходит по первой букве инициалов, затем — по второй букве.

Возможно, что структуры требуется отсортировать так, чтобы не перемещать их в памяти. Например, они очень большие, и многократное копирование целых структур занимает много времени; или по каким-то другим причинам перемещать структуры нельзя. При таком ограничении нужно вывести на экран или в файл отсортированный список. В этом случае применяют **сортировку по указателям**, в которой используется дополнительный массив переменных специального типа — указателей.

Указатель — это переменная, в которой можно сохранить адрес любой переменной заданного типа.

То есть содержимое указателя — это адрес памяти. Чтобы избежать случайных ошибок, каждому указателю при объявлении присваивается тип данных, адреса которых он может хранить. Например, объявление

```
type PBook=^TBook;
```

вводит новый тип данных — указатель на структуру типа TBook. Адреса переменных других типов в такой указатель записывать

нельзя. Имя типа-указателя удобно начинать с буквы «*P*» от английского слова *pointer* — указатель.

Для сортировки массива *Books* нужно разместить в памяти массив таких указателей и одну вспомогательную переменную, которая будет использована при сортировке:

```
var p: array[1..N] of PBook;
    p1: PBook;
```

Следующий этап — расставить указатели так, чтобы *i*-й указатель был связан с *i*-й структурой из массива *Books*:

```
for i:=1 to N do p[i]:=@Books[i];
```

Знак @ обозначает операцию взятия адреса, т. е. в указатель записывается адрес структуры.

Для того чтобы от указателя перейти к объекту, на который он ссылается, используют операцию ^. Например, в нашем случае (после показанной выше начальной установки указателей) запись `p[i]^` обозначает то же самое, что и `Books[i]`, а `p[i]^.title` — то же самое, что и `Books[i].title`.

Теперь можно перейти к сортировке. Рассмотрим идею на примере массива из трёх структур. Сначала указатели стоят по порядку (рис. 6.1, *а*). В результате сортировки нужно переставить их так, чтобы *p*[1] указывал на первую структуру в отсортированном списке, *p*[2] — на вторую и т. д. (рис. 6.1, *б*).

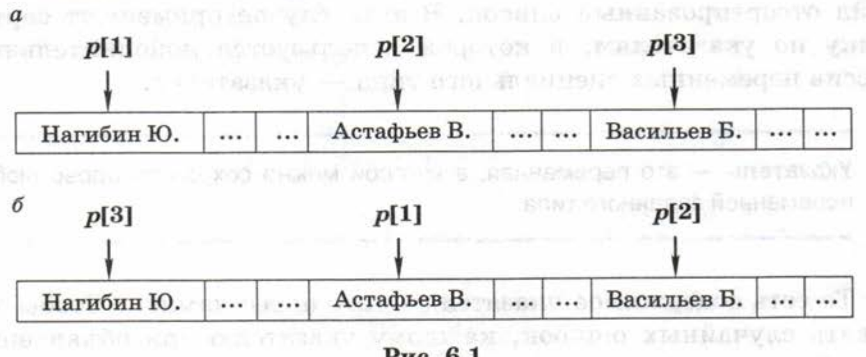


Рис. 6.1

Обратите внимание, что при этом сами структуры в памяти не перемещаются.

При сортировке обращение к полям структур идёт через указатели, и меняются местами тоже указатели, а не сами структуры:

```
for i:=1 to N-1 do
  for j:=N-1 downto i do
    if p[j]^.author > p[j+1]^.author then
      begin
        p1:=p[j]; p[j]:=p[j+1];
        p[j+1]:=p1;
      end;
```

Здесь использован *метод пузырька*, а *p*1 — это временная переменная типа PBook, которая служит для перестановки указателей.

Теперь можно вывести отсортированные данные, обращаясь к ним через указатели (а не через массив *Books*):

```
for i:=1 to N do
  writeln(p[i]^.author, '; ',
          p[i]^.title, '; ',
          p[i]^.count);
```

Вопросы и задания

1. Что такое структура? В чём её отличие от массива?
2. В каких случаях использование структур даёт преимущества? Какие именно? Приведите примеры.

3. Как объявляется новый тип данных в Паскале? Выделяется ли при этом память?
4. Как обращаются к полю структуры? Расскажите о точечной нотации.
5. Как определить, сколько байтов памяти выделяется на структуру?
6. Что такое типизированный файл? Чем он отличается от текстового?
7. Как работать с типизированными файлами?
8. Как можно сортировать структуры?
9. В каких случаях при сортировке желательно не перемещать структуры в памяти?
10. Что такое указатель?
11. Как записать в указатель адрес переменной?
12. Как обращаться к полям структуры через указатель?
13. Как используются указатели при сортировке?

Подготовьте сообщение

а) «Структуры в языке Си»
б) «Структуры в языке Javascript»

Задачи

1. Опишите структуру, в которой хранится информация о:

а) видеозаписи;
б) сотруднике фирмы;
в) самолёте;
г) породистой собаке.

2. Постройте программу, которая работает с базой данных в виде типизированного файла. Ваша СУБД (система управления базой данных) должна иметь следующие возможности:
а) просмотр записей;
б) добавление записей;
в) удаление записей;
г) сортировка по одному из полей (через указатели).

§ 40
Динамические массивы

Что это такое?

Когда мы объявляем массив, место для него выделяется во время трансляции, т. е. до выполнения программы. Такой массив называется **статическим**.

В то же время иногда размер данных заранее неизвестен. Например, пусть в файле записан массив чисел, которые нужно отсортировать. Их количество неизвестно, но известно, что такой массив помещается в оперативную память. В этом случае есть два варианта: 1) выделить заранее максимально большой блок памяти и 2) выделять память уже *во время выполнения программы* (т. е. **динамически**), когда станет известен необходимый размер массива.

Другой пример — задача составления *алфавитно-частотного словаря*. В файле находится список слов. Нужно вывести в другой файл все различные слова, которые встречаются в файле, и определить, сколько раз встречается каждое слово. Здесь проблема состоит в том, что нужный размер массива можно узнать только тогда, когда все различные слова будут найдены и, таким образом, задача решена. Поэтому нужно сделать так, чтобы массив мог «расширяться» в ходе работы программы.

Эти задачи приводят к понятию **динамических структур данных**, которые позволяют во время выполнения программы:

- создавать новые объекты в памяти;
- изменять их размер;
- удалять их из памяти, когда они не нужны.

Память под эти объекты выделяется в специальной области, которую обычно называют **«кучей»** (англ. *heap*).

Размещение в памяти

Задача 1. Требуется ввести с клавиатуры целое значение *N*, затем *N* целых чисел и вывести на экран эти числа в порядке возрастания.

Поскольку для сортировки все числа необходимо удерживать в памяти, нужно заводить массив, в который будут записаны все элементы. Поэтому алгоритм решения задачи на псевдокоде выглядит так:

```
прочитать данные из файла в массив
отсортировать их по возрастанию
вывести массив на экран
```

Все эти операции для обычных массивов подробно рассматривались в курсе 10 класса, поэтому здесь мы остановимся только на главной проблеме: как разместить в памяти массив, размер которого до выполнения программы неизвестен.

Для подобных случаев в версии языка Паскаль, которая поддерживается в среде *FreePascal*, существуют **динамические массивы**, которые объявляются без указания размера:

```
var A: array of integer;
```

Использовать сразу такой массив нельзя, поскольку его размер неизвестен. Попытка обращения к элементу, например *A*[1], вызывает ошибку и аварийный останов программы.

Когда значение переменной *N* введено, можно фактически разместить массив в памяти, используя процедуру SetLength (англ. *set length* — установить длину):

```
SetLength(A,N);
```

Далее массив A используется так же, как и обычный (статический) массив. Остаётся один вопрос: в каком диапазоне находятся его индексы?

Вы помните, что границы изменения индексов обычного (статического) массива задаются при его объявлении, причём начальный индекс может быть любым. Индексы динамического массива *всегда начинаются с нуля*, так что к начальному элементу нужно обращаться как *A*[0], а к последнему — как *A*[*N*–1]. Например, чтение данных с клавиатуры выполняется в цикле:

```
for i:=0 to N-1 do read(A[i]);
```

Кроме того, массив «знает» свою длину, которая вычисляется с помощью стандартной функции Length (в переводе с англ. — длина), и максимальный индекс, который возвращает функция High (в переводе с англ. — высокий). Поэтому предыдущий цикл можно заменить на такой:

```
for i:=0 to High(A) do read(A[i]);
```

Размер массива (количество элементов в нём) можно вычислить как Length(A) или High(A)+1.

Как только массив станет не нужен, можно удалить его из памяти, установив нулевую длину:

```
SetLength(A,0);
```

Для такого (удалённого) массива нулевой длины функция Length(A) вернёт значение 0.

Таким же образом можно работать и с динамическими матрицами. Они объявляются как «массив массивов»:

```
var A: array of array of integer;
```

Для определения её размеров в процедуре SetLength нужно указать два параметра — количество строк и количество столбцов:

```
SetLength(A,4,3);
```

Функция High возвращает максимальный индекс строки (минимальный индекс всегда равен 0):

```
writeln(High(A));            {= 3}
```

Для определения границ изменения второго индекса (максимального номера столбца) нужно вызывать эту функцию для отдельной строки:

```
writeln(High(A[0]));         {= 2}
```

Использование в подпрограммах

Динамические массивы можно передавать как параметры подпрограмм (процедур и функций). Например, процедуру для вывода на экран целочисленного массива можно написать так:

```
procedure printArray(X: array of integer);
begin
  for i:=0 to High(X) do write(X[i], ' ');
end;
```

Динамические массивы можно передать в подпрограмму как изменяемые параметры (с помощью ключевого слова **var**). В этом случае все изменения, сделанные в подпрограмме, применяются к массиву, переданному вызывающей программой, а не к его копии.

Расширение массива

Задача 2. С клавиатуры вводятся натуральные числа, ввод заканчивается числом 0. Нужно вывести на экран эти числа в порядке возрастания.

Как и в предыдущей задаче, для сортировки нужно предварительно сохранить все числа в оперативной памяти (в массиве). Но проблема в том, что размер этого массива становится известен только тогда, когда будут введены все числа. Что же делать?

В первую очередь приходит в голову такой вариант: при вводе каждого следующего ненулевого числа расширять массив на 1 элемент и записывать введённое число в последний элемент массива:

```
read(x);
while x<>0 do begin
  SetLength(A,Length(A)+1);
  A[High(A)]:=x;
  read(x)
end;
```

Здесь x — это целая переменная. К счастью, при таком расширении массива значения всех существующих элементов сохраняются.

Чем плох такой подход? Дело в том, что память в «куче» выделяется блоками. Поэтому при каждом увеличении длины массива последовательно выполняются три операции:

1) выделение блока памяти нового размера;
2) копирование в этот блок всех «старых» элементов;
3) удаление «старого» блока памяти из «кучи».

Видно, что «накладные расходы» очень велики, т. е. мы заставляем компьютер делать слишком много вспомогательной работы.

Ситуацию можно немного исправить, если увеличивать массив не каждый раз, а, скажем, после каждых 10 введённых элементов. То есть когда все свободные элементы массива заполнены, к нему добавляется ещё 10 новых элементов. При этом нужно считать фактическое количество записанных в массив значений,

потому что определить их, как в предыдущей программе, через функцию Length(A), будет невозможно:

```
N:=0;
read(x);
while x<>0 do begin
  if N>High(A) then
    SetLength(A, Length(A)+10);
  A[N]:=x;
  N:=N+1;
  read(x)
end;
```

Здесь целая переменная N — это счётчик введённых чисел.

Теперь, когда все числа записаны в массив, можно отсортировать их любым известным методом, например методом пузырька или с помощью быстрой сортировки (эти алгоритмы изучались в 10 классе). Закончить программу вы можете самостоятельно.

Как это работает?

Чтобы грамотно применять динамические массивы, необходимо разобраться в том, как они работают. Для этого выведем на экран размер массива из 100 элементов (целых чисел):

```
SetLength(A, 100);
write(sizeof(A));
write(100*sizeof(integer));
```

Вы с удивлением обнаружите, что программа выводит числа 4 и 200, т. е. функция sizeof считает, что размер массива равен 4 байтам, хотя на самом деле 100 целых переменных должны занимать 200 байтов. Более того, величина sizeof(A) не зависит от фактического размера массива. Поэтому можно сделать вывод, что размер элементов тут вообще не учитывается.

На самом деле, в переменной A хранится *адрес* массива в памяти (рис. 6.2), который действительно занимает 4 байта. Таким образом, фактически A — это *указатель*. Если массив имеет

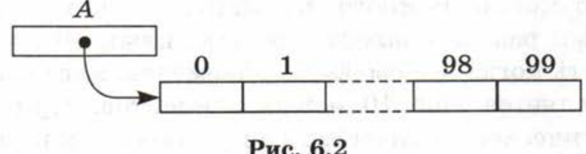


Рис. 6.2

нулевой размер, этот указатель равен нулю (нулевой адрес в Паскале обозначается nil).

Что из этого следует? Представьте, например, что мы построили структуру, которая состоит из массива (поле *data*) и количества используемых элементов в нём (поле *size*):

```
type TArray=record
       data: array of integer;
       size: integer;
     end;
```

Допустим создана переменная типа TArray:

```
var A: TArray;
```

для которой выделен в памяти внутренний массив *data* и заполнен целыми числами:

```
SetLength(A.data,10);
for i:=0 to 9 do A.data[i]:=i;
A.size:=10;
```

Что будет, если мы попытаемся сохранить такую структуру в типизированном файле? Несложно понять, что в файл запишутся только элементы структуры, т. е. адрес массива *data* и количество элементов *size*. Сами элементы не входят в структуру, поэтому сохранены не будут. Если после этого мы прочитаем из файла такую структуру, адрес *data* будет недействителен и использовать его нельзя. Поэтому при сохранении в файле структур с динамическими полями нужно принимать специальные меры по сохранению содержимого массивов.

Динамическая матрица — это указатель на массив указателей:

```
var A: array of array of integer;
```

Если применить к ней процедуру SetLength с одним параметром

```
SetLength(A,10);
```

то в памяти будет выделен массив указателей на строки, причём память под сами строки не выделяется. То есть A[1] (строка матрицы с индексом 1) — это указатель, к которому можно «привязать» динамический массив любого размера, например так:

```
for i:=0 to High(A) do
  SetLength(A[i],i+1);
```

Таким образом, мы получили матрицу, где все строки имеют разную длину:

```
writeln(High(A[0]));      {= 1}
writeln(High(A[9]));      {= 10}
```

Вопросы и задания

1. Приведите примеры задач, в которых использование динамических массивов даёт преимущества (какие именно?).
2. Что такое динамические структуры данных? Где выделяется память под эти данные?
3. Как объявить в программе динамический массив и задать его размер?
4. Как расширить массив в ходе работы программы? Не потеряются ли при этом уже записанные в нём данные?
5. Как определить границы изменения индексов динамического массива? Нужно ли хранить его размер в отдельной переменной?
6. Как удалить массив из памяти?
7. Как разместить в памяти динамическую матрицу?
8. Как передать динамический массив в подпрограмму?
9. Какие проблемы могут возникнуть при сохранении динамических массивов и матриц в файлах? Как вы предлагаете их решать?

Подготовьте сообщение

а) «Динамические массивы в языке Си»

б) «Динамические массивы в языке Javascript»

в) «Списки в языке Python как динамические массивы»

Задачи

1. Напишите полные программы для решения задач, рассмотренных в тексте параграфа.
2. Введите с клавиатуры число N и вычислите все простые числа в диапазоне от 2 до N, используя решето Эратосфена.
3. Введите с клавиатуры число N и запишите в массив первые N простых чисел.
4. Введите с клавиатуры число N и запишите в массив первые N чисел Фибоначчи (напомним, что они задаются рекуррентной формулой $F_n = F_{n-1} + F_{n-2}$, $F_1 = F_2 = 1$).
5. Напишите функцию, которая находит максимальный элемент переданного ей динамического массива.
6. Напишите подпрограмму, которая находит максимальный и минимальный элементы переданного ей динамического массива (используйте изменяемые параметры).

7. Напишите рекурсивную функцию, которая считает сумму значений элементов переданного ей динамического массива.
8. Напишите функцию, которая сортирует значения переданного ей динамического массива, используя алгоритм «быстрой сортировки» (см. учебник для 10 класса).

§ 41
Списки

Что такое список?

Задача 1. В файле находится список слов, среди которых есть повторяющиеся. Каждое слово записано в отдельной строке. Требуется построить алфавитно-частотный словарь: все различные слова должны быть записаны в другой файл в алфавитном порядке, справа от каждого слова указано, сколько раз оно встречается в исходном файле.

Для решения задачи нам нужно составить *список*, в котором хранить пары «слово — количество». Список составляется по мере чтения файла, т. е. это динамическая структура.

Список — это упорядоченный набор элементов одного типа, для которых введены операции вставки (включения) и удаления (исключения).

Обычно используют **линейные списки,** в которых для каждого элемента (кроме первого) можно указать предыдущий, а для каждого элемента, кроме последнего, — следующий.

Вернёмся к нашей задаче построения алфавитно-частотного словаря. Алгоритм, записанный в виде псевдокода, может выглядеть так:

```
нц пока есть слова в файле
  прочитать очередное слово
  если оно есть в списке то
    увеличить на 1 счётчик для этого слова
  иначе
    добавить слово в список
    записать 1 в счётчик слова
  все
кц
```

Теперь нужно записать все шаги этого алгоритма с помощью операторов языка программирования.

Использование динамического массива

В нашем случае каждый элемент списка должен содержать пару значений: слово (символьную строку) и счётчик этих слов (целое число). Поэтому элементы списка — это структуры, тип которых можно описать так:

```
type TPair = record
       word: string;      {слово}
       count: integer;    {счётчик}
     end;
```

Для организации списка будем использовать динамические массивы *FreePascal*.

Здесь размер массива становится известен только в конце работы программы. Поэтому требуется динамический массив, состоящий из описанных выше структур. С ним нужно выполнять следующие операции:

- искать заданное слово в списке;
- увеличивать счётчик заданного слова на 1;
- вставлять слово в определённое место списка (так, чтобы сохранить алфавитный порядок).

Как и в предыдущем параграфе, будем расширять размер массива сразу на 10 элементов[1], чтобы не выделять память слишком часто.

Объявим структуру-список:

```
type TWordList = record
       data: array of TPair; {динамический массив}
       size: integer;       {количество элементов}
     end;
```

Напомним, что количество фактически используемых элементов массива *size* может быть меньше, чем количество элементов, размещённых в памяти.

Введём переменную L типа TWordList:

```
var L: TWordList;
```

В начале основной программы очистим список и установим для него нулевую длину:

[1] Возможны и другие варианты, например можно увеличивать размер массива в 2 раза.

```
SetLength(L.data,0);
L.size:= 0;
```

Основной цикл (чтение данных из файла и построение списка) можно записать так (для последующего объяснения строки в теле цикла пронумерованы):

```
while not eof(F) do begin
  readln(F,s);                                  {1}
  p:=Find(L,s);                                 {2}
  if p>=0 then                                  {3}
    Inc(L.data[p].count)                        {4}
  else begin                                    {5}
    p:=FindPlace(L,s);                          {6}
    InsertWord(L,p,s);                          {7}
  end;                                          {8}
end;
```

Здесь используются две вспомогательные переменные: символьная строка s (типа string) и целая переменная *p*. В строке 1 программы очередное слово читается из файла в строку *s*. Затем с помощью функции Find определяется, есть ли оно в списке (строка 2). Если есть (функция Find вернула существующий индекс), увеличиваем счётчик этого слова на 1 (строки 3–4) с помощью стандартной процедуры Inc.

Если в списке слова ещё нет (функция Find вернула –1), нужно найти место, куда его вставить, так чтобы не нарушился алфавитный порядок. Это делает функция FindPlace, которая должна возвращать номер элемента массива, *перед* которым нужно вставить прочитанное слово. Вставку выполняет процедура InsertWord.

Здесь встретилось обозначение с двумя точками: L.data[p].count. Вспомним, что *L* — это структура-список, у него есть поле-массив *data*. В этом массиве происходит обращение к элементу с номером *p*. Этот элемент — структура типа TPair, в составе которой есть поле *count*. Таким образом, L.data[p].count означает: «Поле *count* в составе *p*-го элемента массива *data*, который входит в структуру *L*».

Когда список готов, остаётся вывести его в выходной файл:

```
Assign(F,'output.dat');
Rewrite(F);
for p:=0 to L.size-1 do
  writeln(F,L.data[p].word,': ',L.data[p].count);
Close(F);
```

Для каждого элемента списка в файл выводится хранящееся в нём слово и через двоеточие — сколько раз оно встретилось в тексте.

Таким образом, нам остаётся написать функции Find и FindPlace, а также процедуру InsertWord.

Функция Find принимает список и слово, которое нужно искать. Из курса 10 класса вы знакомы с двумя алгоритмами поиска: линейным и двоичным. Здесь для простоты будем использовать линейный поиск. В цикле проходим все элементы (не забывая, что их нумерация в динамическом массиве начинается с нуля). Как только очередное слово списка совпадёт с образцом, возвращаем в качестве результата функции номер этого элемента. Если просмотрены все элементы и совпадения не было, функция вернёт −1.

```
function Find(L: TWordList; word: string): integer;
var i: integer;
begin
  Find:=-1;
  for i:=0 to L.size-1 do
    if L.data[i].word=word then begin
      Find:= i;
      break;
    end;
end;
```

Функция FindPlace также принимает в качестве параметров список и слово. Она находит место вставки нового слова в список, при котором сохраняется алфавитный порядок расположения слов. Результат функции — номер слова, перед которым нужно вставить заданное. Для этого нужно найти в списке слово, которое «больше» заданного. Если такое слово не найдено, новое слово вставляется в конец списка:

```
function FindPlace(L: TWordList; word: string):
           integer;
var i, p: integer;
begin
  p:=-1;
  for i:=0 to L.size-1 do
    if L.data[i].word > word then begin
      p:=i;
      break;
    end;
```

```
    if p < 0 then p:=L.size;
    FindPlace:=p;
  end;
```

Процедура `InsertWord` вставляет слово *word* в позицию *k* в список *L*:

```
  procedure InsertWord(var L: TWordList; k: integer;
                       word: string);
  var i: integer;
  begin
    IncSize(L);                                {1}
    for i:=L.size-1 downto k+1 do              {2}
      L.data[i]:= L.data[i-1];                 {3}
    L.data[k].word:= word;                     {4}
    L.data[k].count:= 1;                       {5}
  end;
```

Поскольку в список добавляется новый элемент, его размер увеличивается. Для этого введена процедура `IncSize`, которая вызывается в строке 1 (мы напишем её позже). Далее в цикле сдвигаем все последние элементы, включая элемент с номером *k*, на одну ячейку к концу массива (строки 2–3). Таким образом, элемент с номером *k* освобождается. В строке 4 в него записывается новое слово, а в строке 5 счётчик этого слова устанавливается равным 1.

Процедура `IncSize` увеличивает размер списка на 1 элемент. Когда нужный размер становится больше, чем размер динамического массива, массив расширяется сразу на 10 элементов.

```
  procedure IncSize(var L: TWordList);
  begin
    L.size:= L.size+1;
    if L.size > Length(L.data) then
      SetLength(L.data,Length(L.data)+10);
  end;
```

Процедура `IncSize` в программе должна располагаться выше вызывающей её процедуры `InsertWord`.

Приведём окончательную структуру программы:

```
  program AlphaList;
  { объявления типов TPair и TWordList }
  var F: text;
```

```
    s: string;
    L: TWordList;
    p: integer;
  { процедуры и функции }
begin
  SetLength(L.data,0);
  L.size:=0;
  Assign(F,'input.dat');
  Reset(F);
  { основной цикл: составление списка слов }
  Close(F);
{ вывод результата в файл }
end.
```

Блоки, выделенные серым фоном, уже были написаны ранее в этом параграфе.

Заметим, что если известно максимальное количество разных слов в файле (скажем, не более 1000), то же самое можно сделать и на основе обычного (статического) массива, в котором память выделена заранее на максимальное число элементов.

Модульность

При разработке больших программ нужно разделить работу между программистами так, чтобы каждый делал свой независимый блок (**модуль**). Все подпрограммы, входящие в модуль, должны быть связаны друг с другом, но слабо связаны с другими процедурами и функциями. В нашей программе в отдельный модуль можно вынести все операции со списком слов.

Модуль в языке Паскаль, в отличие от основной программы, начинается со слова unit, после которого ставится название модуля.

```
unit WordList;
interface
 ...
implementation
 ...
end.
```

В модуле два основных раздела: **interface** (интерфейс, общедоступная часть) и **implementation** (реализация, недоступная другим модулям). В разделе **interface** обычно размещают объявления типов данных, функций и процедур, а в разделе

implementation — программный код. В нашей программе модуль может выглядеть так:

```
unit WordList;
interface
  type TPair = record
         word: string;
         count: integer;
       end;
       TWordList = record
         data: array of TPair;
         size: integer;
       end;
function Find(L: TWordList; word: string): integer;
function FindPlace(L: TWordList;
                   word: string): integer;
procedure InsertWord(var L: TWordList; k: integer;
                     word: string);
implementation
{ процедуры и функции }
end.
```

В секции **interface** мы расположили объявление типов данных, которые будут нужны основной программе, и заголовки подпрограмм этого модуля, которые могут вызываться извне. Всё, что находится в секции **implementation**, скрыто от «внешнего мира». В частности, там могут быть внутренние подпрограммы, которые «видны» только внутри модуля (в нашем случае это процедура IncSize).

Структура модуля в чём-то подобна айсбергу: видна только «надводная часть» (**interface**), а значительно более весомая «подводная часть» (**implementation**) скрыта. За счёт этого все, кто используют модуль, могут не думать о том, как именно он выполняет свою работу. Это один из приёмов, которые позволяют справляться со сложностью больших программ.

Модуль подключается к основной программе (или к другому модулю) с помощью ключевого слова **uses**. Если программа использует несколько модулей, все они перечисляются через запятую после слова **uses**.

Наша основная программа, использующая модуль WordList, выглядит так:

```
program AlphaList;
uses WordList;  { подключение модуля }
var F: text;
    s: string;
    L: TWordList;
    p: integer;
begin
  {тело основной программы}
end.
```

Разделение программы на модули облегчает понимание и совершенствование программы, потому что каждый модуль можно разрабатывать, изучать и оптимизировать независимо от других. Кроме того, такой подход ускоряет трансляцию больших программ, так как каждый модуль транслируется отдельно, причём только в том случае, если он был изменён.

Связные списки

Линейный список иногда представляется в программе в виде **связного списка**, в котором каждый элемент может быть размещён в памяти в произвольном месте, но должен содержать ссылку (указатель) на следующий элемент. У последнего элемента эта ссылка нулевая (в Паскале — `nil`), она показывает, что следующего элемента нет. Кроме того, нужно хранить где-то (в указателе `Head`) адрес первого элемента («головы») списка, иначе список будет недоступен (рис. 6.3).

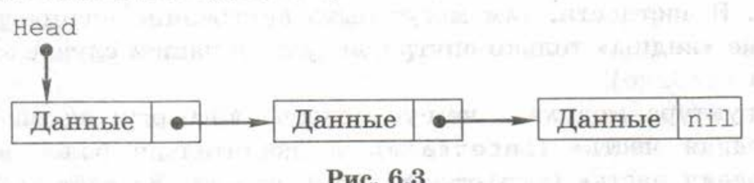


Рис. 6.3

Если замкнуть связный список в кольцо, так чтобы последний элемент содержал ссылку на первый, получается **циклический список** (рис. 6.4).

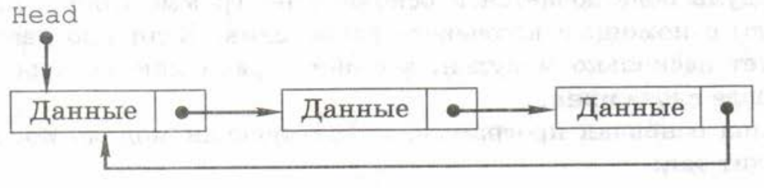


Рис. 6.4

Поскольку элементы связного списка содержат ссылки только на следующий элемент, к предыдущему перейти нельзя. Поэтому перебор возможен только в одном направлении. Этот недостаток устранён в **двусвязном списке**, где каждый элемент хранит адрес как следующего, так и предыдущего элемента (рис. 6.5).

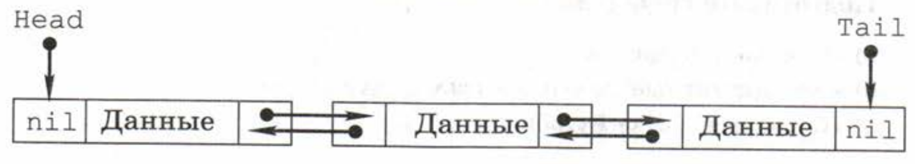


Рис. 6.5

Для такого списка обычно хранятся два адреса: «голова» списка (указатель `Head`) и его «хвост» (указатель `Tail`). Можно организовать и циклический двусвязный список. Использование двух указателей для каждого элемента приводит к дополнительному расходу памяти и усложнению всех операций со списком, потому что при добавлении и удалении элемента нужно правильно расставить оба указателя.

Применение связных списков приводит к более сложным алгоритмам, чем работа с динамическими массивами; рассматривать соответствующие программы мы не будем.

Вопросы и задания

1. Что такое список? Какие операции он допускает?
2. Верно ли, что элементы в списке упорядочены?
3. Какой метод поиска в списке можно использовать? Обсудите разные варианты.
4. Как добавить элемент в линейный список, сохранив заданный порядок сортировки?
5. Как можно представить список в программе? В каких случаях для этого можно использовать обычный массив?
6. Объясните запись `L.data[i].word`.
7. Что такое модуль? Зачем используют модули?
8. Как оформляется текст модуля? Как по нему отличить модуль от основной программы?
9. Что размещается в секциях **`interface`** и **`implementation`**?
10. Можно ли все переменные и подпрограммы поместить в секцию **`interface`**? Чем это плохо?
11. Как подключается модуль к основной программе или другому модулю?
12. Что такое связный список?

13. Что такое циклический список? Попытайтесь придумать задачу, где после завершения просмотра списка нужно начать просмотр заново.
14. Сравните односвязный и двусвязный списки. Покажите на примерах. В чём достоинства и недостатки одного и второго типов?

Подготовьте сообщение

а) «Списки в языке Си»

б) «Ассоциативные массивы в языке Javascript»

в) «Словари в языке Python»

Задачи

1. Постройте программу, которая составляет алфавитно-частотный словарь для заданного файла со списком слов. Используйте модуль, содержащий все операции со списком.

*2. В программе из задачи 1 измените функцию `Find` так, чтобы в ней использовался двоичный поиск.

3. В программе из задачи 2 объедините функции `Find` и `FindPlace`, заменив их на одну функцию. Если слово найдено в списке, функция работает так же, как `Find`: возвращает номер слова в списке. Если слово не найдено, функция должна вернуть отрицательное число: номер элемента массива, перед которым нужно вставить слово, со знаком минус.

*4. В программе из задачи 3 выведите все найденные слова в файл в порядке убывания частоты, т. е. в начале списка должны стоять слова, которые встречаются в файле чаще всех.

§ 42
Стек, очередь, дек

Что такое стек?

Представьте себе стопку книг (подносов, кирпичей и т. п.). С точки зрения информатики, её можно воспринимать как список элементов, расположенных в определённом порядке. Этот список имеет одну особенность — удалять и добавлять элементы можно только с одной («верхней») стороны. Действительно, для того чтобы вытащить какую-то книгу из стопки, нужно сначала снять все те книги, которые находятся на ней. Положить книгу сразу в середину тоже нельзя.

Стек (англ. *stack* — стопка) — это линейный список, в котором элементы добавляются и удаляются только с одного конца (англ. **LIFO**: *Last In – First Out* — последний пришёл — первым ушёл).

На рисунке 6.6 показаны примеры стеков вокруг нас, в том числе автоматный магазин и детская пирамидка.

Рис. 6.6

Как вы знаете из учебника для 10 класса, стек используется при выполнении программ: в нём хранятся адреса возврата из подпрограмм, параметры, передаваемые функциям и процедурам, а также локальные переменные.

Задача 1. В файле записаны целые числа. Нужно вывести их в другой файл в обратном порядке.

В этой задаче очень удобно использовать стек. Для стека определены две операции:

- добавить элемент на вершину стека (англ. *push* — втолкнуть);
- получить элемент с вершины стека и удалить его из стека (англ. *pop* — вытолкнуть).

Запишем алгоритм решения на псевдокоде. Сначала читаем данные и добавляем их в стек:

```
нц пока файл не пуст
  прочитать x
  добавить x в стек
кц
```

Теперь верхний элемент стека — это последнее число, прочитанное из файла. Поэтому остаётся «вытолкнуть» все записанные в стек числа, они будут выходить в обратном порядке:

```
нц пока стек не пуст
  вытолкнуть число из стека в x
  записать x в файл
кц
```

Использование динамического массива

Поскольку стек — это линейная структура данных с переменным количеством элементов, для создания стека в программе мы можем использовать динамический массив. Конечно, можно организовать стек из обычного (статического) массива, но его будет невозможно расширить сверх размера, выделенного при трансляции.

Для рассмотренной выше задачи 1 структура-стек содержит динамический целочисленный массив и количество используемых в нём элементов:

```
type TStack = record
       data: array of integer;
       size: integer;
     end;
```

Будем считать, что стек «растёт» от начала к концу массива, т. е. вершина стека — это последний элемент.

Для работы со стеком нужны две подпрограммы:

- процедура Push, которая добавляет новый элемент на вершину стека;
- функция Pop, которая возвращает верхний элемент стека и убирает его из стека.

Приведём эти подпрограммы:

```
procedure Push(var S: TStack; x: integer);
begin
  if S.size>High(S.data) then
    SetLength(S.data, Length(S.data)+10);
  S.data[S.size]:= x;
  S.size:=S.size+1;
end;
```

```
function Pop(var S:TStack): integer;
begin
  S.size:=S.size-1;
  Pop:=S.data[S.size];
end;
```

Обратите внимание, что здесь структура типа TStack изменяется внутри подпрограмм, поэтому этот параметр должен быть изменяемым (описан с помощью var).

Заметим, что если нам понадобится стек, который хранит данные другого типа (например, символы, символьные строки или структуры), в объявлении типа и в приведённых подпрограммах нужно просто заменить **integer** на нужный тип.

Кроме того, введём процедуру Initstack, которая заполняет поля структуры начальными значениями (выполняет инициализацию стека):

```
procedure InitStack(var S: TStack);
begin
  SetLength(S.data,0);
  S.size:=0;
end;
```

Теперь несложно написать цикл ввода данных в стек из файла:

```
InitStack(S);
while not eof(F) do begin
  read(F,x);
  Push(S,x);
end;
```

Здесь S — переменная типа TStack; F — файловая переменная, связанная с файлом, открытым на чтение; x — целая переменная. Вывод результата в файл выполняется так:

```
for i:=0 to S.size-1 do begin
  x:= Pop(S);
  writeln(F, x);
end;
```

Здесь i — целая переменная, а F — файловая переменная, связанная с файлом, открытым на запись.

Вычисление арифметических выражений

Вы не задумывались, как компьютер вычисляет арифметические выражения, записанные в такой форме: (5+15)/(4+7-1)?

Такая запись называется **инфиксной** — в ней знак операции расположен *между* операндами (данными, участвующими в операции). Инфиксная форма неудобна для автоматических вычислений из-за того, что выражение содержит скобки и его нельзя вычислить за один проход слева направо.

В 1920 г. польский математик Ян Лукасевич предложил **префиксную форму**, которую стали называть польской нотацией. В ней знак операции расположен *перед* операндами. Например, выражение `(5+15)/(4+7-1)` может быть записано в виде

```
/ + 5 15 - + 4 7 1.
```

Скобки здесь не требуются, так как порядок операций строго определён: сначала выполняются два сложения (`+ 5 15` и `+ 4 7`), затем вычитание, и, наконец, деление. Первой стоит последняя операция.

В середине 1950-х гг. была предложена **обратная польская нотация**, или **постфиксная форма** записи, в которой знак операции стоит *после* операндов:

```
5 15 + 4 7 + 1 - /
```

В этом случае также не нужны скобки, и выражение может быть вычислено за один просмотр с помощью стека следующим образом:

- если очередной элемент — число (или переменная), он записывается в стек;
- если очередной элемент — операция, то она выполняется с верхними элементами стека, и после этого в стек помещается результат выполнения этой операции.

Покажем, как работает этот алгоритм (стек «растёт» снизу вверх) (рис. 6.7).

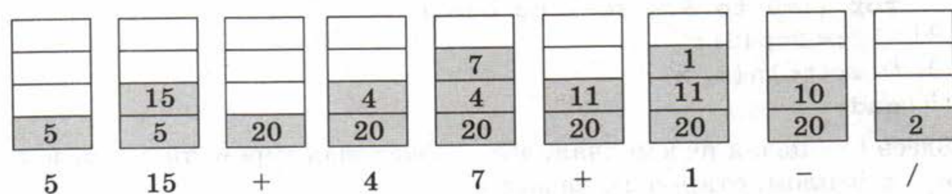


Рис. 6.7

В результате в стеке остаётся значение заданного выражения.

Скобочные выражения

Задача 2. Вводится символьная строка, в которой записано некоторое (арифметическое) выражение, использующее скобки трёх типов: `()`, `[]` и `{}`. Проверить, правильно ли расставлены скобки.

Например, выражение `()[{()[]}]` — правильное, потому что каждой открывающей скобке соответствует закрывающая, и вложенность скобок не нарушается. Выражения

`[()     [[[()     [()}     )(     ([)]`

неправильные. В первых трёх есть непарные скобки, а в последних двух не соблюдается вложенность скобок.

Начнём с задачи, в которой используется только один вид скобок. Её можно решить с помощью счётчика скобок. Сначала счётчик равен нулю. Строка просматривается слева направо, если очередной символ — открывающая скобка, то счётчик увеличивается на 1, если закрывающая — уменьшается на 1. В конце просмотра счётчик должен быть равен нулю (все скобки парные), кроме того, во время просмотра он не должен становиться отрицательным (должна соблюдаться вложенность скобок).

В исходной задаче (с тремя типами скобок) хочется завести три счётчика и работать с каждым отдельно. Однако это решение неверное. Например, для выражения `({[)}]` условия правильности выполняются отдельно для каждого вида скобок, но не для выражения в целом.

Задачи, в которых важна вложенность объектов, удобно решать с помощью стека. Нас интересуют только открывающие и закрывающие скобки, на остальные символы можно не обращать внимания.

Строка просматривается слева направо. Если очередной символ — открывающая скобка, нужно поместить её на вершину стека. Если это закрывающая скобка, то проверяем, что лежит на вершине стека: если там соответствующая открывающая скобка, то её нужно просто снять со стека. Если стек пуст или на вершине лежит открывающая скобка другого типа, выражение неверное и нужно закончить просмотр. В конце обработки правильной строки стек должен быть пуст. Кроме того, во время просмотра не должно быть ошибок. Работа такого алгоритма иллюстрируется на рисунке (для правильного выражения) (рис. 6.8).

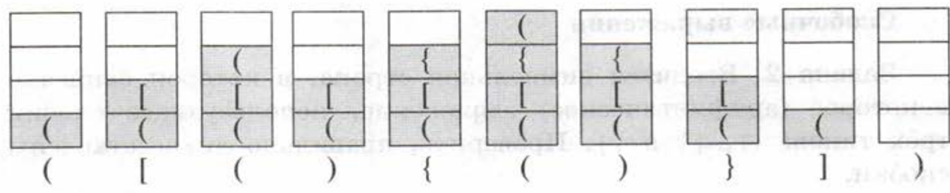


Рис. 6.8

Введём следующие переменные:

```
type TStack=record
       data: array of char;
       size: integer;
     end;
```

Отличие стека S от стека в предыдущей задаче только в том, что он содержит не целые числа, а символы (типа **char**). Поэтому приводить подпрограммы Push и Pop мы не будем, вы можете их переделать самостоятельно. Для удобства добавим только логическую функцию, которая возвращает значение True (истина), если стек пуст:

```
function isEmpty(S: TStack): boolean;
begin
  isEmpty:=(S.size=0);
end;
```

Введём строковые константы *L* и *R*, которые содержат все виды открывающих и соответствующих закрывающих скобок:

```
const L = '([{';
      R = ')]}';
```

Объявим переменные основной программы:

```
var S: TStack;
    p, i: integer;
    str: string;
    err: boolean;
    c: char;
```

Переменная *str* — это исходная строка. Логическая переменная *err* будет сигнализировать об ошибке. Сначала ей присваивается значение False («ложь»).

В основном цикле меняется целая переменная *i*, которая обозначает номер текущего символа, переменная *p* используется как вспомогательная:

```
for i:=1 to Length(str) do begin
  p:=Pos(str[i], L);                  {1}
  if p>0 then Push(S,str[i]);         {2}
  p:=Pos(str[i],R);                   {3}
  if p>0 then begin                   {4}
    if isEmpty(S) then err:=True          {5}
    else begin
      c:=Pop(S);                          {6}
      if p<>Pos(c,L) then err:=True       {7}
    end;
    if err then break                     {8}
  end
end;
```

Сначала мы ищем символ *str*[*i*] в строке *L*, т. е. среди открывающих скобок (строка 1). Если это действительно открывающая скобка, помещаем её в стек (2). Далее ищем символ среди закрывающих скобок (3). Если нашли, то в первую очередь проверяем, не пуст ли стек. Если стек пуст, выражение неверное и переменная err принимает истинное значение. Если в стеке что-то есть, снимаем символ с вершины стека в символьную переменную *c* (6). В строке (7) сравнивается тип (номер) закрывающей скобки *p* и номер открывающей скобки, найденной на вершине стека. Если они не совпадают, выражение неправильное, и в переменную *err* записывается значение True. Если при обработке текущего символа обнаружено, что выражение неверное (значение переменной *err* — True), нужно закончить цикл досрочно с помощью оператора break (8).

В конце программы остаётся вывести результат на экран:

```
if not err
  then writeln('Выражение правильное.')
  else writeln('Выражение неправильное.');
```

Очереди, деки

Все мы знакомы с принципом очереди: первый пришёл — первый обслужен (англ. **FIFO**: *First In — First Out*). Соответствующая структура данных в информатике тоже называется очередью.

Очередь — это линейный список, для которого введены две операции:
- добавление нового элемента в конец очереди;
- удаление первого элемента из очереди.

Очередь — это не просто теоретическая модель. Операционные системы используют очереди для организации сообщений между программами: каждая программа имеет свою очередь сообщений. Контроллеры жёстких дисков формируют очереди запросов ввода и вывода данных. В сетевых маршрутизаторах создаётся очередь из пакетов данных, ожидающих отправки.

Задача 3. Рисунок задан в виде матрицы A, в которой элемент $A[y, x]$ определяет цвет пикселя (x, y) на пересечении строки y и столбца x. Требуется перекрасить в цвет 2 одноцветную область, начиная с пикселя (x_0, y_0). На рисунке 6.9 показан результат такой заливки для матрицы из 5 строк и 5 столбцов с начальной точкой (2,1).

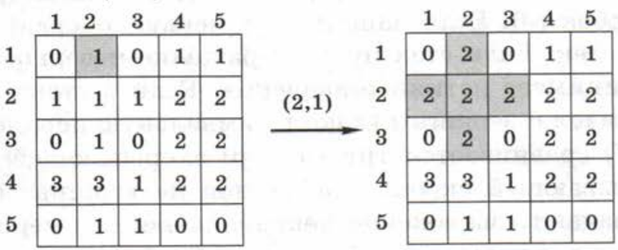


Рис. 6.9

Эта задача актуальна для графических программ. Один из возможных вариантов решения использует очередь, элементы которой — координаты пикселей (точек):

```
добавить в очередь точку (x0,y0)
запомнить цвет начальной точки
нц пока очередь не пуста
  взять из очереди точку (x,y)
  если A[y,x]=цвету начальной точки то
    A[y,x]:=2;
    добавить в очередь точку (x-1,y)
    добавить в очередь точку (x+1,y)
    добавить в очередь точку (x,y-1)
    добавить в очередь точку (x,y+1)
  все
кц
```

Конечно, в очередь добавляются только те точки, которые находятся в пределах рисунка (матрицы A). Заметим, что в этом алгоритме некоторые точки могут быть добавлены в очередь несколько раз (подумайте, когда это может случиться). Поэтому алгоритм можно несколько улучшить, как-то помечая точки, уже добавленные в очередь, чтобы не добавлять их повторно (попробуйте сделать это самостоятельно).

Две координаты точки связаны между собой, поэтому в программе лучше объединить их в структуру TPoint (от англ. *point* — точка), а очередь составить из таких структур:

```
Type TPoint = record
       x, y: integer;
     end;
     TQueue = record
       data: array of TPoint;
       size: integer
     end;
```

Для удобства построим функцию Point, которая формирует структуру типа TPoint по заданным координатам:

```
function Point(x,y: integer): TPoint;
begin
  Point.x:=x;
  Point.y:=y
end;
```

Для работы с очередью, основанной на динамическом массиве, введём две подпрограммы:

- процедура Put добавляет новый элемент в конец очереди; если нужно, массив расширяется блоками по 10 элементов;
- функция Get возвращает первый элемент очереди и удаляет его из очереди (обработку ошибки «очередь пуста» вы можете сделать самостоятельно); все следующие элементы сдвигаются к началу массива.

```
procedure Put(var Q: TQueue; pt: TPoint);
begin
  if Q.size > High(Q.data) then
    SetLength(Q.data, Length(Q.data)+10);
  Q.data[Q.size]:=pt;
  Q.size:=Q.size+1
end;
```

```
function Get(var Q:TQueue): TPoint;
var i: integer;
begin
  Get:=Q.data[0];
  Q.size:=Q.size-1;
  for i:=0 to Q.Size-1 do
    Q.data[i]:=Q.data[i+1]
end;
```

Остаётся написать основную программу. Объявляем константы и переменные:

```
const XMAX = 5; YMAX = 5;
      NEW_COLOR = 2;
var Q: TQueue;
  x0, y0, color: integer;
  A: array[1..YMAX,1..XMAX] of integer;
  pt: TPoint;
```

Предположим, что матрица A заполнена. Задаём исходную точку, с которой начинается заливка, запоминаем её «старый» цвет и добавляем эту точку в очередь:

```
x0:=2; y0:=1;
color:=A[y0,x0];
Put(Q,Point(x0,y0));
```

Основной цикл практически повторяет алгоритм на псевдокоде:

```
while not isEmpty(Q) do begin
  pt:=Get(Q);
  if A[pt.y,pt.x]=color then begin
    A[pt.y,pt.x]:=NEW_COLOR;
    if pt.x>1    then Put(Q,Point(pt.x-1,pt.y));
    if pt.x<XMAX then Put(Q,Point(pt.x+1,pt.y));
    if pt.y>1    then Put(Q,Point(pt.x,pt.y-1));
    if pt.y<YMAX then Put(Q,Point(pt.x,pt.y+1))
  end
end;
```

Здесь функция `isEmpty` — такая же, как и для стека: возвращает `True`, если очередь пуста (поле *size* равно нулю).

В приведённом примере начало очереди всегда совпадает с первым элементом массива (имеющим индекс 0). При этом удаление элемента из очереди неэффективно, потому что требуется

сдвинуть все оставшиеся элементы к началу массива. Существует и другой подход, при котором элементы очереди не передвигаются. Допустим, что мы знаем, что количество элементов в очереди всегда меньше N. Тогда можно выделить статический массив из N элементов (с индексами от 1 до N) и хранить в отдельных переменных номера первого элемента очереди («головы», англ. *head*) и последнего элемента («хвоста», англ. *tail*). На рисунке 6.10, *а* показана очередь из 5 элементов. В этом случае удаление элемента из очереди сводится просто к увеличению переменной Head (рис. 6,10, *б*).

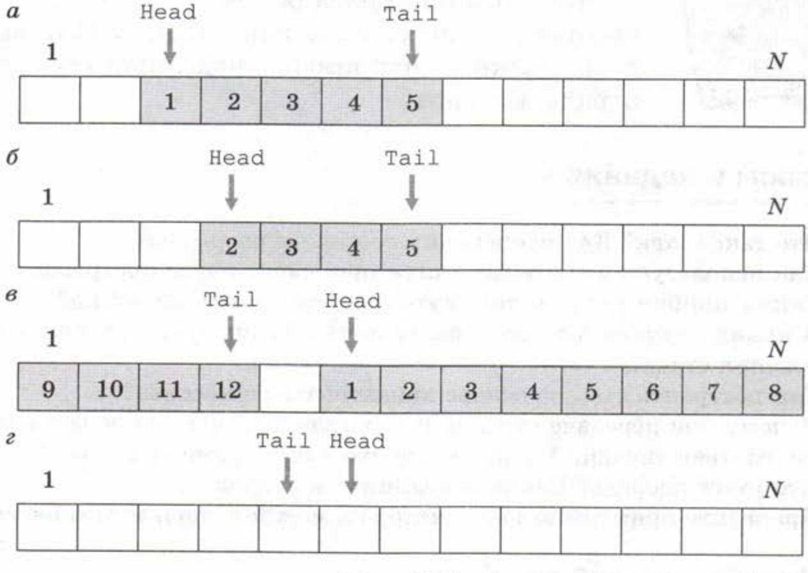


Рис. 6.10

При добавлении элемента в конец очереди переменная *Tail* увеличивается на 1. Если она перед этим указывала на последний элемент массива, то следующий элемент записывается в начало массива, а переменной *Tail* присваивается значение 1. Таким образом, массив оказывается замкнутым «в кольцо». На рисунке 6.10, *в* показана целиком заполненная очередь, а на рис. 6.10, *г* — пустая очередь. Один элемент массива всегда остаётся незанятым, иначе невозможно будет различить состояния «очередь пуста» и «очередь заполнена».

Отметим, что приведённая здесь модель описывает работу кольцевого буфера клавиатуры, который может хранить до 15 двухбайтных слов.

Кроме того, очередь можно построить на основе связного списка (см. § 41). Детали такой реализации можно найти в литературе или в Интернете.

Существует еще одна линейная динамическая структура данных, которая называется дек.

Дек — это линейный список, в котором можно добавлять и удалять элементы как с одного, так и с другого конца.

Из этого определения следует, что дек может работать и как стек, и как очередь. С помощью дека можно, например, моделировать колоду игральных карт.

Вопросы и задания

1. Что такое стек? Какие операции со стеком разрешены?
2. Как используется системный стек при выполнении программ?
3. Какие ошибки могут возникнуть при использовании стека?
4. В каких случаях можно использовать обычный массив для моделирования стека?
5. Как построить стек на основе динамического массива?
6. Почему при передаче стека в подпрограммы, приведённые в параграфе, соответствующий параметр должен быть изменяемым?
7. Что такое очередь? Какие операции она допускает?
8. Приведите примеры задач, в которых можно использовать очередь.

Подготовьте сообщение

а) «Моделирование стека и очереди в языке Си»
б) «Моделирование стека и очереди в языке Python»
в) «Моделирование очереди с помощью стеков»
г) «Очередь с приоритетом»

Задачи

1. Напишите программу, которая «переворачивает» массив, записанный в файл, с помощью стека. Размер массива неизвестен. Все операции со стеком вынесите в отдельный модуль.
2. Напишите программу, которая вычисляет значение арифметического выражения, записанного в постфиксной форме. Выражение вводится с клавиатуры в виде символьной строки.

3. Напишите программу, которая проверяет правильность скобочного выражения с четырьмя видами скобок: (), [], {} и <>. Все операции со стеком вынесите в отдельный модуль.

*4. Найдите в литературе или в Интернете алгоритм перевода арифметического выражения из инфиксной формы в постфиксную и напишите программу, которая решает эту задачу.

5. Напишите программу, которая выполняет заливку одноцветной области заданным цветом. Матрица, содержащая цвета пикселей, вводится из файла. Затем с клавиатуры вводятся координаты точки заливки и цвет заливки. На экран нужно вывести матрицу, которая получилась после заливки. Все операции с очередью вынесите в отдельный модуль.

*6. Перепишите программу из задачи 4 — используйте статический массив для организации очереди. Считайте, что в очереди может быть не более 100 элементов. Предусмотрите обработку ошибки «очередь переполнена».

*7. Напишите программу решения задачи о заливке области, помечая при этом точки, добавленные в очередь, чтобы не добавлять их повторно. В чём преимущества и недостатки такого алгоритма?

§ 43 Деревья

Что такое дерево?

Как вы знаете из учебника 10 класса, **дерево** — это структура, отражающая иерархию (отношения подчинённости, многоуровневые связи). Напомним некоторые основные понятия, связанные с деревьями.

Дерево состоит из **узлов** и связей между ними (они называются **дугами**). Самый первый узел, расположенный на верхнем уровне (в него не входит ни одна стрелка-дуга), — это **корень дерева**. Конечные узлы, из которых не выходит ни одна дуга, называются **листьями**. Все остальные узлы, кроме корня и листьев, — это промежуточные узлы.

Из двух связанных узлов тот, который находится на более высоком уровне, называется **родителем**, а другой — **сыном**. Корень — это единственный узел, у которого нет родителя; у листьев нет сыновей.

Используются также понятия «предок» и «потомок». **Потомок** какого-то узла — это узел, в который можно перейти по

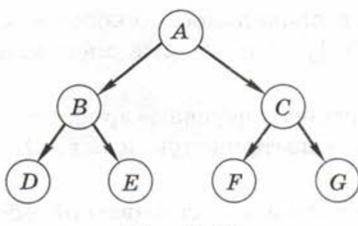


Рис. 6.11

стрелкам от узла-предка. Соответственно, **предок** какого-то узла — это узел, из которого можно перейти по стрелкам в данный узел.

В дереве на рис. 6.11 родитель узла E — это узел B, а предки узла E — это узлы A и B, для которых узел E — потомок. Потомками узла A (корня дерева) являются все остальные узлы.

Высота дерева — это наибольшее расстояние (количество дуг) от корня до листа. Высота дерева, приведённого на рис. 6.11, равна 2.

Формально **дерево** можно определить следующим образом:
1) пустая структура — это дерево;
2) дерево — это корень и несколько связанных с ним отдельных (не связанных между собой) деревьев.

Здесь множество объектов (деревьев) определяется через само это множество на основе простого базового случая (пустого дерева). Такой приём называется **рекурсией** (см. главу 8 учебника для 10 класса). Согласно этому определению, дерево — это рекурсивная структура данных. Поэтому можно ожидать, что при работе с деревьями будут полезны рекурсивные алгоритмы.

Чаще всего в информатике используются *двоичные* (или *бинарные*) деревья, т. е. такие, в которых каждый узел имеет не более двух сыновей. Их также можно определить рекурсивно.

Двоичное дерево:
1) пустая структура — это двоичное дерево;
2) двоичное дерево — это корень и два связанных с ним отдельных двоичных дерева (левое и правое поддеревья).

Деревья широко применяются в следующих задачах:
- поиск в большом массиве неменяющихся данных;
- сортировка данных;
- вычисление арифметических выражений;
- оптимальное кодирование данных (метод сжатия Хаффмана).

Деревья поиска

Известно, что для того, чтобы найти заданный элемент в неупорядоченном массиве из N элементов, может понадобиться N сравнений. Теперь предположим, что элементы массива организованы в виде специальным образом построенного дерева, например как показано на рис. 6.12.

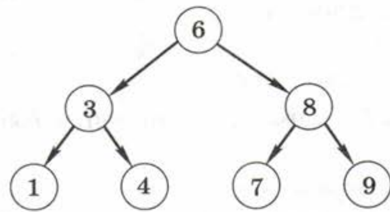


Рис. 6.12

Значения, связанные с узлами дерева, по которым выполняется поиск, называются **ключами** этих узлов (кроме ключа узел может содержать множество других данных). Перечислим важные свойства дерева, показанного на рис. 6.12:

- слева от каждого узла находятся узлы, ключи которых меньше или равны ключу данного узла;
- справа от каждого узла находятся узлы, ключи которых больше или равны ключу данного узла.

Дерево, обладающее такими свойствами, называется **двоичным деревом поиска**.

Например, пусть нужно найти узел, ключ которого равен 4. Начинаем поиск по дереву с корня. Ключ корня — 6 (больше заданного), поэтому дальше нужно искать только в левом поддереве и т. д. Если при линейном поиске в массиве за одно сравнение отсекается 1 элемент, здесь — сразу примерно половина оставшихся. Количество операций сравнения в этом случае пропорционально $\log_2 N$, т. е. алгоритм имеет асимптотическую сложность $O(\log_2 N)$. Конечно, нужно учитывать, что предварительно дерево должно быть построено. Поэтому такой алгоритм выгодно применять в тех случаях, когда данные меняются редко, а поиск выполняется часто (например, в базах данных).

Обход двоичного дерева

Обойти дерево — это значит «посетить» все узлы по одному разу. Если перечислить узлы в порядке их посещения, мы представим данные в виде списка.

Существуют несколько способов обхода дерева:

- КЛП — «корень – левый – правый» (обход в прямом порядке):

```
посетить корень
обойти левое поддерево
обойти правое поддерево
```

- ЛКП — «левый – корень – правый» (симметричный обход):

```
обойти левое поддерево
посетить корень
обойти правое поддерево
```

- ЛПК — «левый – правый – корень» (обход в обратном порядке):

```
обойти левое поддерево
обойти правое поддерево
посетить корень
```

Как видим, это рекурсивные алгоритмы. Они должны заканчиваться без повторного вызова, когда текущий корень — пустое дерево.

Рассмотрим дерево, которое может быть составлено для вычисления арифметического выражения `(1 + 4) * (9 - 5)` (рис. 6.13).

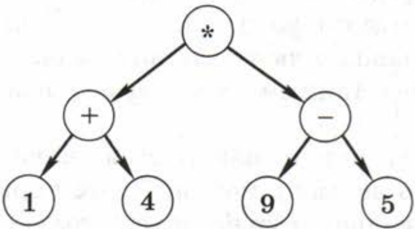


Рис. 6.13

Выражение вычисляется по такому дереву снизу вверх, т. е. посещение корня дерева — это последняя выполняемая операция.

Различные типы обхода дают последовательность узлов:

КЛП: `* + 1 4 - 9 5`
ЛКП: `1 + 4 * 9 - 5`
ЛПК: `1 4 + 9 5 - *`

В первом случае мы получили префиксную форму записи арифметического выражения, во втором — привычную нам инфиксную форму (только без скобок), а в третьем — постфиксную форму. Напомним, что в префиксной и в постфиксной формах скобки не нужны.

Обход КЛП называется «обходом в глубину», потому что сначала мы идём вглубь дерева по левым поддеревьям, пока не дойдём до листа. Такой обход можно выполнить с помощью стека следующим образом:

```
записать в стек корень дерева
нц пока стек не пуст
  выбрать узел V с вершины стека
  посетить узел V
  если у узла V есть правый сын то
    добавить в стек правого сына V
  все
  если у узла V есть левый сын то
    добавить в стек левого сына V
  все
кц
```

На рисунке 6.14 показано изменение состояния стека при таком обходе дерева, изображенного на рис. 6.13. Под стеком записана метка узла, который посещается (например, данные из этого узла выводятся на экран).

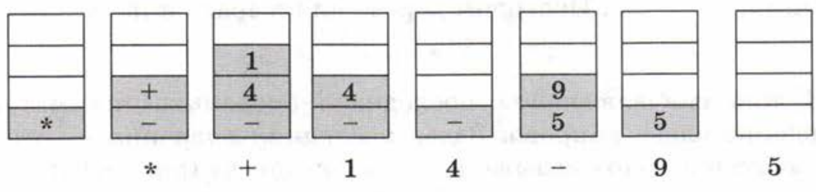


Рис. 6.14

Существует ещё один способ обхода, который называют **обходом в ширину**. Сначала посещают корень дерева, затем — всех его сыновей, затем — сыновей сыновей («внуков») и т. д., постепенно спускаясь на один уровень вниз. Обход в ширину для приведённого выше дерева даст такую последовательность посещения узлов:

```
* + - 1 4 9 5
```

Для того чтобы выполнить такой обход, применяют очередь. В очередь записывают узлы, которые необходимо посетить. На псевдокоде обход в ширину можно записать так:

```
записать в очередь корень дерева
нц пока очередь не пуста
  выбрать первый узел V из очереди
  посетить узел V
  если у узла V есть левый сын то
    добавить в очередь левого сына V
  все
  если у узла V есть правый сын то
    добавить в очередь правого сына V
  все
кц
```

Вычисление арифметических выражений

Один из способов вычисления арифметических выражений основан на использовании дерева. Сначала выражение, записанное в линейном виде (в одну строку), нужно «разобрать» и построить соответствующее ему дерево. Затем в результате прохода по этому дереву от листьев к корню вычисляется результат.

Для простоты будем рассматривать только арифметические выражения, содержащие числа и знаки четырёх арифметических операций: + – * /. Построим дерево для выражения

```
40 - 2 * 3 - 4 * 5
```

Нужно сначала найти последнюю операцию, просматривая выражение слева направо. Здесь последняя операция — это второе вычитание, оно оказывается в корне дерева (рис. 6.15).

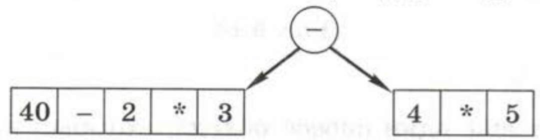


Рис. 6.15

Как выполнить этот поиск в программе? Известно, что операции выполняются в порядке **приоритета** (старшинства): сначала операции с более высоким приоритетом (слева направо), потом — с более низким (также слева направо). Отсюда следует важный вывод.

> **!** В корень дерева нужно поместить последнюю из операций с наименьшим приоритетом.

Теперь нужно построить таким же способом левое и правое поддеревья (рис. 6.16).

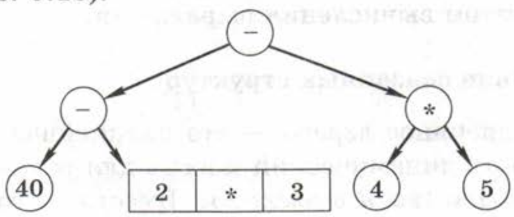


Рис. 6.16

Левое поддерево требует ещё одного шага (рис. 6.17).

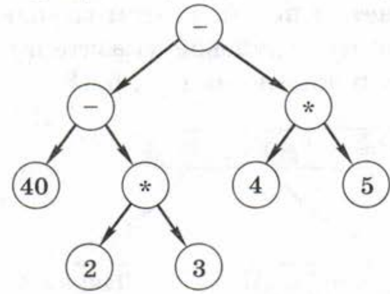


Рис. 6.17

Эта процедура рекурсивная, её можно записать в виде псевдокода:

```
найти последнюю выполняемую операцию
если операций нет то
  создать узел-лист
  выход
все
поместить найденную операцию в корень дерева
построить левое поддерево
построить правое поддерево
```

Рекурсия заканчивается, когда в оставшейся части строки нет ни одной операции, значит, там находится число (это лист дерева).

Теперь вычислим выражение по дереву. Если в корне находится знак операции, её нужно применить к результатам вычисления поддеревьев:

```
n1:=значение левого поддерева
n2:=значение правого поддерева
результат:=операция(n1,n2)
```

Снова получился рекурсивный алгоритм.

Возможен особый случай (на нём заканчивается рекурсия), когда корень дерева содержит число (т. е. это лист). Это число и будет результатом вычисления выражения.

Использование связанных структур

Поскольку двоичное дерево — это нелинейная структура данных, использовать динамический массив для размещения элементов не очень удобно (хотя возможно). Вместо этого будем использовать связанные узлы. Каждый такой узел — это структура, содержащая три области: область данных, ссылка на левое поддерево (указатель) и ссылка на правое поддерево (второй указатель). У листьев нет сыновей, в этом случае в указатели будем записывать значение nil (нулевой указатель). Дерево, состоящее из трёх таких узлов, показано на рис. 6.18.

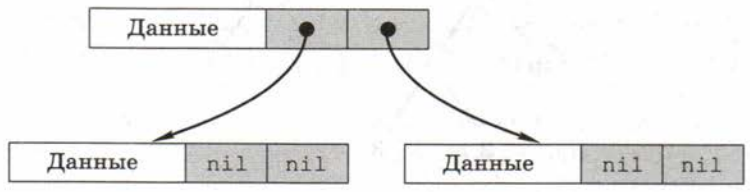


Рис. 6.18

В данном случае область данных узла будет содержать одно поле — символьную строку, в которую записывается знак операции или число в символьном виде.

Введём два новых типа: TNode — узел дерева, и PNode — указатель (ссылку) на такой узел:

```
type
  PNode = ^TNode;
  TNode = record
    data: string[20];
    left, right: PNode
  end;
```

Самый важный момент — выделение памяти под новую структуру. Предположим, что *p* — это переменная-указатель типа PNode. Для того чтобы выделить память под новую структуру и записать адрес выделенного блока в *p*, используется процедура *New* (англ. *new* — новый):

```
New(p);
```

Как программа определяет, сколько памяти нужно выделить? Чтобы ответить на этот вопрос, вспомним, что указатель p указывает на структуру типа TNode, размер которой и определяет размер выделяемого блока памяти.

Для освобождения памяти служит процедура Dispose (англ. *dispose* — ликвидировать):

```
Dispose(p);
```

В основной программе объявим одну переменную типа PNode — это будет ссылка на корень дерева:

```
var T: PNode;
```

Вычисление выражения сводится к двум вызовам функций:

```
T:=Tree(s);
writeln('Результат: ', Calc(T));
```

Здесь предполагается, что арифметическое выражение записано в символьной строке *s*, функция Tree строит в памяти дерево по этой строке, а функция Calc — вычисляет значение выражения по готовому дереву.

При построении дерева нужно выделять в памяти новый узел и искать последнюю выполняемую операцию — это будет делать функция LastOp. Она возвращает 0, если ни одной операции не обнаружено, в этом случае создаётся лист — узел без потомков. Если операция найдена, её обозначение записывается в поле *data*, а в указатели записываются адреса поддеревьев, которые строятся рекурсивно для левой и правой частей выражения:

```
function Tree(s: string): PNode;
var k: integer;
begin
  New(Tree);                {выделить память}
  k:=LastOp(s);
  if k=0 then begin         {создать лист}
    Tree^.data:=s;
    Tree^.left:=nil;
    Tree^.right:=nil;
  end
  else begin                {создать узел-операцию}
    Tree^.data:=s[k];
    Tree^.left:=Tree(Copy(s,1,k-1));
    Tree^.right:=Tree(Copy(s,k+1,Length(s)-k))
  end
end;
```

Функция Calc тоже будет рекурсивной:

```
function Calc(Tree: PNode): integer;
var n1, n2, res: integer;
begin
  if Tree^.left = nil then
    Val(Tree^.data, Calc, res)
  else begin
    n1:=Calc(Tree^.left);
    n2:=Calc(Tree^.right);
    case Tree^.data[1] of
      '+': Calc:=n1+n2;
      '-': Calc:=n1-n2;
      '*': Calc:=n1*n2;
      '/': Calc:=n1 div n2;
     else Calc:=MaxInt
    end
  end
end;
```

Если ссылка, переданная функции, указывает на лист (нет левого поддерева), то значение выражения — это результат преобразования числа из символьной формы в числовую (с помощью процедуры Val). В противном случае вычисляются значения для левого и правого поддеревьев и к ним применяется операция, указанная в корне дерева. В случае ошибки (неизвестной операции) функция возвращает значение MaxInt — максимальное целое число.

Осталось написать функцию LastOp. Нужно найти в символьной строке последнюю операцию с минимальным приоритетом. Для этого составим функцию, возвращающую приоритет операции (переданного ей символа):

```
function Priority(op: char): integer;
begin
  case op of
    '+','-': Priority:=1;
    '*','/': Priority:=2;
    else     Priority:=100
  end
end;
```

Сложение и вычитание имеют приоритет 1, умножение и деление — приоритет 2, а все остальные символы (не операции) — приоритет 100 (условное значение).

Функция LastOp может выглядеть так:

```
function LastOp(s: string): integer;
var i, minPrt: integer;
begin
  minPrt:=50;
  LastOp:=0;
  for i:=1 to Length(s) do
    if Priority(s[i])<=minPrt then begin
      minPrt:=Priority(s[i]);
      LastOp:=i
    end
end;
```

Обратите внимание, что в условном операторе указано нестрогое неравенство, чтобы найти именно *последнюю* операцию с наименьшим приоритетом. Начальное значение переменной *minPrt* можно выбрать любое между наибольшим приоритетом операций (2) и условным кодом не-операции (100). Тогда если найдена любая операция, условный оператор срабатывает, а если в строке нет операций, условие всегда ложно и в переменной *LastOp* остается начальное значение 0.

Хранение двоичного дерева в массиве

Двоичные деревья можно хранить в (динамическом) массиве почти так же, как и списки. Вопрос о том, как сохранить структуру (взаимосвязь узлов), решается следующим образом. Если нумерация элементов массива *A* начинается с 1, то сыновья элемента *A*[*i*] — это *A*[2**i*] и *A*[2**i*+1]. На рисунке 6.19 показан порядок расположения элементов в массиве для дерева, соответствующего выражению

```
40 - 2 * 3 - 4 * 5
```

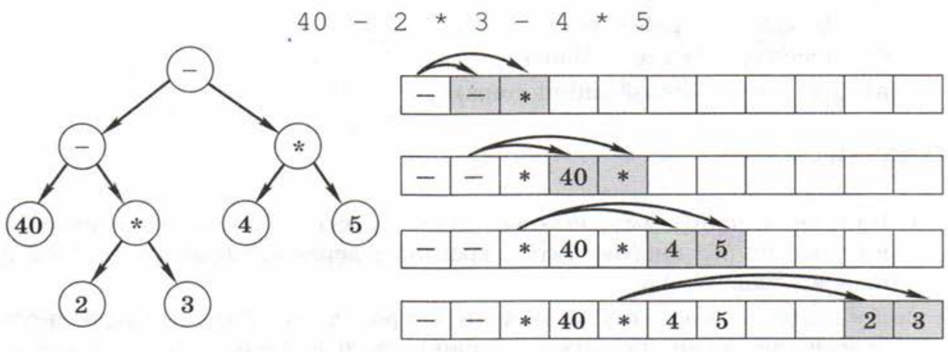


Рис. 6.19

Алгоритм вычисления выражения остаётся прежним, изменяется только метод хранения данных. Обратите внимание, что некоторые элементы остались пустыми, это значит, что их родитель — лист дерева.

Вопросы и задания

1. Дайте определение понятий «дерево», «корень», «лист», «родитель», «сын», потомок», «предок», «высота дерева».
2. Где используются структуры типа «дерево» в информатике и в других областях?
3. Объясните рекурсивное определение дерева.
4. Можно ли считать, что линейный список — это частный случай дерева?
5. Какими свойствами обладает дерево поиска?
6. Подумайте, как можно построить дерево поиска из массива данных.
7. Какие преимущества имеет поиск с помощью дерева?
8. Что такое обход дерева?
9. Какие способы обхода дерева вы знаете? Придумайте другие способы обхода.
10. Как строится дерево для вычисления арифметического выражения?
11. Как можно представить дерево в программе на Паскале?
12. Как указать, что узел дерева не имеет левого (правого) сына?
13. Как выделяется память под новый узел?
14. Как вы думаете, почему рекурсивные алгоритмы работы с деревьями получаются проще, чем нерекурсивные?

15. Как хранить двоичное дерево в массиве? Можно ли использовать такой приём для хранения деревьев, в которых узлы могут иметь больше двух сыновей? Приведите пример.

Подготовьте сообщение

а) «Деревья в языке Си»
б) «Деревья в языке Python»
в) «Диграммы связей (mind maps)»

Задачи

1. Напишите программу, которая вводит и вычисляет арифметическое выражение без скобок. Все операции с деревом вынесите в отдельный модуль.
2. Добавьте в программу из задачи 1 процедуры обхода построенного дерева так, чтобы получить префиксную и постфиксную записи введённого выражения.

*3. Добавьте в программу задачи 2 процедуру обхода дерева в ширину.

*4. Усовершенствуйте программу из задачи 1, чтобы она могла вычислять выражения со скобками.

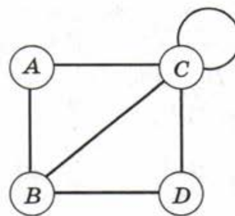
*5. Включите в вашу программу вычисления арифметического выражения без скобок обработку некоторых ошибок (например, два знака операций подряд). Поработайте в парах: обменяйтесь программами с соседом и попробуйте найти выражение, при котором его программа завершится аварийно (не выдаст сообщение об ошибке).

*6. Напишите программу вычисления арифметического выражения, которая хранит дерево в виде массива. Все операции с деревом вынесите в отдельный модуль.

§ 44
Графы

Что такое граф?

Как вы знаете из курса 10 класса, **граф** — это набор **узлов (вершин)** и связей между ними **(рёбер)**. Информацию об узлах и связях графа обычно хранят в виде таблицы специального вида — **матрицы смежности** (рис. 6.20).



	A	*B*	*C*	*D*
A	0	1	1	0
B	1	0	1	1
C	1	1	1	1
D	0	1	1	0

Рис. 6.20

Единица на пересечении строки *A* и столбца *B* означает, что между узлами *A* и *B* есть связь. Ноль указывает на то, что связи нет. Матрица смежности симметрична относительно главной диагонали (выделенные фоном ячейки в таблице). Единица на главной диагонали обозначает **петлю** — ребро, которое начинается и заканчивается в одной и той же вершине (в данном примере — в вершине *C*). Строго говоря, граф — это математический объект, а не рисунок. Конечно, его можно нарисовать на плоскости, но матрица смежности не даёт никакой информации о том, как

именно следует располагать узлы друг относительно друга. Для таблицы, приведённой на рис. 6.20, возможны, например, такие варианты (рис. 6.21).

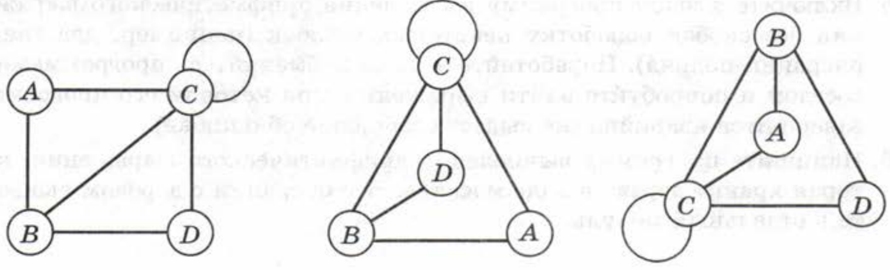
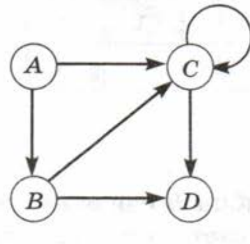


Рис. 6.21

В рассмотренном примере все узлы связаны, т. е. между любой парой узлов существует **путь** — последовательность рёбер, по которым можно перейти из одного узла в другой. Такой граф называется **связным**.

Вспоминая материал предыдущего параграфа, можно сделать вывод, что дерево — это частный случай связного графа, в котором нет замкнутых путей — **циклов**.

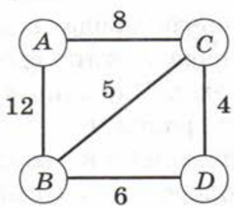
Если для каждого ребра указано направление, граф называют **ориентированным (орграфом)**. Рёбра орграфа называют **дугами**. Его матрица смежности не всегда симметричная. Единица, стоящая на пересечении строки *A* и столбца *B*, говорит о том, что существует дуга из вершины *A* в вершину *B* (рис. 6.22).



	A	**B**	**C**	**D**
A	0	1	1	0
B	0	0	1	1
C	0	0	1	1
D	0	0	0	0

Рис. 6.22

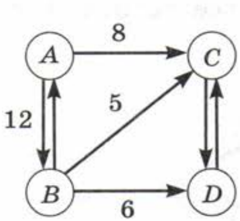
Часто с каждым ребром связывают некоторое число — **вес ребра**. Это может быть, например, расстояние между городами или стоимость проезда. Такой граф называется **взвешенным**. Информация о взвешенном графе хранится в виде **весовой матрицы**, содержащей веса рёбер (рис. 6.23).



	A	B	C	D
A		12	8	
B	12		5	6
C	8	5		4
D		6	4	

Рис. 6.23

У взвешенного орграфа весовая матрица может быть несимметрична относительно главной диагонали (рис. 6.24).



	A	B	C	D
A		12	8	
B	12		5	6
C				4
D			4	

Рис. 6.24

Если связи между двумя узлами нет, на бумаге можно оставить ячейку таблицы пустой, а при хранении в памяти компьютера записывать в неё условный код, например 0, –1 или очень большое число (∞), в зависимости от задачи.

«Жадные» алгоритмы

Задача 1. Известна схема дорог между несколькими городами. Числа на схеме (рис. 6.25) обозначают расстояния (дороги не прямые, поэтому неравенство треугольника может нарушаться). Нужно найти кратчайший маршрут из города A в город F.

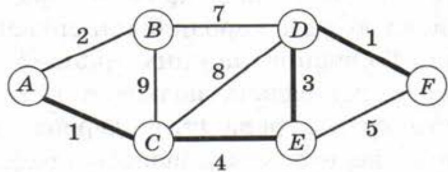


Рис. 6.25

Первая мысль, которая приходит в голову: на каждом шаге выбирать кратчайший маршрут до ближайшего города, в котором мы ещё не были. Для заданной схемы на первом этапе едем в город C (длина 1), далее — в E (длина 4), затем в D (длина 3) и, наконец, в F (длина 1). Общая длина маршрута равна 9.

Алгоритм, который мы применили, называется **«жадным»**. Он состоит в том, чтобы на каждом шаге многоходового процесса выбирать наилучший в данный момент вариант, не думая о том, что впоследствии этот выбор может привести к худшему решению.

Для данной схемы «жадный» алгоритм даёт оптимальное решение, но так будет далеко не всегда. Например, для той же задачи с другой схемой (рис. 6.26) «жадный» алгоритм даст маршрут $A–B–D–F$ длиной 10, хотя существует более короткий маршрут $A–C–E–F$ длиной 7.

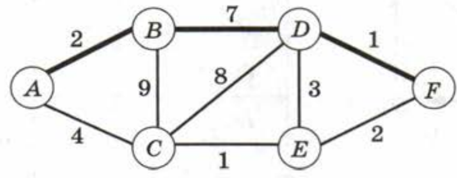


Рис. 6.26

«Жадный» алгоритм не всегда позволяет получить оптимальное решение.

Однако есть задачи, в которых «жадный» алгоритм всегда приводит к правильному решению. Одна из таких задач (её называют **задачей Прима–Крускала** в честь Р. Прима и Д. Крускала, которые независимо предложили её в середине XX века) формулируется так.

Задача 2. В стране Лимонии есть N городов, которые нужно соединить линиями связи. Между какими городами нужно проложить линии связи, чтобы все города были связаны в одну систему и общая длина линий связи была наименьшей?

В теории графов эта задача называется задачей построения **минимального остовного дерева** (т. е. дерева, связывающего все вершины). Остовное дерево для связного графа с N вершинами имеет $N - 1$ ребро.

Рассмотрим «жадный» алгоритм решения этой задачи, предложенный Крускалом:

1) начальное дерево — пустое;
2) на каждом шаге к дереву добавляется ребро минимального веса, которое ещё не входит в дерево и не приводит к появлению цикла в дереве.

На рисунке 6.27 показано минимальное остовное дерево для одного из рассмотренных выше графов (сплошные жирные линии).

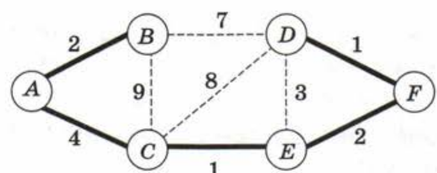


Рис. 6.27

Здесь возможна такая последовательность добавления рёбер: *CE*, *DF*, *AB*, *EF*, *AC*. Обратите внимание, что после добавления ребра *EF* следующее «свободное» ребро минимального веса — это *DE* (длина 3), но оно образует цикл с рёбрами *DF* и *EF*, поэтому не было включено в дерево.

При программировании этого алгоритма сразу возникает вопрос: как определить, что ребро ещё не включено в дерево и не образует цикла в нём? Существует очень красивое решение этой проблемы, основанное на «раскраске» вершин.

Сначала все вершины «раскрашиваются» в разные цвета (т. е. им присваиваются разные числовые коды):

```
for i:=1 to N do col[i]:=i;
```

Здесь N — количество вершин, а *col* — целочисленный массив с индексами от 1 до N.

Затем в цикле N–1 раз (именно столько рёбер нужно включить в дерево) выполняем следующие операции:

1) ищем ребро минимальной длины среди всех рёбер, *концы которых окрашены в разные цвета*;
2) найденное ребро (*iMin*,*jMin*) добавляется в список выбранных, и все вершины, имеющие цвет *col*[*jMin*], перекрашиваются в цвет *col*[*iMin*].

Приведём полностью основной цикл программы:

```
for k:=1 to N-1 do begin
    {поиск ребра с минимальным весом}
  min:=MaxInt;
  for i:=1 to N do
    for j:=1 to N do
      if (col[i]<>col[j]) and (W[i,j] < min)
        then begin
          iMin:=i; jMin:=j; min:=W[i,j];
        end;
    {добавление ребра в список выбранных}
  ostov[k,1]:=iMin;
  ostov[k,2]:=jMin;
    {перекрашивание вершин}
  for i:=1 to N do
    if col[i]=col[jMin] then
      col[i]:=col[iMin];
end;
```

Здесь W — целочисленная матрица размера $N \times N$ (индексы строк и столбцов начинаются с 1); *ostov* — целочисленный массив из N–1 строк и двух столбцов для хранения выбранных рёбер (для каждого ребра хранятся номера двух вершин, которые оно соединяет).

После окончания цикла остаётся вывести результат — рёбра из массива *ostov*:

```
for i:=1 to N-1 do
  writeln('(', ostov[i,1], ',', ostov[i,2], ')');
```

Кратчайшие маршруты

В предыдущем пункте мы познакомились с задачей выбора кратчайшего маршрута и увидели, что в ней «жадный» алгоритм не всегда даёт правильное решение. В 1960 г. Э. Дейкстра предложил алгоритм, позволяющий найти все кратчайшие расстояния от одной вершины графа до всех остальных и соответствующие им маршруты. Предполагается, что длины всех рёбер (расстояния между вершинами) положительные.

Рассмотрим уже знакомую схему, в которой не сработал «жадный» алгоритм (рис. 6.28).

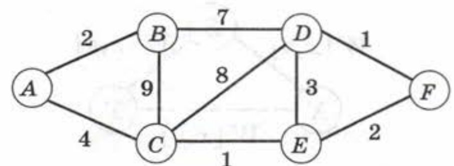


Рис. 6.28

Алгоритм Дейкстры использует дополнительные массивы: в одном (назовём его R) хранятся кратчайшие (на данный момент) расстояния от исходной вершины до каждой из вершин графа, а во втором (массив P) — вершина, из которой нужно «приехать» в данную вершину.

Сначала записываем в массив R расстояния от исходной вершины A до всех вершин, а в соответствующие элементы массива P — вершину A (рис. 6.29).

	A	B	C	D	E	F
R	0	2	4	∞	∞	∞
P	×	A	A			

Рис. 6.29

Знак ∞ обозначает, что прямого пути из вершины A в данную вершину нет (в программе вместо ∞ можно использовать очень большое число). Таким образом, вершина A уже рассмотрена и соответствующий элемент массива R выделен фоном. В первый элемент массива P записан символ ×, обозначающий начальную точку маршрута (в программе можно использовать несуществующий номер вершины, например 0).

Из оставшихся вершин находим вершину с минимальным значением в массиве R: это вершина B. Теперь проверяем пути, проходящие через эту вершину: не позволят ли они сократить маршрут к другим вершинам, которые мы ещё не посещали. Идея состоит в следующем: если сумма весов $W[x,z] + W[z,y]$ меньше, чем вес $W[x,y]$, то из вершины X лучше ехать в вершину Y не напрямую, а через вершину Z (рис. 6.30).

Проверяем наш граф: ехать из A в C через B невыгодно (получается путь длиной 11 вместо 4), а вот в вершину D можно про-

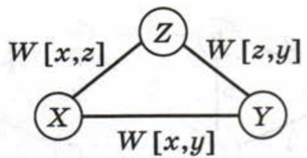


Рис. 6.30

ехать (путь длиной 9), поэтому запоминаем это значение вместо ∞ в массиве R и записываем вершину B на соответствующее место в массив P («в D приезжаем из B») (рис. 6.31).

	A	B	C	D	E	F
R	0	2	4	9	∞	∞
P	×	A	A	B		

Рис. 6.31

Вершины E и F по-прежнему недоступны.

Следующей рассматриваем вершину C (для неё значение в массиве R минимально). Оказывается, что через неё можно добраться до E (длина пути 5) (рис. 6.32).

	A	B	C	D	E	F
R	0	2	4	9	5	∞
P	×	A	A	B	C	

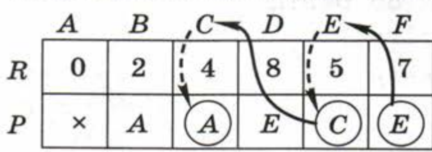
Рис. 6.32

Затем посещаем вершину E, которая позволяет достигнуть вершины F и улучшить минимальную длину пути до вершины D (рис. 6.33).

	A	B	C	D	E	F
R	0	2	4	8	5	7
P	×	A	A	E	C	E

Рис. 6.33

После рассмотрения вершин F и D таблица не меняется. Итак, мы получили, что кратчайший маршрут из A в F имеет длину 7, причём он приходит в вершину F из E. Как же получить весь маршрут? Нужно просто посмотреть в массиве P, откуда лучше всего ехать в E — выясняется, что из вершины C, а в вершину C — напрямую из начальной точки A (рис. 6.34).



	A	B	C	D	E	F
R	0	2	4	8	5	7
P	×	A	A	E	C	E

Рис. 6.34

Поэтому кратчайший маршрут A–C–E–F. Обратите внимание, что этот маршрут «раскручивается» в обратную сторону, от конечной вершины к начальной. Заметим, что полученная таблица содержит *все* кратчайшие маршруты из вершины A во все остальные вершины, а не только из A в F.

Алгоритм Дейкстры можно рассматривать как своеобразный «жадный» алгоритм: действительно, на каждом шаге из всех невыбранных вершин выбирается такая вершина X, что длина пути от A до X минимальна, если ехать только через уже выбранные вершины. Однако можно доказать, что это расстояние — действительно минимальная длина пути от A до X. Предположим, что для всех предыдущих выбранных вершин это свойство справедливо. При этом X — это ближайшая невыбранная вершина, которую можно достичь из начальной точки, проезжая только через выбранные вершины. Все остальные пути в X, проходящие через ещё не выбранные вершины, будут длиннее, поскольку все рёбра имеют положительную длину. Таким образом, найденная длина пути из A в X — минимальная.

После завершения алгоритма, когда все вершины выбраны, в массиве R находятся длины кратчайших маршрутов.

В программе объявим константу и переменные:

```
const N = 6;
var W: array[1..N,1..N] of integer;
    active: array [1..N] of boolean;
    R, P: array [1..N] of integer;
    i, j, min, kMin: integer;
```

Массив W — это весовая матрица, её удобно вводить из файла. Логический массив *active* хранит состояние вершин (просмотрена

или не просмотрена): если значение *active*[*i*] истинно, то вершина активна (ещё не просматривалась).

В начале программы присваиваем начальные значения (объяснение см. выше), сразу помечаем, что вершина 1 просмотрена (не активна), с неё начинается маршрут.

```
for i:=1 to N do begin
  active[i]:=True;
  R[i]:= W[1,i];
  P[i]:=1;
end;
active[1]:=False;
P[1]:=0;
```

В основном цикле, который выполняется $N - 1$ раз (так, чтобы все вершины были просмотрены), среди активных вершин ищем вершину с минимальным соответствующим значением в массиве R и проверяем, не лучше ли ехать через неё:

```
for i:=1 to N-1 do begin
  {поиск новой рабочей вершины R[j] -> min}
  min:=MaxInt; {максимальное целое число}
  for j:=1 to N do
    if active[j] and (R[j]<min) then begin
      min:=R[j];
      kMin:=j;
    end;
  active[kMin]:=False;
  {проверка маршрутов через вершину kMin}
  for j:=1 to N do
    if R[kMin]+W[kMin,j]<R[j] then begin
      R[j]:=R[kMin]+W[kMin,j];
      P[j]:=kMin
    end
end;
```

В конце программы выводим оптимальный маршрут в обратном порядке:

```
i:=N;
while i<>0 do begin {для начальной вершины P[i]=0}
  write(i:5);
  i:=P[i]   {переход к следующей вершине}
end;
```

Алгоритм Дейкстры, как мы видели, находит кратчайшие пути *из одной заданной вершины* во все остальные. Найти все кратчайшие пути (*из любой вершины в любую другую*) можно с помощью **алгоритма Флойда–Уоршелла**, основанного на той же самой идее сокращения маршрута (иногда бывает короче ехать через промежуточные вершины, чем напрямую):

```
for k:=1 to N
  for i:=1 to N
    for j:=1 to N
      if W[i,k]+W[k,j]<W[i,j] then
        W[i,j]:= W[i,k] + W[k,j];
```

В результате исходная весовая матрица графа W размером N × N превращается в матрицу, хранящую длины оптимальных маршрутов. Для того чтобы найти сами маршруты, нужно использовать еще одну дополнительную матрицу, которая выполняет ту же роль, что и массив P в алгоритме Дейкстры.

Некоторые задачи

С графами связаны некоторые классические задачи. Самая известная из них — **задача коммивояжёра** (бродячего торговца).

Задача 3. Бродячий торговец должен посетить N городов по одному разу и вернуться в город, откуда он начал путешествие. Известны расстояния между городами (или стоимость переезда из одного города в другой). В каком порядке нужно посещать города, чтобы суммарная длина пути (или стоимость) оказалась наименьшей?

Эта задача оказалась одной из самых сложных задач оптимизации. По сей день известно только одно надёжное решение — полный перебор вариантов, число которых равно факториалу значения $N - 1$. Это число с увеличением N растёт очень быстро, быстрее, чем любая степень N. Уже для $N = 20$ такое решение требует огромного времени вычислений: компьютер, проверяющий 1000 вариантов в секунду, будет решать задачу «в лоб» около четырёх миллионов лет. Поэтому математики прилагали большие усилия для того, чтобы сократить перебор — не рассматривать те варианты, которые заведомо не дают лучших результатов, чем уже полученные. В реальных ситуациях нередко оказы-

ваются полезными приближённые решения, которые не гарантируют точного оптимального решения, но позволяют получить приемлемый вариант.

Приведём формулировки ещё некоторых задач, которые решаются с помощью теории графов. Алгоритмы их решения вы можете найти в литературе или в Интернете.

Задача 4 (о максимальном потоке). Есть система труб, которые имеют соединения в N узлах. Один узел S является источником, ещё один T — стоком. Известны пропускные способности каждой трубы. Надо найти наибольший поток от источника к стоку.

Задача 5. Имеются N населённых пунктов, в каждом из которых живут p_i школьников ($i = 1, ..., N$). В каком пункте нужно разместить школу, чтобы общее расстояние, проходимое всеми учениками по дороге в школу, было минимальным?

Задача 6 (о наибольшем паросочетании). Есть M мужчин и N женщин. Каждый мужчина указывает несколько женщин (от 0 до N), на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от 0 до M), за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.

Вопросы и задания

1. Что такое граф?
2. Как обычно задаются связи узлов в графах?
3. Что такое матрица смежности?
4. Что такое петля? Как «увидеть» её в матрице смежности?
5. Что такое путь?
6. Какой граф называется связным?
7. Что такое орграф?
8. Как по матрице смежности отличить орграф от неориентированного графа?
9. Что такое взвешенный граф? Как может храниться в памяти информация о нём?
10. Что такое «жадный» алгоритм? Всегда ли он позволяет найти лучшее решение?

Подготовьте сообщение

а) «Работа с графами в языке Си»
б) «Работа с графами в языке Python»
в) «Жадный алгоритм в задаче коммивояжера»
г) «Метод ветвей и границ»
д) «Алгоритм Литтла»
е) «Задача о максимальном потоке»
ж) «Задача о кенигсбергских мостах»
з) «Использование графов для анализа данных в Интернете»
и) «Теория графов в практических задачах»

Задачи

1. Напишите программу, которая вводит из файла весовую матрицу графа и строит для него минимальное остовное дерево.
2. Оцените асимптотическую сложность алгоритма Прима–Крускала.
3. Напишите программу, которая вводит из файла весовую матрицу графа, затем вводит с клавиатуры номера начальной и конечной вершин и определяет кратчайший маршрут.
4. Напишите программу, которая вводит из файла весовую матрицу графа и определяет длины всех кратчайших маршрутов с помощью алгоритма Флойда–Уоршелла.
5. Оцените асимптотическую сложность алгоритмов Дейкстры и Флойда–Уоршелла.
6. Напишите программу, которая решает задачу коммивояжёра для 5 городов методом полного перебора. Можно ли использовать её для 50 городов?

*7. Напишите программу, которая решает задачу 5 (о размещении школы). Для определения кратчайших путей используйте алгоритм Флойда–Уоршелла. Весовую матрицу графа вводите из файла.

§ 45
Динамическое программирование

Что такое динамическое программирование?

Мы уже встречались с последовательностью чисел Фибоначчи (см. учебник для 10 класса):

$$F_1 = F_2 = 1;\ F_n = F_{n-1} + F_{n-2} \text{ при } n > 2.$$

Для их вычисления можно использовать рекурсивную функцию:

```
function Fib(N: integer): integer;
begin
  if N<3 then
    Fib:=1
  else Fib:=Fib(N-1)+Fib(N-2);
end;
```

Каждое из этих чисел связано с предыдущими, вычисление F_5 приводит к рекурсивным вызовам, которые показаны на рис. 6.35. Таким образом, мы два раза вычислили F_3, три раза — F_2 и два раза — F_1. Рекурсивное решение очень простое, но оно неоптимально по быстродействию: компьютер выполняет лишнюю работу, повторно вычисляя уже найденные ранее значения.

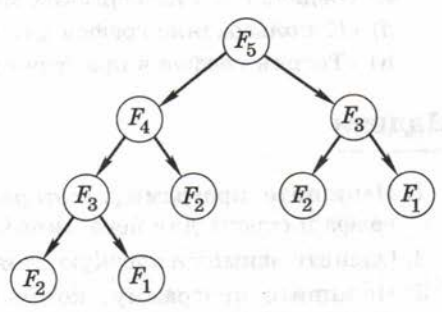


Рис. 6.35

Где же выход? Например, можно хранить все предыдущие числа Фибоначчи в массиве. Пусть этот массив называется F:

```
const N = 10;
var F: array[1..N] of integer;
```

Тогда для вычисления всех чисел Фибоначчи от F_1 до F_N можно использовать цикл:

```
F[1]:=1; F[2]:=1;
for i:=3 to N do
  F[i]:=F[i-1]+F[i-2];
```

Динамическое программирование — это способ решения сложных задач путем сведения их к более простым подзадачам того же типа.

Такой подход впервые систематически применил американский математик Р. Беллман при решении сложных многошаговых задач оптимизации. Его идея состояла в том, что оптимальная последовательность шагов оптимальна на любом участке.

Например, пусть нужно перейти из пункта A в пункт E через один из пунктов B, C или D (на рис. 6.36 числами обозначена «стоимость» маршрута).

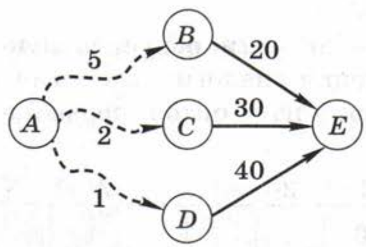


Рис. 6.36

Пусть уже известны оптимальные маршруты из пунктов B, C и D в пункт E (они обозначены сплошными линиями) и их «стоимость». Тогда для нахождения оптимального маршрута из A в E нужно выбрать вариант, который даст минимальную стоимость по сумме двух шагов. В данном случае это маршрут A–B–E, стоимость которого равна 25. Как видим, такие задачи решаются «с конца», т. е. решение начинается от конечного пункта.

В информатике динамическое программирование часто сводится к тому, что мы храним в памяти решения всех задач меньшей размерности. За счёт этого удаётся ускорить выполнение программы. Например, на одном и том же компьютере вычисление F_{45} с помощью рекурсивной функции требует около 8 секунд, а с использованием массива — менее 0,01 с.

Заметим, что в данной простейшей задаче можно обойтись вообще без массива:

```
f2:=1; f1:=1;
for i:=3 to N do begin
  FN:=f1+f2;
  f2:=f1;
  f1:=FN;
end;
```

Задача 1. Найти количество K_N цепочек, состоящих из N нулей и единиц, в которых нет двух стоящих подряд единиц.

При больших N решение задачи методом перебора потребует огромного времени вычисления. Для того чтобы использовать метод динамического программирования, нужно:

1) выразить K_N через предыдущие значения последовательности $K_1, K_2, ..., K_{N-1}$;
2) выделить массив для хранения всех предыдущих значений K_i ($i = 1, ..., N - 1$).

Самое главное — вывести **рекуррентную формулу**, выражающую K_N через решения аналогичных задач меньшей размерности. Рассмотрим цепочку из N битов, первый элемент которой — 0 (рис. 6.37).

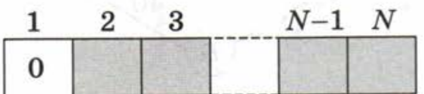


Рис. 6.37

Поскольку дополнительный 0 не может привести к появлению двух соседних единиц, подходящих последовательностей длины N с нулём в начале существует столько, сколько подходящих последовательностей длины $N - 1$, т. е. K_{N-1}. Если же первый символ — это 1, то вторым обязательно должен быть 0, а остальная цепочка из $N - 2$ битов должна быть правильной, без двух соседних единиц (рис. 6.38). Поэтому подходящих последовательностей длиной N с единицей в начале существует столько, сколько подходящих последовательностей длины $N - 2$, т. е. K_{N-2}.

Рис. 6.38

В результате получаем: $K_N = K_{N-1} + K_{N-2}$. Значит, для вычисления очередного числа нам нужно знать два предыдущих.

Теперь рассмотрим простые случаи. Очевидно, что есть две последовательности длины 1 (0 и 1), т. е. $K_1 = 2$. Далее, есть 3 подходящих последовательности длины 2 (00, 01 и 10), поэтому $K_2 = 3$. Легко понять, что решение нашей задачи — число Фибоначчи: $K_N = F_{N+2}$.

Поиск оптимального решения

Задача 2. В цистерне N литров молока. Есть бидоны объёмом 1, 5 и 6 литров. Нужно разлить молоко в бидоны так, чтобы все бидоны были заполнены и количество используемых бидонов было минимальным.

Человек, скорее всего, будет решать задачу перебором вариантов. Наша задача осложняется тем, что требуется написать программу, которая решает задачу для любого введённого числа N.

Самый простой подход — заполнять сначала бидоны самого большого размера (6 л), затем — меньшие и т. д. Это так называемый «жадный» алгоритм. Как вы знаете, он не всегда приводит к оптимальному решению. Например, для $N = 10$ «жадный» алгоритм даёт решение $6 + 1 + 1 + 1 + 1$ — всего 5 бидонов, в то время как можно обойтись двумя $(5 + 5)$.

Как и в любом решении, использующем динамическое программирование, *главная задача — составить рекуррентную формулу*. Сначала определим оптимальное число бидонов K_N, а потом подумаем, как определить, какие именно бидоны нужно использовать.

Представим себе, что мы выбираем бидоны постепенно. Тогда последний выбранный бидон может иметь, например, объём 1 л, в этом случае $K_N = 1 + K_{N-1}$. Если последний бидон имеет объём 5 л, то $K_N = 1 + K_{N-5}$, а если 6 л, то $K_N = 1 + K_{N-6}$. Так как нам нужно выбрать минимальное значение, то

$$K_N = 1 + \min(K_{N-1}, K_{N-2}, K_{N-6}).$$

Вариант, выбранный при поиске минимума, определяет последний добавленный бидон, его объём нужно сохранить в отдельном массиве P. Этот массив будет использован для определения количества выбранных бидонов каждого типа. В качестве начального значения берём $K_0 = 0$.

Полученная формула применима при $N \geq 6$. Для меньших N используются только те данные, которые есть в таблице (рис. 6.39). Например:

$$K_3 = 1 + K_2 = 3,\ K_5 = 1 + \min(K_4, K_0) = 1.$$

На рисунке 6.39 показаны массивы для $N = 10$.

N	0	1	2	3	4	5	6	7	8	9	10
K	0	1	2	3	4	1	1	2	3	4	2
P	(0)	1	1	1	1	(5)	6	1	1	1	(5)

Рис. 6.39

Как по массиву P определить оптимальный состав бидонов? Пусть, например, $N = 10$. Из массива `P` находим, что последний

добавленный бидон имеет объём 5 л. Остаётся 10 – 5 = 5 л, в элементе $P[5]$ тоже записано значение 5, поэтому второй бидон тоже имеет объём 5 л. Остаток 0 л означает, что мы полностью определили набор бидонов.

Можно заметить, что такая процедура очень похожа на алгоритм Дейкстры, и это не случайно. В алгоритмах Дейкстры и Флойда–Уоршелла, по сути, используется метод динамического программирования.

Задача 3 (задача о куче). Из камней весом p_i ($i = 1, ..., N$) требуется набрать кучу весом ровно W или, если это невозможно, максимально близкую к W (но меньшую, чем W).

Эта задача относится к трудным задачам целочисленной оптимизации, которые решаются только полным перебором вариантов. Каждый камень может входить в кучу (обозначим это состояние как 1) или не входить (0). Поэтому нужно выбрать цепочку, состоящую из N битов. При этом количество вариантов равно 2^N, и при больших N полный перебор практически невыполним.

Динамическое программирование позволяет найти решение задачи значительно быстрее. Идея состоит в том, чтобы сохранять в массиве решения всех более простых задач этого типа (при меньшем количестве камней и меньшем весе W).

Построим матрицу T, где элемент $T[i,w]$ — это оптимальный вес, полученный при попытке собрать кучу весом w из i первых по счёту камней (w изменяется от 0 до W). Очевидно, что первый столбец заполнен нулями (при заданном нулевом весе никаких камней не берём).

Рассмотрим первую строку (есть только один камень). В начале этой строки будут стоять нули, а дальше, начиная со столбца p_1, — значения p_1 (взяли единственный камень). Это простые варианты задачи, решения для которых легко подсчитать вручную. Рассмотрим пример, когда требуется набрать вес 8 из камней весом 2, 4, 5 и 7 единиц (рис. 6.40).

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0								
5	0								
7	0								

Рис. 6.40

Теперь предположим, что строки с 1-й по $(i-1)$-ю уже заполнены. Перейдем к i-й строке, т. е. добавим в набор i-й камень. Он может быть взят или не взят в кучу. Если мы не добавляем его в кучу, то $T[i,w] = T[i-1,w]$, т. е. решение не меняется от добавления в набор нового камня. Если камень с весом добавлен в кучу, то остаётся «добрать» остаток оптимальным образом (используя только предыдущие камни), т. е. $T[i,w] = T[i-1,w-p_i] + p_i$.

Как же решить, брать или не брать камень? Надо проверить, в каком случае полученное решение будет больше (ближе к w). Таким образом, получается рекуррентная формула для заполнения таблицы:

$$T[i,w] = \begin{cases} T[i-1,w], \text{ при } w < p_i, \\ \max(T[i-1,w], T[i-1,w-p_i] + p_i), \text{ при } w \ge p_i. \end{cases}$$

Используя эту формулу, заполняем таблицу по строкам, сверху вниз; в каждой строке — слева направо (рис. 6.41).

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0	0	2	2	4	4	6	6	6
5	0	0	2	2	4	5	6	7	7
7	0	0	2	2	4	5	6	7	7

Рис. 6.41

Видим, что сумму 8 набрать невозможно, ближайшее значение — 7 (правый нижний угол таблицы).

Эта таблица содержит все необходимые данные для определения выбранной группы камней. Действительно, если камень с весом p_i не включён в набор, то $T[i,w] = T[i-1,w]$, т. е. число в таблице не меняется при переходе на строку вверх. Начинаем с правого нижнего угла таблицы, идём вверх, пока значения в столбце равны 7. Последнее такое значение — для камня с весом 5, поэтому он и выбран. Вычитая его вес из суммы, получаем $7 - 5 = 2$, переходим во второй столбец на одну строку вверх, и снова идём вверх по столбцу, пока значение не меняется (равно 2). Так как мы успешно дошли до самого верха таблицы, взят первый камень с весом 2.

Заметим, что в рассмотренном случае есть и ещё одно решение — взять один камень с весом 7 (подумайте, как в подобных случаях находить все решения задачи).

Как мы уже отмечали, количество вариантов в задаче для N камней равно 2^N, т. е. алгоритм полного перебора имеет асимптотическую сложность $O(2^N)$. В данном алгоритме количество операций равно числу элементов таблицы, т. е. сложность нашего алгоритма — $O(N \cdot w)$. Однако нельзя сказать, что он имеет линейную сложность, так как есть ещё сильная зависимость от заданного веса w. Такие алгоритмы называют *псевдополиномиальными*. В них ускорение вычислений достигается за счёт использования дополнительной памяти для хранения промежуточных результатов.

Количество решений

Задача 4. У исполнителя Утроитель две команды, которым присвоены номера:

1) `прибавь 1`
2) `умножь на 3`

Первая из них увеличивает число на экране на 1, вторая — утраивает его. Программа для Утроителя — это последовательность команд. Сколько есть программ, которые число 1 преобразуют в число $N = 20$?

Заметим, что при выполнении любой из команд число увеличивается (не может уменьшаться). Начнем с простых случаев, с которых будем начинать вычисления. Понятно, что для $N = 1$ существует только одна программа — пустая, не содержащая ни одной команды. Для $N = 2$ есть тоже только одна программа, состоящая из команды сложения. Если через K_N обозначить количество разных программ для получения числа N из 1, то $K_1 = K_2 = 1$.

Теперь рассмотрим общий случай, чтобы построить рекуррентную формулу, связывающую K_N с предыдущими элементами последовательности $K_1, K_2, \ldots, K_N$, т. е. с решениями таких же задач для меньших N.

Если число N не делится на 3, то последней командой для его получения может быть только операция сложения, поэтому $K_N = K_{N-1}$. Если N делится на 3, то последней командой может быть как сложение, так и умножение. Поэтому нужно сложить K_{N-1} (количество программ с последней командой сложения)

и $K_{N/3}$ (количество программ с последней командой умножения). В итоге получаем:

$$K_N = \begin{cases} K_{N-1}, \text{если } N \text{ не делится на 3,} \\ K_{N-1} + K_{N/3}, \text{если } N \text{ делится на 3.} \end{cases}$$

Остаётся заполнить таблицу для всех значений от 1 до заданного $N = 20$. Для небольших значений эту задачу легко решить вручную:

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
K_N	1	1	2	2	2	3	3	3	5	5	5	7	7	7	9	9	9	12	12	12

Заметим, что количество вариантов меняется только в тех столбцах, где N делится на 3, поэтому из всей таблицы можно оставить только эти столбцы (и добавив следующее значение, кратное трём):

N	1	3	6	9	12	15	18	21
K_N	1	2	3	5	7	9	12	15

Заданное число 20 попадает в последний интервал (от 18 до 21), поэтому ответ в данной задаче — 12.

При составлении программы с полной таблицей нужно выделить в памяти целочисленный массив K, индексы которого изменяются от 1 до N, и заполнить его по приведённым выше формулам:

```
K[1]:=1;
for i:=2 to N do begin
  K[i]:=K[i-1];
  if i mod 3 = 0 then
    K[i]:=K[i]+K[i div 3];
end;
```

Ответом будет значение $K[N]$.

Задача 5 (размен монет). Сколькими различными способами можно выдать сдачу размером W рублей, если есть монеты достоинством p_i ($i = 1, \ldots, N$)? Для того чтобы сдачу всегда можно было выдать, будем предполагать, что в наборе есть монета достоинством 1 рубль ($p_1 = 1$).

Это задача, так же, как и задача о куче, решается только полным перебором вариантов, число которых при больших N очень велико. Будем использовать динамическое программирование, сохраняя в массиве решения всех задач меньшей размерности (для меньших значений N и W).

В матрице T значение $T[i,w]$ будет обозначать количество вариантов сдачи размером w рублей (w изменяется от 0 до W) при использовании первых i монет из набора. Очевидно, что при нулевой сдаче есть только один вариант (не дать ни одной монеты), так же и при наличии только одного типа монет (напомним, что $p_1 = 1$) есть тоже только один вариант. Поэтому нулевой столбец и первую строку таблицы можно заполнить сразу единицами. Для примера мы будем рассматривать задачу для $W = 10$ и набора монет достоинством 1, 2, 5 и 10 рублей (рис. 6.42).

	0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1
2	1										
5	1										
10	1										

Рис. 6.42

Таким образом, мы определили простые базовые случаи, от которых «отталкивается» рекуррентная формула.

Теперь рассмотрим общий случай. Заполнять таблицу будем по строкам, слева направо. Для вычисления $T[i,w]$ предположим, что мы добавляем в набор монету достоинством p_i. Если сумма w меньше, чем p_i, то количество вариантов не увеличивается, и $T[i,w] = T[i-1,w]$. Если сумма больше p_i, то к этому значению нужно добавить количество вариантов с «участием» новой монеты. Если монета достоинством p_i использована, то нужно учесть все варианты «разложения» остатка $w - p_i$ на все доступные монеты, т. е. $T[i,w] = T[i-1,w] + T[i,w - p_i]$. В итоге получается рекуррентная формула

$$T[i,w] = \begin{cases} T[i-1,w], \text{ при } w < p_i, \\ T[i-1,w] + T[i-1,w-p_i], \text{ при } w \ge p_i, \end{cases}$$

которая используется для заполнения таблицы (рис. 6.43).

	0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	2	2	3	3	4	4	5	5	6
5	1	1	2	2	3	4	5	6	7	8	10
10	1	1	2	2	3	4	5	6	7	8	(11)

Рис. 6.43

Ответ к задаче находится в правом нижнем углу таблицы.

Вы могли заметить, что решение этой задачи очень похоже на решение задачи о куче камней. Это не случайно, две эти задачи относятся к классу сложных задач, для решения которых известны только переборные алгоритмы. Использование методов динамического программирования позволяет ускорить решение за счёт хранения промежуточных результатов, однако требует дополнительного расхода памяти.

Вопросы и задания

1. Что такое динамическое программирование?
2. Какой смысл имеет выражение «динамическое программирование» в теории многошаговой оптимизации?
3. Какие шаги нужно выполнить, чтобы применить динамическое программирование к решению какой-либо задачи?
4. За счёт чего удаётся ускорить решение сложных задач методом динамического программирования?
5. Какие ограничения есть у метода динамического программирования?

Подготовьте сообщение

а) «Задача о рюкзаке»

б) «Задачи на подпоследовательности»

в) «Задачи на поиск оптимального маршрута»

Задачи

1. Напишите программу, которая определяет оптимальный набор бидонов в задаче 2 из параграфа. С клавиатуры или из файла вводится объём цистерны, количество типов бидонов и их размеры.
2. Напишите программу, которая решает задачу 3 о куче камней заданного веса, рассмотренную в тексте параграфа.

*3. **Задача о ранце.** Есть N предметов, для каждого из которых известен вес p_i ($i = 1,\ldots, N$) и стоимость c_i ($i = 1,\ldots, N$). В ранец можно взять

предметы общим весом не более W. Напишите программу, которая определяет самый дорогой набор предметов, который можно унести в ранце.

4. У исполнителя Калькулятор две команды, которым присвоены номера:

   1) `прибавь 1`
   3) `умножь на 4`

   Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число 1 в число N, введённое с клавиатуры. Используйте сокращённую таблицу.

5. У исполнителя Калькулятор три команды, которым присвоены номера:

   1) `прибавь 1`
   2) `умножь на 3`
   3) `умножь на 4`

   Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число 1 в число N, введённое с клавиатуры.

6. У исполнителя Калькулятор две команды, которым присвоены номера:

   1) `прибавь 1`
   2) `увеличь каждый разряд числа на 1`

   Сколько есть программ, которые число 24 преобразуют в число 46?

7. У исполнителя Калькулятор две команды, которым присвоены номера:

   1) `прибавь 1`
   2) `увеличь каждый разряд числа на 1`

   Сколько существует программ, которые число 26 преобразуют в число 49?

*8. Прямоугольный остров разделён на квадраты так, что его размеры — $N \times M$ квадратов. В каждом квадрате зарыто некоторое число золотых монет, эти данные хранятся в матрице (двумерном массиве) Z, где $Z[i, j]$ — число монет в квадрате с координатами (i, j). Пират хочет пройти из юго-западного угла острова в северо-восточный, причём он может двигаться только на север или на восток. Как пирату собрать наибольшее количество монет? Напишите программу, которая находит оптимальный путь пирата и число монет, которое ему удастся собрать.

Практические работы к главе 6

Работа № 41 «Решето Эратосфена»
Работа № 42 «Длинные числа»

Работа № 43 «Ввод и вывод структур»
Работа № 44 «Чтение структур из файла»
Работа № 45 «Сортировка структур с помощью указателей»
Работа № 46 «Динамические массивы»
Работа № 47 «Расширяющиеся динамические массивы»
Работа № 48 «Алфавитно-частотный словарь»
Работа № 49 «Модули»
Работа № 50 «Вычисление арифметических выражений»
Работа № 51 «Проверка скобочных выражений»
Работа № 52 «Заливка области»
Работа № 53 «Вычисление арифметических выражений»
Работа № 54 «Хранение двоичного дерева в массиве»
Работа № 55 «Алгоритм Прима–Крускала»
Работа № 56 «Алгоритм Дейкстры»
Работа № 57 «Алгоритм Флойда–Уоршелла»
Работа № 58 «Числа Фибоначчи»
Работа № 59 «Задача о куче»
Работа № 60 «Количество программ»
Работа № 61 «Размен монет»

ЭОР к главе 6 на сайте ФЦИОР (http://fcior.edu.ru)

www

- Решето Эратосфена
- Основные структуры данных
- Сущность модульного программирования. Программный модуль
- Работа с указателями и структурами (на примере языка Pascal)
- Линейные структуры данных. Список, стек, очередь
- Организация и работа со стеком
- Организация и работа с очередью
- Основы теории графов. Способы представления графов. Обход графа
- Задача о кратчайших путях. Алгоритм Флойда, Дейкстры
- Задачи оптимизации. Динамическое программирование

Самое важное в главе 6

- Структура — это сложный тип данных, который позволяет объединить данные разных типов. Элементы структуры на-

зывают полями. При обращении к полям структуры используется точечная нотация:

<имя структуры>.<имя поля>.

- Указатель — это переменная, в которой можно хранить адрес другой переменной заданного типа. В указателе можно запомнить адрес новой переменной, место для которой выделено в памяти во время работы программы.
- Динамические массивы — это массивы, память для которых выделяется во время работы программы. Динамический массив в программе на языке Паскаль — это указатель, в который записывается адрес выделенного блока памяти. При записи такой переменной в файл сохранится только значение указателя, а значения элементов массива будут потеряны.
- Список — это упорядоченный набор элементов одного типа, для которых введены операции вставки (включения) и удаления (исключения).
- Стек — это линейный список, в котором добавление и удаление элементов разрешаются только с одного конца. Системный стек применяется для хранения адресов возврата из подпрограмм и размещения локальных переменных.
- Дерево — это структура данных, которая моделирует иерархию — многоуровневую структуру. Дерево — рекурсивная структура, поэтому для его обработки удобно использовать рекурсивные алгоритмы. Деревья используются в задачах поиска, сортировки, вычисления арифметических выражений.
- Граф — это набор узлов и связывающих их рёбер. Информация о графе чаще всего хранится в виде матрицы смежности или весовой матрицы. Наиболее известные задачи, которые решаются с помощью теории графов, — поиск оптимальных маршрутов.
- Динамическое программирование — это метод, позволяющий ускорить решение задачи за счёт хранения решений аналогичных задач меньшей размерности. Для его использования нужно вывести рекуррентную формулу, связывающее решение задачи с решением задач меньшей размерности, и определить простые базовые случаи (условие окончания рекурсии).

Глава 7
Объектно-ориентированное программирование

§ 46
Что такое ООП?

Как вы знаете, работа первых компьютеров сводилась к вычислениям по заданным формулам различной сложности. Число переменных и массивов в программе было невелико, так что программист мог легко удерживать в памяти все взаимосвязи между ними и детали алгоритма.

С каждым годом производительность компьютеров росла, и человек «поручал» им всё более и более трудоёмкие задачи. Компьютеры следующих поколений стали использоваться для создания сложных информационных систем (например, банковских) и моделирования процессов, происходящих в реальном мире. Новые задачи требовали более сложных алгоритмов, объём программ вырос до сотен тысяч и даже миллионов строк, число переменных и массивов измерялось в тысячах.

Программисты столкнулись с проблемой сложности, которая превысила возможности человеческого разума. Один человек уже не способен написать надёжно работающую серьезную программу, так как не может «охватить взглядом» все её детали. Поэтому в разработке большинства современных программ принимает участие множество специалистов. При этом возникает новая проблема: нужно разделить работу между ними так, чтобы каждый мог работать независимо от других, а потом готовую программу можно было бы собрать вместе из готовых блоков, как из кубиков.

Как отмечал известный нидерландский программист Эдсгер Дейкстра, человечество ещё в древности придумало способ управления сложными системами: «разделяй и властвуй». Это означает, что исходную систему нужно разбить на подсистемы (выполнить **декомпозицию**) так, чтобы работу каждой из них можно было рассматривать и совершенствовать независимо от других.

Для этого в классическом (процедурном) программировании используют метод проектирования «сверху вниз»: сложная задача разбивается на части (подзадачи и соответствующие им *алгоритмы*), которые затем снова разбиваются на более мелкие подзадачи и т. д. (рис. 7.1). Однако при этом задачу «реального мира» приходится переформулировывать, представляя все данные в виде переменных, массивов, списков и других структур данных. При моделировании больших систем объём этих данных увеличивается, они становятся плохо управляемыми, и это приводит к большому числу ошибок. Так как любой алгоритм может обратиться к любым глобальным (общедоступным) данным, повышается риск случайного недопустимого изменения каких-то значений.

Рис. 7.1

В конце 60-х годов XX века появилась новая идея — применить в разработке программ тот подход, который использует человек в повседневной жизни. Люди воспринимают мир как множество **объектов** — предметов, животных, людей — это отмечал ещё в XVII веке французский математик и философ Рене Декарт. Все объекты имеют внутреннее устройство и состояние, свойства (внешние характеристики) и поведение. Чтобы справиться со сложностью окружающего мира, люди часто не вникают в детали внутреннего устройства и игнорируют многие свойства объектов, ограничиваясь лишь теми, которые необходимы для решения их практических задач. Такой приём называется абстракцией.

! **Абстракция** — это выделение существенных характеристик объекта, отличающих его от других объектов.

Для разных задач существенные свойства одного и того же объекта могут быть совершенно разными. Например, услышав слово «кошка», многие подумают о пушистом усатом животном, которое мурлыкает, когда его гладят. В то же время ветеринарный врач представляет скелет, ткани и внутренние органы кошки, которую ему нужно лечить. В каждом из этих случаев применение абстракции даёт свою модель одного и того же объекта, поскольку различны цели моделирования.

Как применить принцип абстракции в программировании? Поскольку формулировка задач, решаемых на компьютерах, всё более приближается к формулировкам реальных жизненных задач, возникла такая идея: представить программу в виде множества объектов (моделей), каждый из которых обладает своими свойствами и поведением, но его внутреннее устройство скрыто от других объектов. Тогда решение задачи сводится к моделированию взаимодействия этих объектов. Построенная таким образом модель задачи называется *объектной*. Здесь тоже идёт проектирование «сверху вниз», только не по алгоритмам (как в процедурном программировании), а по объектам. Если нарисовать схему такой декомпозиции, она будет представлять собой граф, так как каждый объект может обмениваться данными со всеми другими (рис. 7.2).

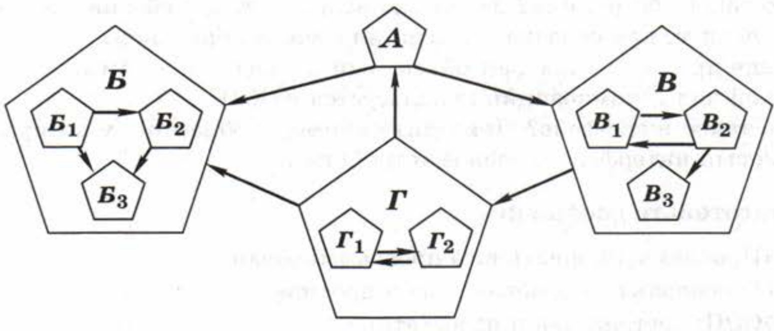


Рис. 7.2

Здесь *А*, *Б*, *В* и *Г* — объекты «верхнего уровня»; $Б_1$, $Б_2$ и $Б_3$ — подобъекты объекта *Б* и т. д.

Для решения задачи «на верхнем уровне» достаточно определить, *что* делает тот или иной объект, не заботясь о том, *как* именно он это делает. Таким образом, для преодоления сложности мы используем *абстракцию*, т. е. сознательно отбрасываем второстепенные детали.

Если построена **объектная модель задачи** (выделены объекты и определены правила обмена данными между ними), можно поручить разработку каждого из объектов отдельному программисту (или группе), которые должны написать соответствующую часть программы, т. е. определить, *как именно* объект выполняет свои функции. При этом конкретному разработчику не обязательно держать в голове полную информацию обо всех объектах, нужно лишь строго соблюдать соглашения о способе обмена данными (*интерфейсе*) «своего» объекта с другими.

Программирование, основанное на моделировании задачи реального мира как множества взаимодействующих объектов, принято называть **объектно-ориентированным программированием (ООП)**. Более строгое определение мы дадим немного позже.

Вопросы и задания

1. Почему со временем неизбежно изменяются методы программирования?
2. Что такое декомпозиция, зачем она применяется?
3. Что такое процедурное программирование? Какой вид декомпозиции в нём используется?
4. Какие проблемы в программировании привели к появлению ООП?
5. Как выполняется декомпозиция алгоритмов в процедурных языках программирования?
6. Что такое абстракция? Зачем она используется в обычной жизни?
7. Объясните, как связана абстракция с моделированием.
8. Какие преимущества даёт объектный подход в программировании?
9. Какой вид декомпозиции используется в ООП?
10. Что такое интерфейс? Приведите примеры объектов, у которых одинаковый интерфейс и разное устройство.

Подготовьте сообщение

а) «Проблемы процедурного программирования»

б) «Глобальные переменные: за и против»

в) «ООП: достоинства и недостатки»

§ 47 Объекты и классы

Как мы увидели в предыдущем параграфе, для того чтобы построить объектную модель, нужно:

- выделить взаимодействующие **объекты**, с помощью которых можно достаточно полно описать поведение моделируемой системы;
- определить **свойства** объектов, существенные в данной задаче;
- описать **поведение** (возможные действия) объектов, т. е. команды, которые объекты могут выполнить.

Этап разработки модели, на котором решаются перечисленные выше задачи, называется **объектно-ориентированным анализом (ООА)**. Он выполняется до того, как программисты напишут самую первую строчку кода, и во многом определяет качество и надёжность будущей программы.

Рассмотрим объектно-ориентированный анализ на примере простой задачи. Пусть нам необходимо изучить движение автомобилей на шоссе, например, для того, чтобы определить, достаточна ли его пропускная способность. Как построить объектную модель этой задачи? Прежде всего, нужно разобраться, что такое объект.

Объектом можно назвать то, что имеет чёткие границы и обладает **состоянием** и **поведением**.

Состояние объекта определяет его возможное поведение. Например, лежачий человек не может прыгнуть, а незаряженное ружьё не выстрелит.

В нашей задаче объекты — это дорога и двигающиеся по ней машины. Машин может быть несколько, причём все они, с точки зрения нашей задачи, имеют общие свойства. Поэтому нет смысла подробно описывать каждую машину по отдельности: достаточно один раз определить их общие черты, а потом просто сказать, что все машины ими обладают. В ООП для этой цели вводится специальный термин — «класс».

Класс — это множество объектов, имеющих общую структуру и общее поведение.

Например, в рассматриваемой задаче можно ввести два класса — *Дорога* и *Машина*. По условию, дорога одна, а машин может быть много.

Будем рассматривать прямой отрезок дороги, в этом случае объект «дорога» имеет два свойства, важных для нашей задачи: длину и ширину — число полос движения (рис. 7.3). Эти свойства определяют *состояние* дороги. «*Поведение*» дороги может заключаться в том, что число полос меняется, например, из-за ремонта покрытия, но в нашей простейшей модели объект «дорога» не будет изменяться.

Рис. 7.3

Дорога
длина ширина

Рис. 7.4

Схематично класс *Дорога* можно изобразить в виде прямоугольника с тремя секциями: в верхней записывают название класса, во второй части — свойства, а в третьей — возможные действия, которые называют **методами**. В нашей модели дороги два свойства и ни одного метода (рис. 7.4).

Теперь рассмотрим объекты класса *Машина*. Их важнейшие свойства — координаты и скорость движения. Для упрощения будем считать, что:

- все машины одинаковы;
- каждая машина движется по дороге слева направо с постоянной скоростью (скорости разных машин могут быть различными);
- по каждой полосе движения едет только одна машина, так что можно не учитывать обгон и переход на другую полосу;
- если машина выходит за правую границу дороги, вместо неё слева на той же полосе появляется новая машина.

Не все эти допущения выглядят естественно, но такая простая модель позволит понять основные принципы метода.

За координаты машины можно принять расстояние X от левого края рассматриваемого участка шоссе и номер полосы Y (натуральное число — рис. 7.5). Скорость автомобиля V в нашей модели — неотрицательная величина.

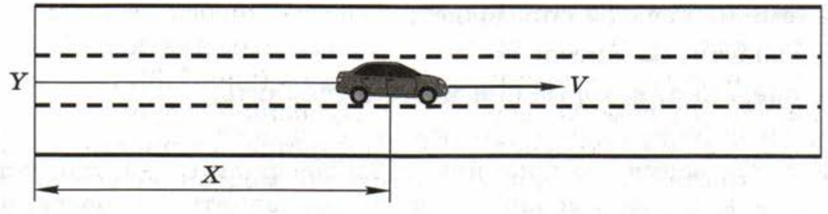


Рис. 7.5

Теперь рассмотрим поведение машины. В данной модели она может выполнять всего одну команду — ехать в заданном направлении (назовём её «двигаться»). Говорят, что объекты класса *Машина* имеют метод «двигаться» (рис. 7.6).

Машина
X(координата) Y(полоса) V(скорость)
двигаться

Рис. 7.6

Метод — это процедура или функция, принадлежащая классу объектов.

Другими словами, метод — это некоторое действие, которое могут выполнять все объекты класса.

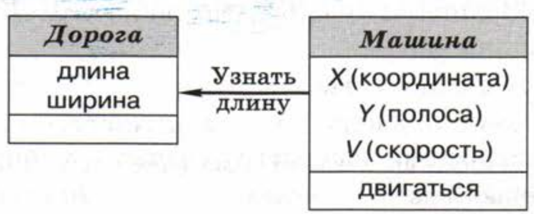


Рис. 7.7

Пока мы построили только модели отдельных объектов (точнее, классов). Чтобы моделировать всю систему, нужно разобраться, как эти объекты взаимодействуют. Объект-машина должен уметь «определить», в каком месте дороги он находится. Для этого машина должна обращаться к объекту «дорога», запрашивая длину дороги (см. стрелку на рис. 7.7).

Схема на рис. 7.7 определяет:

- свойства объектов;
- операции, которые они могут выполнять;
- связи (обмен данными) между объектами.

В то же время мы пока ничего не говорили о том, как устроены объекты и как именно они будут выполнять эти операции, и это не случайно. Согласно принципам ООП, ни один объект не должен зависеть от внутреннего устройства и алгоритмов работы других объектов. Поэтому, построив такую схему, можно поручить разработку двух классов объектов двум программистам, каждый из которых может решать свою задачу независимо от других. Важно только, чтобы все они чётко соблюдали **интерфейс** — правила, описывающие взаимодействие «своих» объектов с остальными.

Вопросы и задания

1. Какие этапы входят в объектно-ориентированный анализ?
2. Что такое объект?
3. Что такое класс? Чем различаются понятия «класс» и «объект»?
4. Что такое метод?
5. Как изображаются классы на схеме?
6. Почему при объектно-ориентированном анализе не уточняют, как именно объекты будут устроены и как они будут решать свои задачи?

Задачи

1. Подумайте, какими свойствами и методами могли бы обладать объекты следующих классов: *Ученик*, *Учитель*, *Школа*, *Экзамен*, *Турнир*, *Урок*, *Страна*, *Браузер*. Придумайте свои классы объектов и выполните их анализ.
2. Добавьте в рассмотренную в параграфе модель светофоры (на дороге их может быть много). Подумайте, какие свойства и методы должны быть у объектов класса *Светофор*. Как могут быть связаны классы *Дорога*, *Светофор* и *Машина* (сравните разные варианты)?
3. Придумайте свою задачу и выполните её объектно-ориентированный анализ. Примеры: моделирование работы магазина, банка, библиотеки и т. п.

§ 48
Создание объектов в программе

Класс Дорога

Объектно-ориентированная программа начинается с описания классов объектов. Класс в программе — это новый тип данных. Как и структура (см. § 39), класс — это сложный тип данных, который может объединять переменные различного типа в единый блок. Однако, в отличие от структуры, класс содержит не только данные, но и методы работы с ними (процедуры и функции).

В нашей программе самый простой класс — это *Дорога*. Объекты этого класса имеют два свойства: длину (англ. *length*), которая может быть вещественным числом, и ширину (англ. *width*) — количество полос, целое число. Для хранения значений свойств используются переменные, принадлежащие объекту, которые называются полями.

Поле — это переменная, принадлежащая объекту.

Значения полей описывают *состояние* объекта, а методы — его *поведение*.

Описание класса *Дорога* в программе на объектной версии Паскаля (здесь имеется в виду *FreePascal* или *Delphi*) выглядит так:

```
type TRoad = class
       Length: real;
       Width: integer;
     end;
```

Эти строки вводят новый тип данных — класс TRoad[1], т. е. сообщают компилятору, что в программе, возможно, будут использоваться объекты этого типа. При этом в памяти не создаётся ни одного объекта. Это описание похоже на чертёж, по которому в нужный момент можно построить сколько угодно таких объектов.

1 Буква «Т» в начале названия класса — это сокращение от слова *type*.

Если мы хотим работать с объектом класса TRoad, в программе нужно объявить соответствующую переменную:

```
var road: TRoad;
```

Однако и это ещё не объект, а ссылка (указатель), т. е. переменная, в которой можно сохранить адрес любого объекта класса TRoad. Чтобы создать сам объект в памяти, нужно вызвать специальный *метод* Create, который называется **конструктором.** Адрес нового объекта записываем в переменную road:

```
road:=TRoad.Create;
```

Созданный объект относится к классу TRoad, поэтому его называют **экземпляром класса** TRoad.

При описании класса мы ничего не говорили о методе Create. Он добавляется ко всем классам по умолчанию, при его вызове все переменные объекта заполняются нулями[1].

Конструктор — это метод класса, который вызывается для создания объекта этого класса.

Свойства дороги можно изменить с помощью точечной нотации, с которой вы познакомились, работая со структурами:

```
road.Length:=60;
road.Width:=3;
```

Полная программа, которая создаёт объект «дорога» (и больше ничего не делает), выглядит так:

```
{$mode objfpc}
type TRoad = class
       Length: real;
       Width: integer;
     end;
var road: TRoad;
begin
  road:=TRoad.Create;
  road.Length:=60;
  road.Width:=3
end.
```

[1] Хотя стандарта на этот счёт нет, так сделано во всех объектных реализациях Паскаля.

Строка {$mode objfpc} по форме похожа на комментарий, потому что заключена в фигурные скобки. Однако для компилятора это команда перейти в режим работы с объектами (англ. *mode* — режим; *object* — объект; FPC — Free Pascal Compiler — свободно распространяемый компилятор Паскаля).

Начальные значения полей можно задавать прямо при создании объекта. Для этого нужно добавить в описание класса новый конструктор. Конструктору будет передаваться два параметра — начальные значения длины и ширины дороги:

```
type TRoad = class
       Length: real;
       Width: integer;
       constructor Create(length0: real;
                          width0: integer)
     end;
```

Реализация (программа) конструктора может выглядеть так:

```
constructor TRoad.Create(length0: real;
                         width0: integer);
begin
  if length0>0 then
       Length:=length0
  else Length:=1;
  if width0>0 then
       Width:=width0
  else Width:=1;
end;
```

Здесь проверяется правильность переданных параметров, чтобы по ошибке длина и ширина дороги не оказались нулевыми или отрицательными[1]. Теперь создавать объект будет проще:

```
road:=TRoad.Create(60, 3);
```

Длина этой дороги — 60 единиц, она содержит 3 полосы.

Таким образом, класс выполняет роль «фабрики», которая «выпускает» (создаёт) объекты «по чертежу» (описанию класса) при вызове конструктора.

1 Конечно, в реальной программе при передаче неправильных данных нужно выдавать сообщение об ошибке.

Класс Машина

Теперь можно описать класс *Машина* (в программе назовём его TCar). Объекты класса TCar имеют три свойства и один метод — процедуру move. Координата X и скорость V — это вещественные значения, а номер полосы P — целое.

```
type TCar = class
       X, V: real;
       P: integer;
       road: TRoad;
       procedure move;
       constructor Create(road0: TRoad;
                          p0: integer; v0: real);
     end;
```

Так как объекты-машины должны обращаться к объекту «дорога», в область данных включено дополнительное поле *road*. Конечно, это не значит, что в состав машины входит дорога. Напомним, что это только ссылка, и сразу после создания объекта-машины нужно записать в неё адрес заранее созданного объекта «дорога». Эту привязку удобно сделать прямо в конструкторе, при создании объекта. Заодно мы определяем полосу движения и скорость, а начальная координата X автоматически устанавливается равной нулю:

```
constructor TCar.Create(road0: TRoad;
                        p0: integer; v0: real);
begin
  road:=road0; P:=p0; V:=v0;
end;
```

Теперь займёмся реализацией (программированием) метода move (англ. *move* — двигаться). В этом методе нужно вычислить новую координату X машины и, если она находится за пределами дороги, установить её в ноль (машина появляется слева на той же полосе). Изменение координаты при равномерном движении описывается формулой

$$X = X_0 + V \cdot \Delta t,$$

где X_0 и X — начальная и конечная координаты, V — скорость, а Δt — время движения. Вспомним, что любое моделирование физических процессов на компьютере происходит в дискретном времени, с некоторым интервалом дискретизации. Для простоты мож-

но измерять время в этих интервалах, а за скорость V принять расстояние, проходимое машиной за один интервал. Тогда метод **move**, описывающий изменение положения машины за один интервал ($\Delta t = 1$), может выглядеть так:

```
procedure TCar.move;
begin
  X:=X+V;
  if X > road.Length then X:=0;
end;
```

Основная программа

В основной программе объявим массив объектов-машин:

```
const N=3;
var cars: array [1..N] of TCar;
```

Как вы помните, это ещё не объекты, а ссылки — переменные, в которые можно записать адреса объектов класса TCar. Теперь нужно создать сами объекты:

```
var i: integer;
...
for i:=1 to N do
  cars[i]:=TCar.Create(road, i, 2.0*i);
```

При вызове конструктора задаются три параметра: адрес объекта «дорога» (его нужно создать до выполнения этого цикла), номер полосы и скорость. В приведённом варианте машина на полосе с номером i движется со скоростью $2i$ единиц за один интервал моделирования.

Сам цикл моделирования получается очень простой: на каждом шаге вызывается метод move для каждой машины:

```
repeat
  for i:=1 to N do cars[i].move;
until keypressed;
```

Этот цикл закончится тогда, когда пользователь нажмёт любую клавишу и функция keypressed вернёт значение True.

Полностью основная программа выглядит так:

```
const N = 3;
var road: TRoad;
    cars: array [1..N] of TCar;
    i: integer;
```

```
begin
  road:=TRoad.Create(60, N);
  for i:=1 to N do
    cars[i]:=TCar.Create(road, i, 2.0*i);
  repeat
    for i:=1 to N do cars[i].move;
  until keypressed;
end.
```

Можно ли было написать такую же программу, не используя объекты? Конечно, да. И она получилась бы короче, чем наш объектный вариант (с учётом описания классов). В чём же преимущества ООП?

Мы уже отмечали, что ООП — это средство разработки больших программ, моделирующих работу сложных систем. В этом случае очень важно, что при использовании объектного подхода:

- основная программа, описывающая решение задачи в целом, получается простой и понятной; все команды напоминают действия в реальном мире («машина № 2, вперёд!»);
- разработку отдельных классов объектов можно поручить разным программистам, при этом каждый может работать независимо от других;
- если объекты классов *Дорога* и *Машина* понадобятся в других разработках, можно будет легко использовать уже готовые классы.

Вопросы и задания

1. Что такое поле в описании класса объекта?
2. Как объявляется класс объектов в программе?
3. Как объявляется переменная для работы с объектом некоторого класса? Что в ней хранится?
4. Как в памяти создаётся экземпляр класса (объект)?
5. Что такое конструктор?
6. Что такое точечная нотация? Как она используется при работе с объектами?
7. Как можно задать начальные значения для полей объекта?
8. Почему в методе `TCar.move` (пример, разобранный в параграфе) не объявлены переменные `X` и `V`?

9. Сравните преимущества и недостатки решения рассмотренной задачи «классическим» способом и с помощью ООП. Сделайте выводы.

Подготовьте сообщение

а) «Классы в языке Си»

б) «Классы в языке Javascript»

в) «Классы в языке Python»

Задачи

1. Добавьте в рассмотренную в параграфе программу операторы, позволяющие изобразить на экране перемещение машин (в текстовом или графическом режиме). Подумайте, какие методы можно добавить для этого в класс `TCar`.

*2. Добавьте в модель из параграфа светофор, который переключается автоматически по программе (например, 5 с горит красный свет, затем 1 с — жёлтый, потом 5 с — зелёный и т. д.). Измените классы так, чтобы машина запрашивала у объекта *Дорога* местоположение ближайшего светофора, а затем обращалась к светофору для того, чтобы узнать, какой сигнал горит. Машины должны останавливаться у светофора с запрещающим сигналом.

§ 49
Скрытие внутреннего устройства

Во время построения объектной модели задачи мы выделили отдельные объекты, которые для обмена данными друг с другом используют *интерфейс* — внешние свойства и методы. При этом все внутренние данные и детали внутреннего устройства объекта должны быть скрыты от «внешнего мира». Такой подход позволяет:

- обезопасить внутренние данные (поля) объекта от изменений (возможно, разрушительных) со стороны других объектов;
- проверять данные, поступающие от других объектов, на корректность, тем самым повышая надёжность программы;
- переделывать внутреннюю структуру и код объекта любым способом, не меняя его внешние характеристики (интерфейс); при этом никакой переделки других объектов не требуется.

Скрытие внутреннего устройства объектов называют **инкапсуляцией** («помещение в капсулу»).

Заметим, что в объектно-ориентированном программировании инкапсуляцией также называют объединение данных и методов работы с ними в одном объекте.

Разберём простой пример. Во многих системах программирования есть класс, описывающий свойства «пера», которое используется при рисовании линий в графическом режиме. Назовём этот класс TPen, в простейшем варианте он будет содержать только одно поле *Color*, которое определяет цвет. Будем хранить код цвета в виде символьной строки, в которой записан шестнадцатеричный код составляющих модели RGB. Например, 'FF00FF' — это фиолетовый цвет, потому что красная (R) и синяя (B) составляющие равны FF_{16} = 255, а зелёной составляющей нет вообще. Класс можно объявить так:

```
type TPen = class
       Color: string;
     end;
```

По умолчанию все члены класса (поля и методы) открытые, общедоступные (*англ.* public). Те элементы, которые нужно скрыть, в описании класса помещают в «частный» раздел (*англ.* private), например, так:

```
type TPen = class
     private
       FColor: string;
     end;
```

В этом примере поле FColor **закрытое**. Имена всех закрытых полей далее будем начинать с буквы «F» (от англ. *field* — поле). К закрытым полям нельзя обратиться извне (это могут делать только методы самого объекта), поэтому теперь невозможно не только изменить внутренние данные объекта, но и просто узнать их значения. Чтобы решить эту проблему, нужно добавить к классу ещё два метода: один из них будет возвращать текущее значение поля *FColor*, а второй — присваивать полю новое значение. Эти методы доступа назовем getColor (в переводе с англ. — получить Color) и setColor (в переводе с англ. — установить Color):

```
type TPen=class
     private
       FColor: string;
     public
       function getColor: string;
       procedure setColor(newColor: string);
     end;
```

Обратите внимание, что оба метода находятся в секции **public (общедоступные)**.

Что же улучшилось по сравнению с первым вариантом (когда поле было открытым)? Согласно принципам ООП, внутренние поля объекта должны быть доступны только с помощью методов. В этом случае внутреннее представление данных может как угодно отличаться от того, как другие объекты «видят» эти данные.

В простейшем случае метод getColor можно написать так:

```
function TPen.getColor: string;
begin
  Result:=FColor;
end;
```

В методе setColor мы можем обрабатывать ошибки, не разрешая присваивать полю недопустимые значения. Например, установим, что символьная строка с кодом цвета, передаваемая нашему объекту, должна состоять из шести символов. Если эти условия не выполняются, будем записывать в поле FColor код чёрного цвета '000000':

```
procedure TPen.setColor(newColor: string);
begin
  if Length(newColor)<>6 then
    FColor:='000000' { если ошибка, то чёрный цвет}
  else FColor:=newColor
end;
```

Теперь если pen — это объект класса TPen, то для установки и чтения его цвета нужно использовать показанные выше методы:

```
pen.setColor ('FFFF00'); {изменение цвета}
writeln( 'цвет пера: ', pen.getColor );
                                {получение цвета}
```

Итак, мы скрыли внутренние данные, но одновременно обращение к свойствам стало выглядеть довольно неуклюже: вместо pen.Color:='FFFF00' теперь нужно писать pen.setColor('FFFF00'). Чтобы упростить запись, во многие объектно-ориентированные языки программирования ввели понятие свойства (англ. *property*), которое внешне выглядит как переменная объекта, но на самом деле при записи и чтении свойства вызываются методы объекта.

Свойство — это способ доступа к внутреннему состоянию объекта, имитирующий обращение к его внутренней переменной.

Свойство color в нашем случае можно определить так:

```
type TPen = class
  private
    FColor: string;
    function getColor: string;
    procedure setColor(newColor: string);
  public
    property color: string read getColor
                           write setColor;
  end;
```

Здесь методы getColor и setColor перенесены в раздел **private**, т. е. закрыты от других объектов. Однако есть общедоступное свойство color строкового типа:

```
property color: string read getColor
                       write setColor;
```

При чтении этого свойства (англ. *read*) вызывается метод getColor, а при записи нового значения (англ. *write*) — метод setColor. В программе можно использовать это свойство так:

```
pen.color:='FFFF00';    {изменение цвета}
writeln( 'цвет пера: ', pen.Color );
                        {получение цвета}
```

Поскольку приведённая выше функция getColor просто возвращает значение поля *FColor* и не выполняет никаких дополнительных действий, можно было вообще удалить метод getColor и объявить свойство так:

```
property color: string read FColor write setColor;
```

В этом случае при чтении выполняется прямой доступ к полю.

Таким образом, с помощью свойства color другие объекты могут изменять и читать цвет объектов класса TPen. Для обмена данными с «внешним миром» важно лишь то, что свойство color — символьного типа, и оно содержит 6-символьный код цвета. При этом внутреннее устройство объектов TPen может быть любым, и его можно менять как угодно. Покажем это на примере.

Хранение цвета в виде символьной строки неэкономно и неудобно, так как большинство стандартных функций используют числовые коды цвета. Поэтому лучше хранить код цвета как целое число, и поле *FColor* сделать целого типа:

```
FColor: integer;
```

При этом необходимо поменять методы getColor и setColor, которые непосредственно работают с этим полем:

```
function TPen.getColor: string;
begin
  Result:=IntToHex(FColor,6);
end;
procedure TPen.setColor(newColor: string);
begin
  if Length(newColor)<>6 then
    FColor:=0    {если ошибка, то чёрный цвет}
  else begin
    FColor:=StrToInt('$'+newColor);
  end;
end;
```

Для перевода числового кода в символьную запись используется функция IntToHex, входящая в библиотеку FreePascal (модуль SysUtils). Её второй параметр — количество цифр, которое будет в шестнадцатеричном числе. Обратный перевод выполняет функция StrToInt. Для того чтобы указать, что число записано в шестнадцатеричной системе, перед ним добавляют символ $.

В этом примере мы принципиально изменили внутреннее устройство объекта: заменили строковое поле на целочисленное. Однако другие объекты даже не «догадаются» о такой замене, потому что сохранился интерфейс — свойство color по-прежнему имеет строковый тип. Таким образом, инкапсуляция позволяет как угодно изменять внутреннее устройство объектов, не затрагивая интерфейс. При этом все остальные объекты изменять не требуется.

Иногда не нужно разрешать другим объектам менять свойство, т. е. требуется сделать свойство «только для чтения» (англ. *read-only*). Пусть, например, мы строим программную модель автомобиля. Как правило, другие объекты не могут непосредственно менять его скорость, однако могут получить информацию о ней — «прочитать» значение скорости. При описании такого свойства слово write и название метода записи не указывают вообще:

```
type TCar = class
  private
    Fv: real;
    ...
  public
    property v: real read Fv;
    ...
  end;
```

Таким образом, доступ к внутренним данным объекта возможен, как правило, только с помощью методов. Применение свойств (**property**) очень удобно, потому что позволяет использовать ту же форму записи, что и при работе с общедоступной переменной объекта.

При использовании скрытия данных длина программы чаще всего увеличивается, однако мы получаем и важные преимущества. Код, связанный с объектом, разделён на две части: общедоступную часть (секция **public**) и закрытую (**private**). Объект взаимодействует с другими объектами только с помощью своих общедоступных свойств и методов (интерфейс) — рис. 7.8. Поэтому при сохранении интерфейса можно как угодно менять внутреннюю структуру данных и код методов, и это никак не будет влиять на другие объекты. Подчеркнём, что всё это становится действительно важно, когда разрабатывается большая программа и необходимо обеспечить её надёжность.

Рис. 7.8

Вопросы и задания

1. Что такое интерфейс объекта?
2. Что такое инкапсуляция? Каковы её цели?
3. Чем различаются секции **public** и **private** в описании классов? Как определить, в какую из них поместить свойство или метод?
4. Почему рекомендуют делать доступ к полям объекта только с помощью методов?
5. Что такое свойство? Зачем во многие языки программирования введено это понятие?
6. Можно ли с помощью свойства обращаться напрямую к полю объекта, не используя метод?

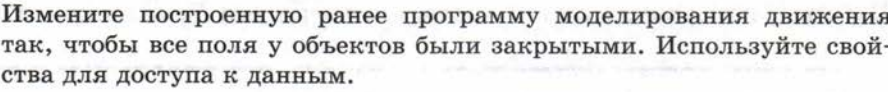
7. Почему методы доступа, которые использует свойство, делают закрытыми?
8. Зачем нужны свойства «только для чтения»? Приведите примеры.
9. Подумайте, в каких ситуациях может быть нужно свойство «только для записи» (которое нельзя прочитать). Как ввести такое свойство в описание класса? Приведите примеры.

Подготовьте сообщение

а) «Инкапсуляция в языке Си»
б) «Инкапсуляция в языке Javascript»
в) «Инкапсуляция в языке Python»

Задача

Измените построенную ранее программу моделирования движения так, чтобы все поля у объектов были закрытыми. Используйте свойства для доступа к данным.



§ 50 Иерархия классов

Классификации

Как в науке, так и в быту, важную роль играет **классификация** — разделение изучаемых объектов на группы (классы), объединённые общими признаками. Прежде всего это нужно для того, чтобы не запутаться в большом количестве данных и не описывать каждый объект заново.

Например, есть много видов фруктов[1] (яблоки, груши, бананы, апельсины и т. д.), но все они обладают некоторыми общими свойствами. Если перевести этот пример на язык ООП, класс *Яблоко* — это **подкласс** (производный класс, **класс-наследник**, потомок) класса *Фрукт*, а класс *Фрукт* — это **базовый класс** (суперкласс, класс-предок) для класса *Яблоко* (а также для классов *Груша*, *Слива*, *Апельсин* и др.).

Стрелка на схеме (рис. 7.9) обозначает наследование. Например, класс *Яблоко* — это наследник класса *Фрукт*.

[1] Фруктами называют сочные съедобные плоды деревьев и кустарников.

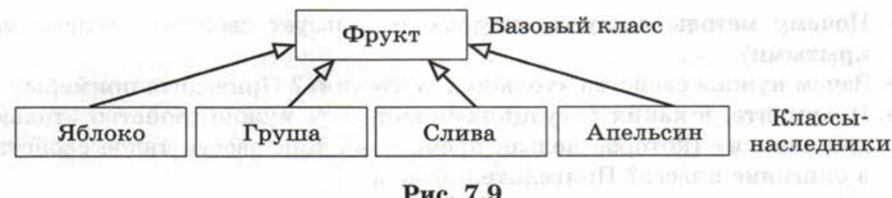


Рис. 7.9

Классический пример научной классификации — классификация животных или растений. Как вы знаете, она представляет собой *иерархию* (многоуровневую структуру). Например, горный клевер относится к роду *Клевер* семейства *Бобовые* класса *Двудольные* и т. д. Говоря на языке ООП, класс *Горный клевер* — это наследник класса *Клевер*, а тот, в свою очередь, — наследник класса *Бобовые*, который является наследником класса *Двудольные* и т. д.

Класс *Б* является наследником класса *А*, если можно сказать, что *Б* — это разновидность *А*.

Например, можно сказать, что яблоко — это фрукт, а горный клевер — одно из растений семейства *Двудольные*. В то же время мы не можем сказать, что «двигатель — это разновидность машины», поэтому класс *Двигатель* не является наследником класса *Машина*. Двигатель — это составная часть машины, поэтому объект класса *Машина* содержит в себе объект класса *Двигатель*. Отношения между двигателем и машиной — это отношение «часть — целое».

Иерархия логических элементов

Рассмотрим такую задачу: составить программу для моделирования управляющих схем, построенных на логических элементах (см. главу 3 в учебнике для 10 класса). Нам нужно «собрать» заданную схему и построить её таблицу истинности.

Как вы уже знаете, перед тем, как программировать, нужно выполнить объектно-ориентированный анализ. Все объекты, из которых состоит схема, — это логические элементы, однако они могут быть разными (НЕ, И, ИЛИ и другие). Попробуем выделить общие свойства и методы всех логических элементов.

Ограничимся только элементами, у которых один или два входа. Тогда иерархия классов может выглядеть, как показано на рис. 7.10.

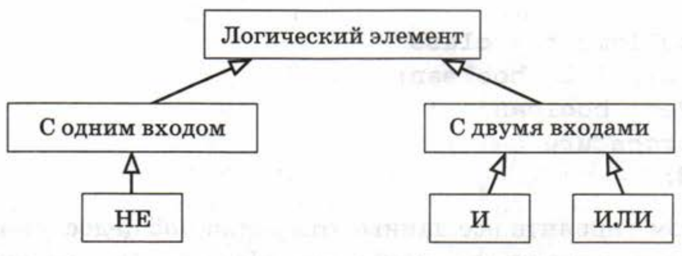


Рис. 7.10

Среди всех элементов с двумя входами мы показали только элементы «И» и «ИЛИ», остальные вы можете добавить самостоятельно.

Итак, для того чтобы не описывать несколько раз одно и то же, классы в программе должны быть построены в виде иерархии. Теперь можно дать определение объектно-ориентированного программирования.

Объектно-ориентированное программирование — это такой подход к программированию, при котором программа представляет собой множество взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию **наследования**.

Базовый класс

Построим первый вариант описания класса *Логический элемент* (TLogElement). Обозначим его входы как *In1* и *In2*, а выход назовём *Res* (от англ. *result* — результат) (рис. 7.11). Здесь состояние логического элемента определяется тремя величинами (*In1*, *In2* и *Res*), это позволяет на основе того же самого базового класса моделировать и элементы с памятью (например, триггеры), а не только статические элементы (как НЕ, И, ИЛИ и т. п.).

ЛогЭлемент
In1 (вход 1) *In2* (вход 2) *Res* (результат)
calc

Рис. 7.11

Любой логический элемент должен уметь вычислять значение выхода по известным входам, для этого введём в класс метод calc:

```
type
  TLogElement = class
    In1, In2: boolean;
    Res: boolean;
    procedure calc;
  end;
```

В таком варианте все данные открытые (общедоступные). Чтобы защитить внутреннее устройство объекта, скроем внутренние поля (добавим при этом в их названия первую букву «F») и введём свойства:

```
type
  TLogElement = class
  private
    FIn1, FIn2: boolean;
    FRes: boolean;
    procedure setIn1(newIn1: boolean);
    procedure setIn2(newIn2: boolean);
    procedure calc;
  public
    property In1: boolean read FIn1 write setIn1;
    property In2: boolean read FIn2 write setIn2;
    property Res: boolean read FRes;
  end;
```

Обратите внимание, что свойство Res — это свойство только для чтения, и другие объекты не могут его менять. Кроме того, мы поместили процедуру calc в скрытый раздел (**private**), потому что пересчёт результата должен выполняться автоматически при изменении любого входного сигнала (другие объекты не должны об этом беспокоиться).

Несложно написать процедуру setIn1 (и аналогичную ей процедуру setIn2), в ней новое входное значение присваивается полю и сразу пересчитывается результат:

```
procedure TLogElement.setIn1(newIn1: boolean);
begin
  FIn1:=newIn1;
  calc;
end;
```

Если внимательно проанализировать построенное описание класса, можно выявить несколько проблем. Во-первых, элемент НЕ имеет только один вход, поэтому не хотелось бы для него открывать доступ к свойству *In2* (это не нужно и может привести к ошибкам).

Во-вторых, процедуру calc невозможно написать, пока мы не знаем, какой именно логический элемент моделируется. Вместе с тем мы знаем, что такую процедуру имеет любой логический элемент, т. е. она должна принадлежать именно классу TLogElement. Здесь можно написать процедуру-«заглушку» (которая ничего не делает):

```
procedure TLogElement.calc;
begin
end;
```

Но нужно как-то дать возможность классам-наследникам изменить этот метод так, чтобы он выполнял нужную операцию. Такой метод называется *виртуальным*. Более точное определение этого понятия мы дадим несколько позже.

Классы-наследники могут по-разному реализовывать один и тот же метод. Такая возможность называется полиморфизмом.

Полиморфизм (от греч. πολυ — много, и μορφη — форма) — это возможность классов-наследников по-разному реализовывать метод, описанный для класса-предка.

Мы уже говорили о том, что метод calc не нужно делать общедоступным (**public**). В то же время его нельзя делать закрытым (**private**), потому что в этом случае он не будет доступен классам-наследникам. В таких случаях в описании класса используется третий блок (кроме **private** и **public**), который называется **protected** (защищённый). Данные и методы в этом блоке доступны для классов-наследников, но недоступны для других классов. В этот же блок **protected** мы переместим и объявление свойства *In2* — оно будет скрыто для элемента «НЕ», а элементы с двумя входами его «откроют» (чуть позже).

```
type
TLogElement = class
  private
    FIn1, FIn2: boolean;
    FRes: boolean;
```

```
    procedure setIn1(newIn1: boolean);
    procedure setIn2(newIn2: boolean);
  protected
    property In2: boolean read FIn2 write setIn2;
    procedure calc; virtual; abstract;
  public
    property In1: boolean read FIn1 write setIn1;
    property Res: boolean read FRes;
  end;
```

Обратите внимание на объявление метода calc: после него стоят слова **virtual** (виртуальный) и **abstract** (в переводе с англ. — абстрактный). Описатель **virtual** говорит о том, что метод calc — виртуальный, и классы-наследники могут его переопределять. Как уже отмечалось, мы должны объявить этот метод (ввести его в описание класса), поскольку он должен быть у любого логического элемента. В то же время невозможно написать процедуру calc, пока неизвестен тип логического элемента. Такой метод называется абстрактным и обозначается описателем **abstract**. Для абстрактного метода не нужно ставить «заглушку».

Абстрактный метод — это метод класса, который объявляется, но не реализуется в классе.

Более того, не существует логического элемента «вообще», как не существует «просто фрукта», не относящегося к какому-то виду. Такой класс в ООП называется абстрактным. Его отличительная черта — хотя бы один абстрактный (нереализованный) метод.

Абстрактный класс — это класс, содержащий хотя бы один абстрактный метод.

Итак, полученный класс TLogElement — это абстрактный класс (компилятор определит это автоматически). *Его можно использовать только для разработки классов-наследников*, создать в программе объект этого класса нельзя.

Чтобы класс-наследник не был абстрактным, он должен переопределить все абстрактные методы предка, в данном случае метод calc. Как это сделать, вы увидите в следующем пункте.

Классы-наследники

Теперь займёмся классами-наследниками от TLogElement. Поскольку у нас будет единственный элемент с одним входом (НЕ), сделаем его наследником прямо от TLogElement (не будем вводить специальный класс «элемент с одним входом»).

```
type
  TNot = class(TLogElement)
    procedure calc; override;
  end;
```

После слова **class** в скобках указано название базового класса. Все объекты класса TNot обладают всеми свойствами и методами класса TLogElement.

Новый класс **переопределяет** метод calc, на это указывает слово **override** (в переводе с англ. — перекрыть). Заметим, что у базового класса TLogElement этот метод не реализован — он абстрактный, поэтому в данном случае мы фактически программируем метод, объявленный в базовом классе. Для элемента «НЕ» он выглядит очень просто:

```
procedure TNot.calc;
begin
  FRes:=not FIn1;
end;
```

Класс TNot уже не абстрактный, потому что абстрактный метод предка переопределён и теперь известно, что делать при вызове метода calc. Поэтому можно создавать объект этого класса и использовать его:

```
var n: TNot;
...
n:=TNot.Create;
n.In1:=False;
writeln(n.Res);
```

Остальные элементы имеют два входа и будут наследниками класса.

```
TLog2In = class(TLogElement)
public
  property In2;
end;
```

Единственное, что делает этот класс, — переводит свойство In2 в раздел **public**, т. е. делает его общедоступным. Отметим, что видимость можно только повышать, т. е. нельзя, например, в наследнике сделать общедоступное свойство класса-предка закрытым или защищённым.

Класс TLog2In — это тоже абстрактный класс, потому что он не переопределил метод calc. Это сделают его наследники TAnd (элемент И) и TOr (элемент ИЛИ), которые определяют конкретные логические элементы:

```
type
  TAnd = class(TLog2In)
    procedure calc; override;
  end;
  TOr = class(TLog2In)
    procedure calc; override;
  end;
```

Реализация переопределённого метода calc для элемента «И» выглядит так:

```
procedure TAnd.calc;
begin
  FRes:=FIn1 and FIn2;
end;
```

Для элемента «ИЛИ» этот метод определяется аналогично.

Обратим внимание на метод setIn1, введённый в базовом классе:

```
procedure TLogElement.setIn1(newIn1: boolean);
begin
  FIn1:=newIn1;
  calc;
end;
```

В нём вызывается метод calc, который пересчитывает значение на выходе логического элемента при изменении входа. Какой же метод будет вызван, если в базовом классе TLogElement он только объявлен, но не реализован?

Проблема в том, что для вызова любой процедуры нужно знать её адрес в памяти. Для обычных методов транслятор сразу записывает в машинный код нужный адрес, потому что он заранее известен. Это так называемое **статическое связывание** (связывание на этапе трансляции), при выполнении программы этот адрес не меняется.

В нашем случае адрес метода неизвестен: в классе TLogElement его нет вообще, а у каждого класса-наследника адрес метода calc свой собственный. Чтобы выйти из положения, используется **динамическое связывание**, т. е. адрес вызываемой процедуры определяется при выполнении программы, когда уже определён тип объекта, с которым мы работаем. Такой метод нужно объявлять виртуальным, что мы и сделали ранее. Это означает не только то, что его могут переопределять наследники, но и то, что будет использоваться динамическое связывание. Теперь можно дать полное определение виртуального метода.

Виртуальный метод — это метод базового класса, который могут переопределить классы-наследники, при этом конкретный адрес вызываемого метода определяется только при выполнении программы.

Теперь мы готовы к тому, чтобы создавать и использовать построенные логические элементы. Например, таблицу истинности для последовательного соединения элементов «И» и «НЕ» можно построить так:

```
var elNot: TNot;
    elAnd: TAnd;
    A, B: boolean;
begin
  elNot:=TNot.Create;
  elAnd:=TAnd.Create;
  writeln('| A | B | not(A&B) ');
  writeln('-------------------');
  for A:=False to True do begin
    elAnd.In1:=A;
    for B:=False to True do begin
      elAnd.In2:=B;
      elNot.In1:=elAnd.res;
      writeln('| ', integer(A), ' | ', integer(B),
              ' | ', integer(elNot.Res))
    end
  end
end.
```

Сначала создаются два объекта — логические элементы НЕ (класс TNot) и ИЛИ (класс TAnd). Далее в двойном цикле перебираются все возможные комбинации значений логических переменных *A* и *B*, они подаются на входы элемента И, а его выход — на вход элемента НЕ. Чтобы при выводе логических значений вместо False и True выводились более компактные обозначения 0 и 1, значения входов и выхода преобразуются к целому типу (**integer**).

Модульность

Как вы знаете из главы 6, большие программы обычно разбивают на модули — внутренне связные, но слабо связанные между собой блоки. Такой подход используется как в классическом программировании, так и в ООП.

В нашей программе с логическими элементами в отдельный модуль можно вынести всё, что относится к логическим элементам. Модуль, содержащий классы логических элементов, на объектной версии языка Паскаль можно записать так:

```
unit log_elem;
{$mode objfpc}
interface
type
  TLogElement = class
  private
    FIn1, FIn2: boolean;
    FRes: boolean;
    procedure setIn1(newIn1: boolean);
    procedure setIn2(newIn2: boolean);
  protected
    property In2: boolean read FIn2 write setIn2;
    procedure calc; virtual; abstract;
  public
    property In1: boolean read FIn1 write setIn1;
    property Res: boolean read FRes;
  end;
  TNot = class(TLogElement)
    procedure calc; override;
  end;
  TLog2In = class(TLogElement)
  public
```

```
    property In2;
  end;
  TAnd = class(TLog2In)
    procedure calc; override;
  end;
  TOr = class(TLog2In)
    procedure calc; override;
  end;

implementation
  procedure TLogElement.setIn1(newIn1: boolean);
  begin
    FIn1:=newIn1; calc;
  end;
  procedure TLogElement.setIn2(newIn2: boolean);
  begin
    FIn2:=newIn2; calc;
  end;
  procedure TNot.calc;
  begin
    FRes:=not FIn1;
  end;
  procedure TAnd.calc;
  begin
    FRes:=FIn1 and FIn2;
  end;
  procedure TOr.calc;
  begin
    FRes:=FIn1 or FIn2;
  end;
end.
```

Чтобы использовать такой модуль, нужно подключить его в основной программе с помощью ключевого слова **uses**, после которого через запятую перечисляются все используемые модули:

```
program logic;
{$mode objfpc}
uses log_elem;
var elNot: TNot;
    elAnd: TAnd;
    ...
```

```
begin
  elNot:=TNot.Create;
  elAnd:=TAnd.Create;
  ...
end.
```

Сообщения между объектами

Когда логические элементы объединяются в сложную схему, желательно, чтобы передача сигналов между ними при изменении входных данных происходила автоматически. Для этого можно немного расширить базовый класс TLogElement, чтобы элементы могли передавать друг другу сообщения об изменении своего выхода.

Для простоты будем считать, что выход любого логического элемента может быть подключён к любому (но только одному!) входу другого логического элемента. Добавим к описанию класса два поля и один метод:

```
type
  TLogElement = class
  private
    FNextEl: TLogElement;
    FNextIn: integer;
    ...
  public
    procedure Link(nextElement: TLogElement;
                   nextIn: integer);
    ...
  end;
```

Поле *FNextEl* хранит ссылку на следующий логический элемент, а поле *FNextIn* — номер входа этого следующего элемента, к которому подключён выход данного элемента. С помощью общедоступного метода Link можно связать данный элемент со следующим:

```
procedure TLogElement.Link(nextElement: TLogElement;
                            nextIn: integer);
begin
  FNextEl:=nextElement;
  FNextIn:=nextIn;
end;
```

Нужно немного изменить методы setIn1 и setIn2: при изменении входа они должны не только пересчитывать выход данного элемента, но и отправлять сигнал на вход следующего

```
procedure TLogElement.setIn1(newIn1: boolean);
begin
  FIn1:=newIn1; calc;
  if FNextEl<>nil then
    case FNextIn of
      1: FNextEl.In1:=res;
      2: FNextEl.In2:=res;
    end;
end;
```

Условие FNextEl<>nil означает «если следующий элемент задан». Если он не был установлен, значение поля *FNextEl* будет равно nil и никакие дополнительные действия не выполняются.

С учётом этих изменений вывод таблицы истинности функции И-НЕ можно записать так (операторы вывода заменены многоточиями):

```
elNot:=TNot.Create;
elAnd:=TAnd.Create;
elAnd.Link(elNot, 1);
...
for A:=False to True do begin
  elAnd.In1:=A;
  for B:=False to True do begin
    elAnd.In2:=B;
    ...
  end;
end;
```

Обратите внимание, что в самом начале мы установили связь элементов И и НЕ с помощью метода Link (связали выход элемента И с первым входом элемента НЕ). Далее в теле цикла обращения к элементу НЕ нет, потому что элемент И автоматически сообщит ему об изменении своего выхода.

Вопросы и задания

1. Что такое классификация? Зачем она нужна? Приведите примеры.
2. В каком случае можно сказать: «Класс *Б* — наследник класса *А*», а когда: «Объект класса *А* содержит объект класса *Б*»? Приведите примеры.

3. Что такое иерархия классов?
4. Объясните приведённую иерархию логических элементов. Обсудите её достоинства и недостатки.
5. Дайте полное определение ООП и объясните его.
6. Что такое базовый класс и класс-наследник? Какие синонимы используются для этих терминов?
7. На примере класса `TLogElement` (пример из параграфа) покажите, как выполнена инкапсуляция.
8. Что такое виртуальный метод?
9. Что такое полиморфизм?
10. Что такое абстрактный класс? Почему нельзя создать объект этого класса?
11. Как транслятор определяет, что тот или иной класс — абстрактный?
12. Что нужно сделать, чтобы класс-наследник абстрактного класса не был абстрактным?
13. Зачем нужен описатель **protected**? Чем он отличается от **private** и **public**?
14. Что означает описатель **override**?
15. Какие преимущества даёт применение модулей в программе?
16. Из каких частей состоит каждый модуль? Что включают в каждую из них?
17. Можно ли всё содержимое модуля включить в секцию **interface**? Чем это плохо?
18. Можно ли всё содержимое модуля включить в секцию **implementation**? Чем это плохо?
19. Объясните, как объекты могут передавать сообщения друг другу.

Подготовьте сообщение

а) «Иерархия классов в языке Си»

б) «Иерархия классов в языке Javascript»

в) «Иерархия классов в языке Python»

Задачи

1. Добавьте в описанную в параграфе иерархию классов элементы «исключающее ИЛИ», «И-НЕ» и «ИЛИ-НЕ».
2. «Соберите» в программе RS-триггер из двух логических элементов «ИЛИ-НЕ», постройте его таблицу истинности (обратите внимание на вариант, когда оба входа нулевые).

§ 51
Программы с графическим интерфейсом

Особенности современных прикладных программ

Большинство современных программ, предназначенных для пользователей, управляется с помощью графического интерфейса. Вы знакомы с понятиями «окно программы», «кнопка», «флажок», «поле ввода», «полоса прокрутки» и т. п. Такие оконные системы чаще всего построены на принципах объектно-ориентированного программирования, т. е. все элементы окон — это объекты, которые обмениваются данными, посылая друг другу сообщения.

Сообщение — это блок данных определённой структуры, который используется для обмена информацией между объектами.

В сообщении указываются:
- адресат (объект, которому посылается сообщение);
- числовой код (тип) сообщения;
- параметры (дополнительные данные), например координаты щелчка мышью или код нажатой клавиши.

Сообщение может быть **широковещательным**, в этом случае вместо адресата указывается особый код и сообщение поступает всем объектам определённого типа (например, всем главным окнам программ).

В программах, которые мы писали раньше, последовательность действия заранее определена — основная программа выполняется строчка за строчкой, вызывая процедуры и функции, все ветвления выполняются с помощью условных операторов (рис. 7.12).

В современных программах порядок действий определяется пользователем, другими программами или поступлением новых данных из внешнего источника (например, из сети), поэтому классическая схема не подходит. Пользователь текстового редактора может щёлкать на любых кнопках и выбирать любые пункты меню в произвольном порядке. Программа-сервер, передающая данные с веб-сайта на компьютер пользователя, начинает действовать только при поступлении очередного запроса. При

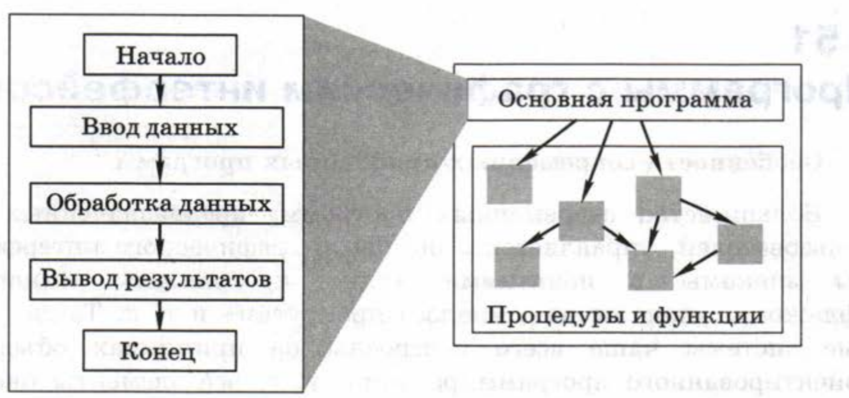


Рис. 7.12

программировании сетевых игр нужно учитывать взаимодействие многих объектов, информация о которых передаётся по сети в случайные моменты времени.

Во всех этих примерах программа должна «включаться в работу» только тогда, когда получит условный сигнал, т. е. произойдёт некоторое *событие* (изменение состояния).

Событие — это переход какого-либо объекта из одного состояния в другое.

События могут быть вызваны действиями пользователя (управление клавиатурой и мышью), сигналами от внешних устройств (переход принтера в состояние готовности, получение данных из сети), получением сообщения от другой программы. При наступлении событий объекты посылают друг другу сообщения.

Таким образом, весь ход выполнения современной программы определяется происходящими событиями, а не жёстко заданным алгоритмом. Поэтому говорят, что программы управляются событиями, а соответствующий стиль программирования называют **событийно-ориентированным**, т. е. основанным на обработке событий.

Нормальный режим работы событийно-ориентированной программы — цикл обработки сообщений. Все сообщения (от мыши, клавиатуры, драйверов устройств ввода и вывода и т. п.) сначала поступают в единую очередь сообщений операционной системы. Кроме того, для каждой программы операционная система соз-

даёт отдельную очередь сообщений и помещает в неё все сообщения, предназначенные именно этой программе (рис. 7.13).

Рис. 7.13

Программа выбирает очередное сообщение из очереди и вызывает специальную процедуру — обработчик этого сообщения (если он есть). Когда пользователь закрывает окно программы, ей посылается специальное сообщение, при получении которого цикл (и работа всей программы) завершается.

Таким образом, главная задача программиста — написать содержание обработчиков всех нужных сообщений. Ещё раз подчеркнём, что последовательность их вызовов точно не определена, она может быть любой в зависимости от действий пользователя и сигналов, поступающих с внешних устройств.

RAD-среды для разработки программ

Разработка программ для оконных операционных систем до середины 1990-х гг. была довольно сложным и утомительным делом. Очень много усилий уходило на то, чтобы написать команды для создания интерфейса с пользователем: разместить элементы в окне программы, написать и правильно оформить обработчики сообщений. Значительную часть своего времени программист занимался трудоёмкой работой, которая почти никак не связана с решением главной задачи. Поэтому возникла естественная мысль — автоматизировать описание окон и их элементов так, чтобы весь интерфейс программы можно было построить без ручного программирования (чаще всего с помощью мыши), а человек думал бы о сути задачи, т. е. об алгоритмах обработки данных.

Такие системы программирования получили название **сред быстрой разработки приложений — RAD-сред** (от англ. *Rapid Application Development*). Разработка программы в RAD-системе состоит из следующих этапов:

- создание формы (так называют шаблон, по которому строится окно программы или диалога); при этом минимальный код добавляется автоматически и сразу получается работоспособная программа;
- расстановка на форме элементов интерфейса (полей ввода, кнопок, списков) с помощью мыши и настройка их свойств;
- создание обработчиков событий;
- написание алгоритмов обработки данных, которые выполняются при вызове обработчиков событий.

Обратите внимание, что при программировании в RAD-средах обычно говорят не об обработчиках сообщений, а об обработчиках событий. Событием в программе может быть не только нажатие клавиши или щелчок мышью, но и перемещение окна, изменение его размеров, начало и окончание выполнения расчётов и т. д. Некоторые сообщения, полученные от операционной системы, библиотека RAD-среды «транслирует» (переводит) в соответствующие события, а некоторые — нет. Более того, программист может вводить свои события и определять процедуры для их обработки.

Одной из первых сред быстрой разработки стала среда Delphi, разработанная фирмой Borland в 1994 г. Самая известная современная профессиональная RAD-система — *Microsoft Visual Studio* — поддерживает несколько языков программирования. Далее для выполнения практических работ мы будем использовать свободную RAD-среду *Lazarus* (lazarus.freepascal.org), которая во многом аналогична Delphi, но позволяет создавать кроссплатформенные программы (для операционных систем Windows, Linux, Mac OS X и др.).

Среды RAD позволили существенно сократить время разработки программ. Однако нужно помнить, что любой инструмент — это только инструмент, который можно использовать грамотно или безграмотно. Использование среды RAD само по себе не гарантирует, что у вас автоматически получится хорошая программа с хорошим пользовательским интерфейсом.

Вопросы и задания

1. Что такое графический интерфейс?
2. Как связан графический интерфейс с объектно-ориентированным подходом к программированию?
3. Что такое сообщение? Какие данные в него входят?
4. Что такое широковещательное сообщение?
5. Что такое обработчик сообщения?
6. Чем принципиально отличаются современные программы от классических?
7. Что такое событие? Какое программирование называют событийно-ориентированным?
8. Как работает событийно-ориентированная программа?
9. Какие причины сделали необходимым создание сред быстрой разработки программ? В чем их преимущество? Приведите примеры.
10. Расскажите про этапы разработки программы в RAD-среде.
11. Объясните разницу между понятиями «событие» и «сообщение».

Подготовьте сообщение

а) «Обработка сообщений в операционных системах»
б) «Современные среды быстрой разработки программ»
в) «Программы с графическим интерфейсом на Python»

§ 52
Основы программирования в RAD-средах

Общий подход

В этом разделе мы продемонстрируем основные принципы программирования в RAD-средах на примере среды Lazarus, которая распространяется свободно и может работать в различных операционных системах — Windows, Mac OS X, Linux. Тем не менее все изучаемые здесь принципы справедливы также и для других аналогичных программ, например Delphi и Visual Studio.

Разработка программы начинается с создания **проекта**. Так называется набор файлов, из которых компилятор строит исполняемый файл программы. В состав проекта обычно входят:

- проект (файл с расширением lpr[1], от Lazarus Project — проект Lazarus), в котором содержится основная программа;

[1] Здесь и далее расширения имён файлов указаны для среды Lazarus.

- настройки проекта (lpi, от Lazarus Project Information — информация о проекте Lazarus);
- модули, из которых состоит программа (pas);
- формы (lfm, от Lazarus Form — форма Lazarus) — описания внешнего вида и свойств окон и их элементов.

В программе с графическим интерфейсом может быть несколько окон, которые называют **формами.** С каждой формой связана пара файлов: в одном (с расширением lfm) хранятся данные о расположении и свойствах элементов интерфейса, а во втором (с расширением pas) — программный код обработчиков сообщений, связанных с этой формой.

Одна форма — *главная*, она появляется на экране при запуске программы. Когда пользователь закрывает главную форму, работа программы завершается.

Простейшая программа

Для создания проекта в Lazarus нужно выбрать пункт меню **Файл – Создать** и в появившемся окне отметить вариант **Проект – Приложение.** При этом создаётся вполне рабочая программа, которую сразу же можно запустить на выполнение клавишей **F9.**

При работе в среде Lazarus используются четыре окна (рис. 7.14):

- главное окно;
- окно Инспектора объектов;
- окно исходного кода;
- окно формы.

Главное окно среды (расположенное сверху) содержит меню, кнопки для быстрого вызова команд и **палитру (библиотеку) компонентов.** Компонентами называются готовые объекты (кнопки, поля ввода, списки и т. п.), которые можно использовать в программах.

Вся программа, согласно принципам ООП, состоит из объектов. Для настройки свойств объектов используется окно **Инспектора объектов,** которое состоит из двух частей. В верхней части показано дерево объектов. В простейшей программе мы увидим всего один объект — форму с именем `Form1`. В нижней части окна несколько вкладок, самые важные из них — **Свойства,** где можно изменить общедоступные свойства объекта, и **События,** на которой устанавливаются обработчики событий этого объекта.

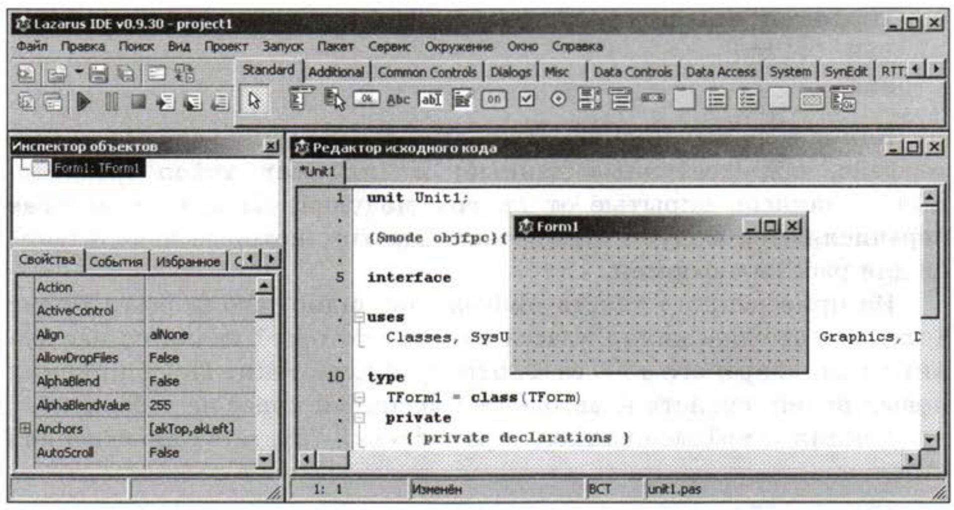


Рис. 7.14

В **окно формы** можно мышью перетаскивать компоненты с палитры и изменять их размеры и расположение. Таким образом, интерфейс программы полностью строится с помощью мыши.

С каждой формой связан программный модуль, в котором описываются классы и обработчики событий, используемые этой формой. Файлы формы и соответствующего ей модуля имеют одинаковое имя, но разные расширения. По умолчанию созданная главная форма программы называется Form1, а модуль — Unit1. Содержание модуля примерно такое:

```
unit Unit1;
interface
uses
  Classes, SysUtils, FileUtil, Forms, Controls,
  Graphics, Dialogs;
type
  TForm1=class(TForm)
  private
    {private declarations}
  public
    {public declarations}
  end;
var
  Form1: TForm1;
```

```
implementation
{$R *.lfm}
end.
```

Как и в любом модуле, здесь есть две секции: **interface** (интерфейс, общедоступные данные) и **implementation** (реализация — данные, скрытые от других модулей). После слова **uses** перечисляются модули библиотеки Lazarus, которые используются для работы с формой.

Из приведённого текста программы видно, что форма (объект `Form1`) — это экземпляр класса `TForm1`, который является наследником стандартного класса `TForm` из библиотеки. Пока никаких новых полей, свойств и методов в созданный класс не добавлено.

Секция **implementation** практически пуста. Единственная строчка

```
{$R *.lfm}
```

похожа на комментарий (в фигурных скобках), однако транслятор обращает на неё внимание. Эта команда подключает файл с тем же именем, что и у модуля, но с расширением lfm (описание формы и размещённых на ней компонентов).

Как вы знаете, модуль не может выполняться самостоятельно. Основная программа находится в файле проекта. Для того чтобы увидеть его, нужно нажать клавиши Ctrl+F12 и выбрать файл с расширением lpr. Этот файл будет загружен в отдельную вкладку редактора:

```
program project1;
uses
  Interfaces, Forms, Unit1;
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

В начале файла подключаются используемые модули `Interfaces` и `Forms` из библиотеки Lazarus, а также модуль `Unit1`, связанный с нашей формой.

Вся программа — это объект с именем `Application`, который относится к классу `TApplication`. Этот объект создаётся автоматически при загрузке модуля `Forms`. В основной программе вызываются три его метода: `Initialize` (начальные установки),

CreateForm (создание формы) и Run (запуск). Цикл обработки сообщений скрыт внутри метода Run, так что здесь тоже использован принцип инкапсуляции (скрытия внутреннего устройства).

Свойства объектов

Инспектор объектов позволяет изменять свойства выделенного объекта (например, формы) и устанавливать обработчики событий для этого объекта. Для этого можно использовать мышь или клавиатуру. Например, можно заметить, что при перемещении формы изменяются её свойства Left (в переводе с англ. — левый) и Top (в переводе с англ. — верхний), которые задают координаты левого верхнего угла формы на экране. В то же время можно вручную изменить эти координаты, вводя новые значения в Инспекторе объектов (при этом форма передвигается).

Если изменять мышью размеры формы (перемещая её границы), то будут изменяться свойства Width (в переводе с англ. — ширина) и Height (в переводе с англ. — высота).

Свойство Name (в переводе с англ. — имя) — это название объекта-формы в программе (имя переменной). В имени можно использовать только латинские буквы, цифры и знак подчёркивания. Если изменить название формы в Инспекторе объектов, скажем, на MainForm, то это название автоматически изменится и в тексте модуля этой формы. Более того, название класса тоже изменится на TMainForm. Это означает, что многие изменения вносятся в код программы автоматически.

Перечислим ещё некоторые важные свойства формы:

- Caption — текст в заголовке окна;
- Color — цвет рабочей области;
- Font — шрифт надписей;
- Visible — видимость (да/нет).

Обработчики событий

На вкладке **События** перечислены все события, которые может обрабатывать форма. Названия их обработчиков в Инспекторе объектов начинаются с букв On (в переводе с англ. — в ответ на...). Чтобы создать обработчик, нужно дважды щёлкнуть мышью на поле справа от его названия. При этом открывается окно редактора и в текст модуля автоматически добавляется пустой обработчик события (шаблон), в который остаётся только добавить нужные команды.

Рассмотрим простой пример. Вы знаете, что многие программы запрашивают подтверждение, когда пользователь завершает их работу. Для этого можно использовать обработчик OnCloseQuery (в переводе с англ. — запрос на закрытие). Обработчик, созданный двойным щелчком мышью, имеет вид:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
                                 var CanClose: boolean);
begin
end;
```

Одновременно этот метод добавляется в описание класса TForm1 (выше секции **private**).

Как видно из заголовка процедуры, в обработчик передаются два параметра:

- Sender — ссылка на объект, от которого пришло сообщение о событии (в данном случае это будет сама форма);
- CanClose — изменяемый логический параметр, в который нужно записать результат запроса: истинное значение означает, что можно закрывать окно, ложное — что нельзя.

В Lazarus есть стандартная функция MessageDlg, которая выводит на экран запрос с несколькими кнопками (рис. 7.15) и позволяет получить ответ пользователя (код нажатой кнопки).

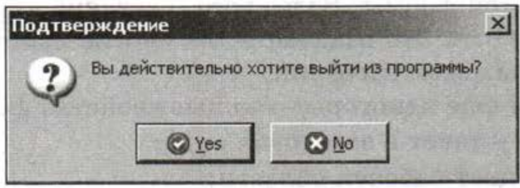


Рис. 7.15

В тело обработчика можно добавить условный оператор, который запишет в переменную CanClose значение True, если пользователь подтвердил выход из программы:

```
procedure TMainForm.FormCloseQuery(Sender: TObject;
                                   var CanClose: boolean);
var res: TModalResult;
begin
  res:=MessageDlg('Подтверждение',
        'Вы действительно хотите выйти из программы?',
        mtConfirmation, [mbYes,mbNo], 0);
```

```
    CanClose:=(res=mbYes);
  end;
```

Здесь вызывается функция MessageDlg, и её результат записывается в переменную *res* типа TModalResult. Если это значение совпадает со встроенной константой *mbYes* (т. е. пользователь нажал на кнопку **Yes**), в переменную *CanClose* записывается значение True, и программа завершается, иначе команда отменяется.

Функции MessageDlg передаются пять параметров:

- заголовок окна;
- текст вопроса;
- тип запроса, определяющий рисунок слева от текста:

mtError	ошибка;
mtWarning	предупреждение;
mtInformation	информация;
mtConfirmation	подтверждение;

- набор (множество) кнопок, которые появляются под текстом; в нашем случае это кнопки **Yes** и **No**, обозначенные константами *mbYes* и *mbNo*;
- номер раздела справочной системы, в котором есть объяснение этой ситуации (у нас нет справочной системы, поэтому ставим 0).

Итак, мы построили простейшую работоспособную программу и познакомились со средой Lazarus. В следующем параграфе вы узнаете, как работать с компонентами.

Вопросы и задания

1. В каком смысле используется термин «проект» в программировании?
2. Из каких файлов состоит типичный проект в RAD-среде?
3. Что такое форма? Почему для описания формы в Lazarus используются два файла?
4. Какие основные окна используются в среде Lazarus? Зачем они нужны?
5. Покажите, что программа, написанная с помощью Lazarus, состоит из объектов.
6. С помощью какого механизма транслятор «подключает» форму?
7. Где расположена основная программа в проекте Lazarus? Объясните все команды основной программы.
8. Почему в основной программе не виден цикл обработки сообщений?

9. Назовите некоторые важнейшие свойства формы. Какими способами можно их изменять?
10. Приведите примеры автоматического построения и изменения кода в RAD-среде.
11. Как создать новый обработчик события? Подумайте, можно ли сделать это вручную.
12. Как передаются параметры сообщения в обработчик?
13. Как можно вывести сообщение об ошибке на экран?

Подготовьте сообщение

«Простая программа на языке C# в Visual Studio»

Задача

Попробуйте изменять какие-нибудь свойства формы, построив обработчик ещё одного события (например, `OnShow` — вывод формы на экран; `OnClick` — щелчок мыши; `OnResize` — изменение размеров).

§ 53 Использование компонентов

Программа с компонентами

В главном окне Lazarus расположена так называемая **палитра компонентов** (рис. 7.16) — библиотека готовых объектов, которые можно добавить в свою программу, просто перетащив их мышью на форму.

Рис. 7.16

Компоненты разбиты на группы. Мы будем использовать компоненты из групп **`Standard`** (Стандартные), **`Additional`** (Дополнительные) и **`Dialogs`** (Диалоги). Каждый значок обозначает определённый компонент. Если задержать указатель мыши над значком компонента, в тексте всплывающей подсказки можно прочитать его название.

Построим простую программу для просмотра рисунков, используя готовые компоненты. В верхней части (формы) на панели разместим кнопку для загрузки файла и флажок-выключатель, который изменяет масштаб рисунка так, чтобы он вписывался в отведённое ему место (рис. 7.17).

Рис. 7.17

Создадим новый проект (как в предыдущем параграфе), изменим имя формы (свойство `Name`) на `MainForm`, а её заголовок (свойство `Caption`) — на «Просмотр рисунков».

Добавим на форму панель — компонент `TPanel` из группы **Standard** (Стандартные). Для этого можно перетащить эту кнопку на форму или щёлкнуть на кнопке и нарисовать прямоугольник, ограничивающий панель. Теперь размеры панели можно изменять, перетаскивая маркеры на границах или изменяя значения свойств `Width` (ширина) и `Height` (высота) в Инспекторе объектов. Панель можно перетаскивать мышью по форме. Хотелось бы, чтобы панель была всё время прижата к верхней границе окна и её размеры изменялись вместе с размерами окна. Для этого нужно установить свойство `Align` (выравнивание) равным `alTop` (англ. *align top* — выровнять по верху).

На созданной панели можно заметить надпись «Panel1», которая нам не нужна. Чтобы убрать её, нужно стереть значение свойства `Caption` (в переводе с англ. — заголовок) панели.

Теперь панель готова, на ней нужно разместить кнопку (компонент `TButton`) и флажок (компонент `TCheckBox`). На кнопке должна быть надпись «Открыть файл» (свойство `Caption`), а справа от флажка — текст «По размерам окна» (тоже свойство `Caption`) — рис. 7.18. Размеры и расположение компонентов нужно поменять с помощью мыши. Имена компонентов

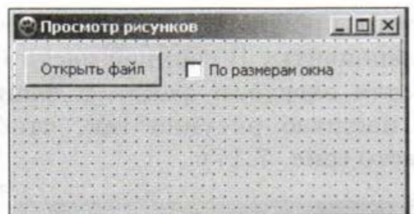


Рис. 7.18

(свойства `Name`) тоже можно изменить, например, на `OpenBtn` и `SizeCb` (имена объектов строятся по тем же правилам, что и имена переменных в Паскале).

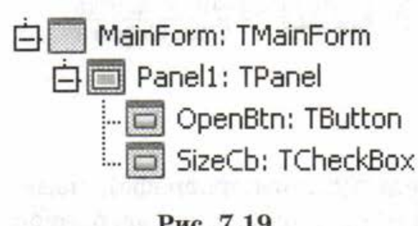


Рис. 7.19

В верхней части Инспектора объектов можно увидеть структуру объектов формы в виде дерева (рис. 7.19). Главный объект — это сама форма `MainForm`, она является *родительским объектом* для панели `Panel1`. Это означает, что при перемещении формы панель перемещается вместе с ней. В свою очередь, панель — это родительский объект для кнопки и флажка. Для того чтобы редактировать свойства и методы какого-то объекта в Инспекторе объектов, его можно выделить прямо на форме или в дереве объектов.

Теперь добавим на форму специальный объект `TImage` (группа **Additional**), который «умеет» отображать рисунки различных форматов. Для того чтобы он заполнял все свободное пространство (кроме панели), нужно установить для него выравнивание `alClient` (свойство `Align`). Изменим название объекта на `Image`.

Остается решить два вопроса:

1) как сделать выбор файла и загрузку его в компонент `Image`;
2) как подгонять размер рисунка по размеру формы.

К счастью, для этого достаточно использовать возможности готовых компонентов. На панели **Dialogs** (Диалоги) есть готовый компонент для выбора рисунка на диске, он называется `TOpenPictureDialog`. У него есть метод `Execute` — функция, которая вызывает *стандартный диалог* выбора файла и возвращает

логическое значение: True, если файл успешно выбран, и False, если пользователь отказался от выбора файла. Имя выбранного файла можно получить, прочитав свойство FileName этого компонента.

Добавим компонент TOpenPictureDialog на форму (в любое место). Это невизуальный компонент, его не будет видно во время выполнения программы. В Инспекторе объектов можно проверить, что для него родительским объектом будет сама форма. Для краткости изменим его название на OpenDlg.

Теперь в случае щелчка на кнопке нужно вызвать метод OpenDlg.Execute и, если он вернёт значение True, загрузить выбранный файл, имя которого получается как значение свойства OpenDlg.FileName. Щелчок на кнопке — это событие, обработчик которого называется OnClick. Создадим шаблон этого обработчика в Инспекторе объектов и запишем в него команды:

```
if OpenDlg.Execute then
  Image.Picture.LoadFromFile( OpenDlg.FileName );
```

Поясним загрузку файла. Компонент Image имеет свойство Picture, в котором хранится изображение. Это тоже объект, у которого, в свою очередь, есть метод LoadFromFile (загрузить из файла), этому методу передаётся имя файла, выбранного пользователем. Теперь можно запустить программу и проверить, как она работает.

Обратите внимание, что изображение выводится в масштабе 1:1 независимо от размера окна. Флажок **По размерам окна** можно включать и выключать, но он никак не влияет на результат. Чтобы исправить ситуацию, будем использовать событие изменения состояния флажка, его обработчик называется OnChange (при изменении). У объекта Image есть логическое свойство Proportional (пропорциональный), если ему присвоить значение True, компонент Image сам выполнит подгонку размеров рисунка под размер свободной области. Таким образом, обработчик события OnChange компонента SizeCb содержит такой оператор:

```
Image.Proportional:=SizeCb.Checked;
```

Свойство Checked (в переводе с англ. — отмечен) — это логическое значение, определяющее состояние выключателя: если он включен, это свойство равно True, если выключен, то False.

Теперь можно запустить готовую программу и проверить её работу. Отметим следующие важные особенности:

- программа целиком состоит из объектов и основана на идеях ООП;
- мы построили программу практически без программирования;
- использование готовых компонентов скрывает от нас сложность выполняемых операций, поэтому скорость разработки программ значительно повышается.

Ввод и вывод данных

Во многих программах нужно, чтобы пользователь вводил текстовую или числовую информацию. Чаще всего для ввода данных применяют поле ввода — компонент TEdit (вкладка **Standard**). Для доступа к введённой строке используют его свойство Text (в переводе с англ. — текст).

Шрифт текста в поле ввода задаётся сложным свойством Font (*англ.* шрифт). Это объект, у которого есть свои свойства, их список можно увидеть в Инспекторе объектов, если щёлкнуть на значке [+]. Например, свойство Size — размер шрифта в пунктах, а Style — свойство-множество, в которое могут входить стили оформления fsBold (жирный), fsItalic (курсив), fsUnderline (подчёркнутый). Если установить шрифт для какого-то объекта, например для формы, все дочерние компоненты по умолчанию будут иметь такой же шрифт.

Программа, которую мы сейчас построим, будет переводить RGB-составляющие цвета в соответствующий шестнадцатеричный код, который используется для задания цвета в языке HTML (см. § 26).

На форме будут расположены (рис. 7.20):

- три поля ввода (в них пользователь может задать значения красной, зелёной и синей составляющих цвета в модели RGB);
- прямоугольник (компонент TShape из группы **Additional**), цвет которого изменяется согласно введённым значениям;
- несколько меток (компонентов TLabel).

Метки — это надписи, которые пользователь не может редактировать, однако их содержание можно изменять из программы через свойство Caption.

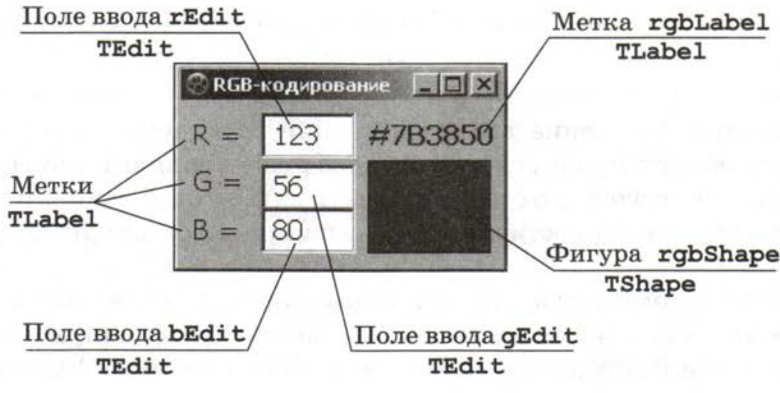


Рис. 7.20

Во время работы программы будут использоваться **поля ввода** rEdit, gEdit и bEdit, метка rgbLabel, с помощью которой будет выводиться результат — код цвета, и фигура rgbShape. В качестве начальных значений полей можно ввести любые целые числа от 0 до 255 (свойство Text).

При изменении содержимого одного из трёх полей ввода нужно обработать введённые данные и вывести результат в заголовок (свойство Caption) метки rgbLabel, а также изменить цвет заливки для фигуры rgbShape. Обработчик события, которое происходит при изменении текста в поле ввода, называется OnChange. Так как при изменении любого из трёх полей нужно выполнить одинаковые действия, для этих компонентов можно установить один и тот же обработчик. Для этого нужно выделить их, удерживая клавишу Shift, и после этого создать новый обработчик двойным щелчком в Инспекторе объектов.

Для того чтобы преобразовать текст из поля ввода в соответствующее целое число, используется стандартная функция StrToInt (для обратного перевода применяется функция IntToStr). Обработчик события OnChange для поля ввода может выглядеть так:

```
procedure TForm1.rEditChange(Sender: TObject);
var r, g, b: integer;
begin
  r:=StrToInt(rEdit.Text);
  g:=StrToInt(gEdit.Text);
  b:=StrToInt(bEdit.Text);
  rgbShape.Brush.Color:=RGBToColor(r,g,b);
```

```
    rgbLabel.Caption:='#'+IntToHex(r,2)+IntToHex(g,2)
                            +IntToHex(b,2);
end;
```

Поясним последние две строки. Фигура класса TShape имеет свойство-объект Brush, которое определяет заливку внутренней области. Свойство Color этого объекта задаёт цвет заливки, который мы строим из составляющих с помощью стандартной функции RGBToColor.

Далее формируется строка, содержащая шестнадцатеричный код цвета. Для перевода значений в шестнадцатеричную систему используется функция IntToHex, второй её параметр 2 указывает на то, что число записывается с двумя знаками.

Вы можете заметить, что при запуске программы код цвета и цвет прямоугольника не изменяются, какие бы значения мы ни установили в полях ввода в Инспекторе объектов. Чтобы исправить ситуацию, нужно вызвать уже готовый обработчик из обработчика OnCreate формы (он вызывается при создании формы):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  rEditChange(rEdit);
end;
```

При вызове в скобках указан объект, который посылает сообщение о событии. Здесь в качестве источника указан компонент rEdit, но в данном случае можно было использовать любой объект, потому что параметр Sender в обработчике OnChange не используется.

Обработка ошибок

Если в предыдущей программе пользователь введёт не числа, а что-то другое (или пустую строку), программа выдаст сообщение о необработанной ошибке на английском языке и предложит завершить работу. Хорошая программа никогда не должна завершаться аварийно, для этого все ошибки, которые можно предусмотреть, надо обрабатывать.

В современных языках программирования есть так называемый **механизм исключений**, который позволяет обрабатывать практически все возможные ошибки. Для этого все «опасные» участки кода (на которых может возникнуть ошибка) нужно поместить в блок **try** — **except**:

```
try
  {"опасные" команды}
except
  {обработка ошибки}
end;
```

Слово *try* по-английски означает «попытаться», *except* — «исключение» (исключительная или ошибочная, непредвиденная ситуация). Программа попадает в блок **except — end** только тогда, когда между **try** и **except** произошла ошибка.

В нашей программе «опасные» команды — это операторы преобразования данных из текста в числа (вызовы функции StrToInt). В случае ошибки мы выведем вместо кода цвета знак вопроса. Улучшенный обработчик с защитой от неправильного ввода принимает вид:

```
try
  r:=StrToInt(rEdit.Text);
  g:=StrToInt(gEdit.Text);
  b:=StrToInt(bEdit.Text);
  rgbShape.Brush.Color:=RGBToColor(r,g,b);
  rgbLabel.Caption:='#'+IntToHex(r,2)+IntToHex(g,2)
                   + IntToHex(b,2);
except
  rgbLabel.Caption:='?';
end;
```

Чтобы увидеть результат такой обработки ошибок, нужно запускать программу отдельно, не из среды Lazarus, иначе система перехватывает ошибки в отладочном режиме.

Существует и другой способ защиты — блокировать при вводе символы, которых быть не должно (буквы, скобки и т. п.). В нашей программе для всех полей ввода можно установить такой обработчик события OnKeyPress (в переводе с англ. — при нажатии клавиши[1]):

```
procedure TForm1.rEditKeyPress(Sender: TObject;
                               var Key: char);
begin
  if not (Key in ['0'..'9',#8]) then Key:=#0;
end;
```

[1] Если в программе нужно обрабатывать русские буквы, используется обработчик OnUTF8Key.

Этому обработчику передаётся изменяемый параметр `Key` — символ, соответствующий нажатой клавише. Если этот символ не входит в допустимый набор (цифры и клавиша BackSpace, имеющая код 8), введённый символ заменяется на символ с кодом 0, который при выводе просто игнорируется.

Вопросы и задания

1. Что такое компоненты? Зачем они нужны?
2. Объясните, как связаны компоненты и идея инкапсуляции.
3. Что такое родительский объект? Что это значит?
4. Объясните роль свойства `Align` в размещении элементов на форме.
5. Что такое стандартный диалог? Как его использовать?
6. Назовите основное свойство флажка-выключателя. Как его использовать?
7. Расскажите о сложных свойствах на примере свойства `Font`.
8. Какой шрифт устанавливается для компонента по умолчанию?
9. Что такое метка?
10. Зачем используются функции `StrToInt` и `IntToStr`?

11. Как обрабатываются ошибки в современных программах? В чём, на ваш взгляд, преимущества и недостатки такого подхода? Приведите примеры.
12. Как при вводе можно блокировать некорректные символы?

Подготовьте сообщение

«Использование компонентов в программе на языке C#»

Задачи

1. Добавьте в программу для построения RGB-кода цвета защиту от ввода слишком больших чисел (бóльших, чем 255).
2. Разработайте программу для перевода морских милей в километры (1 миля = 1852 м).
3. Разработайте программу для решения системы двух линейных уравнений. Обратите внимание на обработку ошибок при вычислениях.
4. Разработайте программу для перевода суммы в рублях в другие валюты.
5. Разработайте программу для перевода чисел из десятичной системы в двоичную, восьмеричную и шестнадцатеричную.
6. Разработайте программу для вычисления информационного объёма рисунка по его размерам и количеству цветов в палитре.
7. Разработайте программу для вычисления информационного объема звукового файла при известных длительности звука, частоте дискретизации и глубине кодирования (числу битов на отсчёт).

§ 54
Совершенствование компонентов

Как вы видели в предыдущем параграфе, на практике нередко нужны поля ввода особого типа, с помощью которых можно вводить целые числа. Стандартный компонент TEdit разрешает вводить любые символы и представляет результат ввода как текстовое свойство Text. Поэтому для того, чтобы получить нужное нам поведение (ввод целых чисел), мы:

- добавили обработчик OnKeyPress, заблокировав ошибочные символы;
- для перевода текстовой строки в число каждый раз использовали функцию StrToInt.

Если такие поля ввода нужны часто и в разных программах, можно избавиться от этих рутинных операций. Для этого создаётся новый компонент, который обладает всеми необходимыми свойствами.

Конечно, можно создавать компонент «с нуля», но так почти никто не делает. Обычно задача сводится к тому, чтобы как-то улучшить существующий стандартный компонент, который уже есть в библиотеке Lazarus. Мы будем совершенствовать компонент TEdit (поле ввода), поэтому, согласно принципам ООП, наш компонент (назовём его TIntEdit) будет наследником класса TEdit, а класс TEdit будет соответственно базовым классом для нового класса TIntEdit:

```
type TIntEdit=class(TEdit)
...
end;
```

Изменения стандартного класса TEdit сводятся к двум пунктам:

- все некорректные символы (кроме цифр и кода клавиши BackSpace) должны блокироваться автоматически, без установки дополнительных обработчиков событий;
- компонент должен уметь сообщать числовое значение; для этого мы добавим к нему свойство Value (в переводе с англ. — значение) целого типа.

Таким образом, описание нового класса выглядит так:

```
type
  TIntEdit = class(TEdit)
  private
    function GetValue: integer;
    procedure SetValue(Val: integer);
  protected
    procedure KeyPress(var Key: Char); override;
  public
    property Value: integer read GetValue
                            write SetValue;
  end;
```

Из предыдущего материала (см. § 49) вам должно быть понятно, что для чтения введённого числового значения используется закрытый метод GetValue, а для записи — закрытый метод SetValue. Эти методы могут выглядеть так:

```
function TIntEdit.GetValue: integer;
begin
  try    Result:=StrToInt(Text);
  except Result:=0; end;
end;
procedure TIntEdit.SetValue(Val: integer);
begin
  Text:=IntToStr(Val);
end;
```

Для преобразования целых чисел из текстового формата в числовой используется функция StrToInt, а для обратного преобразования — функция IntToStr. В метод GetValue введена защита — в случае ошибки функция возвращает 0.

Для обработки вводимых символов будем использовать метод KeyPress. Этот метод не новый, он есть и у базового класса TEdit. Слово override говорит о том, что класс-наследник переопределяет этот метод базового класса TEdit. Если посмотреть в исходные тексты библиотеки Lazarus, метод KeyPress состоит из одной строчки — он вызывает обработчик события OnKeyPress, установленный пользователем. Мы переопределим этот метод так:

```
procedure TIntEdit.KeyPress(var Key: Char);
begin
  if not (Key in ['0'..'9', #8]) then Key:=#0;
  inherited;
end;
```

В первой строке происходит блокировка неверных символов, а во второй с помощью команды `inherited` вызывается перекрытый метод базового класса. Поэтому если пользователь компонента `TIntEdit` установит обработчик `OnKeyPress`, он будет успешно вызван, но уже после того, как введённый символ обработает наша процедура.

Готовый компонент лучше всего поместить в отдельный модуль, назовем его `int_edit`. В среде Lazarus для создания нового модуля нужно выбрать команду меню **Файл – Создать модуль.** Описание класса размещается в секции **interface**, а сами методы — в секции **implementation**. Кроме того, в список используемых модулей (после слова **uses**) нужно добавить модуль `StdCtrls`, в котором описан класс `TEdit`.

Создадим новый проект и сразу добавим в список используемых модулей новый модуль `int_edit`. Эта программа будет переводить целые числа из десятичной системы в шестнадцатеричную (рис. 7.21).

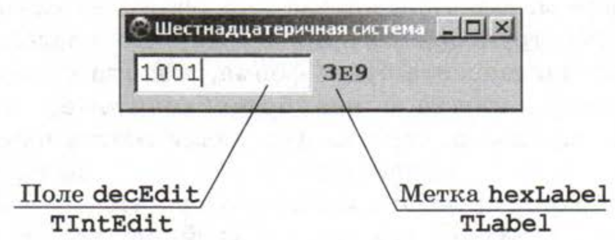


Рис. 7.21

Поместим на форму метку `hexLabel` для вывода шестнадцатеричного значения.

Теперь нужно добавить на форму новый компонент класса `TIntEdit`, но проблема состоит в том, что его нет в палитре компонентов! В этом случае компонент можно добавить при выполнении программы, и все его свойства придётся настраивать вручную, в программе.

Сначала добавим в описание формы переменную типа `TIntEdit`:

```
TForm1=class(TForm)
  ...
  decEdit: TIntEdit;
end;
```

Как вы знаете, это ещё не поле ввода, а ссылка — переменная, в которой можно хранить адрес какого-нибудь поля ввода. Создавать сам объект удобнее всего в обработчике события OnCreate формы (он будет вызван при создании формы в самом начале работы программы):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  decEdit:=TIntEdit.Create(Self);
  decEdit.Text:='100';
  decEdit.Left:=6;
  decEdit.Top:=6;
  decEdit.Width:=115;
  decEdit.Parent:=Self;
end;
```

В первой строке создаётся новый компонент, и его адрес записывается в поле decEdit. Слово Self при вызове конструктора Create означает «этот объект». Поскольку метод FormCreate — это метод формы, в данном случае этот объект — сама форма. Такой вызов конструктора говорит о том, что владельцем (*англ.* owner) нового компонента будет форма, и когда форма будет удалена из памяти, вместе с ней будет уничтожен и компонент decEdit. В следующих строках устанавливаются начальные значения для свойств компонента (Text — содержимое, Left и Top — координаты левого верхнего угла, Width — ширина).

В последней строчке мы меняем свойство Parent (в переводе с англ. — родитель), записывая в него адрес формы (Self). Это очень важно, потому что «родитель» отвечает за показ всех «дочерних объектов» на экране; если этого не сделать, то поле ввода останется невидимым.

Остаётся определить для нового компонента обработчик события OnChange: при изменении содержимого поля ввода нужно показать соответствующее шестнадцатеричное число с помощью метки hexLabel. Сначала создадим сам обработчик. Вручную добавим в описание формы заголовок процедуры:

```
procedure decEditChange(Sender: TObject);
```

а в секцию **implementation** — её текст:

```
procedure TForm1.decEditChange(Sender: TObject);
begin
  hexLabel.Caption:=IntToHex(decEdit.Value,1);
end;
```

Сначала мы запрашиваем числовое (!) значение у поля ввода, используя новое свойство Value, а затем переводим его в шестнадцатеричную систему счисления с помощью функции IntToHex. Её второй параметр — количество шестнадцатеричных цифр; если этот параметр равен 1, выбирается минимально возможное количество цифр (без лидирующих нулей).

Теперь нужно как-то подключить этот обработчик к вновь созданному компоненту. Для этого нельзя использовать Инспектор объектов, потому что в режиме разработки нашего компонента на форме нет. Но существует другой способ — в программе записать в свойство OnChange (оно наследуется от базового класса TEdit) адрес процедуры обработки события. Это нужно сделать при создании формы, добавив в конец обработчика OnCreate строку

```
decEdit.OnChange:=@decEditChange;
```

Символ @ перед именем процедуры обозначает её адрес.

Теперь программа готова и её можно запустить. Фактически мы вручную сделали все операции, которые обычно выполняет среда Lazarus, если компонент добавляется на форму из палитры компонентов. Установить компонент в палитру можно с помощью меню **Пакет**, но это выходит за рамки нашего курса.

Вопросы и задания

1. В каких случаях имеет смысл разрабатывать свои компоненты?
2. Подумайте, в чём достоинства и недостатки использования своих компонентов.
3. Почему программисты редко создают свои компоненты «с нуля»?
4. Объясните, как связаны классы компонентов TIntEdit и TEdit из примера в параграфе. Чем они различаются?
5. В каких секциях модуля нужно расположить описание нового класса и его реализацию (программный код методов)? Объясните, почему нежелательно располагать всё в одной секции.
6. Какие функции используются для преобразования числового значения в текстовое и обратно?
7. Какая функция применяется для перевода числа в шестнадцатеричную систему счисления?
8. Объясните, как работает свойство Value у компонента TIntEdit из примера в параграфе?
9. Почему в приведённом в параграфе примере для обработки вводимых символов мы не устанавливали свой обработчик OnKeyPress?
10. Что означает слово override при описании метода?
11. Как создать компонент во время выполнения программы?

12. Почему компоненты обычно создаются в обработчике `OnCreate` формы?
13. Чем различаются роли владельца компонента и его родительского объекта?
14. Почему свойства нового компонента в данном примере устанавливаются только из программы, а не в Инспекторе объектов?
15. Как установить обработчик события во время выполнения программы?
16. Почему можно использовать обработчик события `OnChange`, который не был объявлен в классе `TIntEdit`?

Подготовьте сообщение

«Создание компонентов в программе на C#»

Задачи

1. Измените рассмотренную в параграфе программу так, чтобы в самом начале метка показывала шестнадцатеричный код числа, которое записано в поле ввода.
2. Разработайте компонент, который позволяет вводить шестнадцатеричные числа.

*3. Используя дополнительные источники, разберитесь, как установить новый компонент в палитру среды Lazarus. Переделайте свою программу так, чтобы компонент добавлялся на форму из палитры.

§ 55
Модель и представление

Одна из важнейших идей технологии быстрого проектирования программ (RAD) — повторное использование написанного ранее готового кода. Чтобы облегчить решение этой задачи, было предложено использовать ещё одну *декомпозицию*: разделить **модель**, т. е. данные и методы их обработки, и **представление** — способ взаимодействия модели с пользователем (интерфейс) (рис. 7.22).

Пусть, например, данные об изменении курса доллара хранятся в виде массива; требуется искать в массиве максимальное и минимальное значения, а также строить приближённые зависимости, позволяющие прогнозировать изменение курса в ближайшем будущем. Это описание задачи на уровне модели.

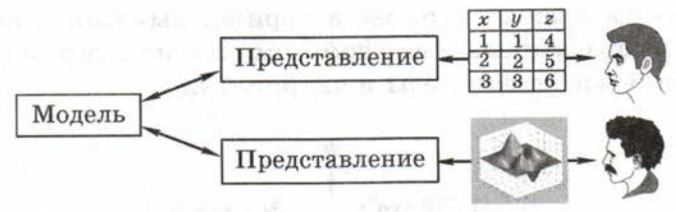


Рис. 7.22

Для пользователя эти данные могут быть представлены в различных формах: в виде таблицы, графика, диаграммы и т. п. Полученные зависимости, приближённо описывающие изменение курса, могут быть показаны в виде формулы или в виде кривой. Это уровень представления или интерфейса с пользователем.

Чем хорошо такое разделение? Его главное преимущество состоит в том, что модель не зависит от представления, поэтому одну и ту же модель можно использовать без изменений в программах, имеющих совершенно различный интерфейс.

Вычисление арифметических выражений: модель

Построим программу, которая вычисляет арифметическое выражение, записанное в символьной строке. Для простоты будем считать, что в выражении используются только:

- целые числа;
- знаки арифметических действий `+ - * /`.

Предположим, что выражение не содержит ошибок и посторонних символов.

Какова **модель** для этой задачи? По условию, данные хранятся в виде символьной строки. Обработка данных состоит в том, что нужно вычислить значение записанного в строке выражения.

Вспомните, что аналогичную задачу мы решали в § 43, где использовалась структура типа «дерево». Теперь мы применим другой способ.

Как вы знаете, при вычислении арифметического выражения последней выполняется крайняя справа операция с наименьшим приоритетом (см. § 43). Таким образом, можно сформулировать следующий алгоритм вычисления арифметического выражения, записанного в символьной строке `s`:

1. Найти в строке `s` последнюю операцию с наименьшим приоритетом (пусть номер этого символа записан в переменной `k`).

2. Используя дважды этот же алгоритм, вычислить выражения слева и справа от символа с номером k и записать результаты вычисления в переменные *n1* и *n2* (рис. 7.23).

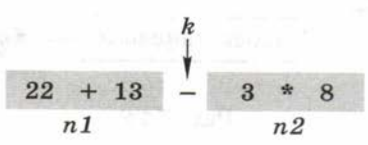


Рис. 7.23

3. Выполнить операцию, символ которой записан в *s*[*k*], с переменными *n1* и *n2*.

Обратите внимание, что в п. 2 этого алгоритма нужно решить ту же самую задачу для левой и правой частей исходного выражения. Как вы знаете, такой прием называется *рекурсией*.

Основную функцию назовём Calc (от англ. *calculate* — вычислить). Она принимает символьную строку и возвращает целое число — результат вычисления выражения, записанного в этой строке. Алгоритм её работы на псевдокоде:

```
k:=номер символа, соответствующего последней операции
если k=0 то
  знач:=перевести всю строку в число
иначе
  n1:=результат вычисления левой части
  n2:=результат вычисления правой части
  знач:=применить найденную операцию к n1 и n2
все
```

Для того чтобы найти последнюю выполняемую операцию, будем использовать функцию LastOp из § 43. Если эта функция вернула 0, то операция не найдена, т. е. вся переданная ей строка — это число (предполагается, что данные корректны).

Теперь можно написать функцию Calc:

```
function Calc ( s: string ) : integer;
var k, n1, n2: integer;
begin
  k:=LastOp (s);
  if k=0 then Calc:=StrToInt(s) {вся строка - число}
  else begin
    n1:=Calc(Copy(s, 1, k-1));  {левая часть}
```

```
    n2:=Calc(Copy(s, k+1, Length(s)-k));
                                  {правая часть}
    case s[k] of                  {выполнить операцию}
      '+': Calc:=n1+n2;
      '-': Calc:=n1-n2;
      '*': Calc:=n1*n2;
      '/': Calc:=n1 div n2;
    end;
  end;
end;
```

Обратите внимание, что функция Calc — рекурсивная, она дважды вызывает сама себя.

Функции Calc и LastOp (а также функцию Priority, которая вызывается из LastOp) удобно объединить в отдельный модуль Model (модуль модели):

```
unit Model;
interface
function Calc(s: string): integer;
implementation
uses SysUtils;
function Priority(op: char): integer;
...
function LastOp(s: string): integer;
...
function Calc(s: string): integer;
...
end.
```

Секция **interface** этого модуля содержит только заголовок функции Calc — это всё, что доступно другим модулям программы. В секции **implementation** подключается модуль SysUtils (в котором находится функция StrToInt) и записаны все функции (многоточие обозначает тело функции). Таким образом, наша модель — это функции, с помощью которых вычисляется арифметическое выражение, записанное в строке.

Вычисление арифметических выражений: представление

Теперь построим интерфейс программы. В верхней части окна будет размещён выпадающий список (компонент TComboBox), в котором пользователь вводит выражение (рис. 7.24). При нажатии на клавишу *Enter* выражение вычисляется и его результат выводится

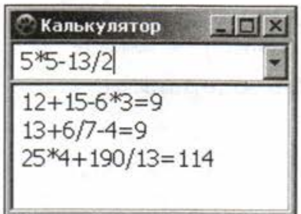


Рис. 7.24

в последней строке многострочного редактора текста (компонента TMemo). Список полезен для того, чтобы можно было вернуться к уже введённому ранее выражению и исправить его. Для этого каждое новое выражение будем добавлять в выпадающий список.

Итак, на форму нужно добавить компонент TComboBox (группа **Standard**). Чтобы прижать его к верху, установим свойство Align, равное alTop. Назовем этот компонент Input (в переводе с англ. — ввод).

Добавляем второй компонент — TMemo (группа **Standard**), устанавливаем для него выравнивание alClient (заполнить всю свободную область) и имя Answers (в переводе с англ. — ответы). Для того чтобы пользователь не мог менять поле вывода, для компонента Answers устанавливаем логическое свойство ReadOnly (только для чтения), равное True.

Логика работы программы может быть записана в виде псевдокода:

```
если нажата клавиша Enter то
  x:= значение выражения
  добавить результат вычислений в конец поля вывода
  если выражения нет в списке то
    добавить его в список
  все
все
```

Для перехвата нажатия клавиши Enter будем использовать обработчик OnKeyPress компонента Input. Клавиша Enter имеет код 13, поэтому условие «если нажата клавиша Enter» запишется в виде

```
if Key=#13 then begin
...
end;
```

Значение выражения будем вычислять с помощью функции Calc:

```
x:= Calc(Input.Text);
```

Эта функция находится в модуле Model, который нужно подключить, добавив в начало секции **implementation** команду

```
uses Model;
```

Компонент TMemo содержит массив строк, которые доступны как свойство-массив Lines. Чтобы добавить к ним новую строку (в конец массива), нужно использовать метод Add (в переводе с англ. — добавить):

```
Answers.Lines.Add (Input.Text+'='+IntToStr(x));
```

Обратите внимание, что результат вычислений переведен в символьный вид с помощью функции IntToStr.

Строки, входящие в выпадающий список, доступны как свойство-массив Items объекта Input. Метод IndexOf служит для поиска строки в списке и возвращает номер найденного элемента (нумерация начинается с нуля) или значение −1, если образец не найден. Поэтому команда добавления в список новой строки выглядит так:

```
i:=Input.Items.IndexOf(Input.Text);
if i<0 then
   Input.Items.Insert(0, Input.Text);
```

Метод Insert добавляет строку в список. На первом месте указывается позиция, в которую добавляется строка (0 — в начало списка). Приведём полностью обработчик OnKeyPress:

```
procedure TForm1.InputKeyPress(Sender: TObject;
                                    var Key: char);
var x, i: integer;
begin
  if Key=#13 then begin
     x:=Calc(Input.Text);
     Answers.Lines.Add(Input.Text+'='+IntToStr(x));
     i:=Input.Items.IndexOf(Input.Text);
     if i<0 then
        Input.Items.Insert(0, Input.Text);
  end;
end;
```

Теперь программу можно запускать и испытывать.

Итак, в этой программе мы разделили модель (данные и средства их обработки) и представление (взаимодействие модели с пользователем), которые разнесены по разным модулям. Это позволяет использовать модуль модели в любых программах, где нужно вычислять арифметические выражения.

Часто к паре «модель — представление» добавляют ещё управляющий блок (контро́ллер), который, например, обрабатывает ошибки ввода данных. Но при программировании в RAD-средах контроллер и представление, как правило, объединяются вместе — контроль данных происходит в обработчиках событий.

Вопросы и задания

1. Чем хорошо разделение программы на модель и интерфейс? Как это связано с особенностями современного программирования?
2. Что обычно относят к модели, а что — к представлению?
3. Что от чего зависит (и не зависит) в паре «модель — представление»?

4. Приведите свои примеры задач, в которых можно выделить модель и представление. Покажите, что для одной модели можно придумать много разных представлений.
5. Объясните алгоритм вычисления арифметического выражения без скобок.
6. Пусть требуется изменить программу вычисления арифметического выражения так, чтобы она обрабатывала выражения со скобками. Что нужно изменить: модель, интерфейс или и то, и другое?

Подготовьте сообщение

а) «Зачем нужны шаблоны проектирования?»

б) «Схема "Модель — представление — контроллер"»

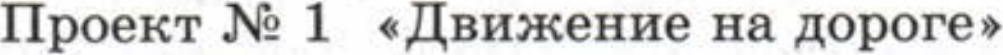
Задачи

1. Измените рассмотренную в параграфе программу так, чтобы она вычисляла выражения с вещественными числами (для перевода вещественных чисел из символьного вида в числовой используйте функцию `StrToFloat`).
2. Добавьте в рассмотренную в параграфе программу обработку ошибок. Подумайте, какие ошибки может сделать пользователь. Какие ошибки могут возникнуть при вычислениях? Как их обработать?

*3. Измените рассмотренную в параграфе программу так, чтобы она вычисляла выражения со скобками. *Подсказка*: нужно искать последнюю операцию с самым низким приоритетом, стоящую *вне* скобок.

4. Постройте программу «Калькулятор» для выполнения вычислений с целыми числами:

Практические работы к главе 7



Проект № 1 «Движение на дороге»
Работа № 62 «Скрытие внутреннего устройства объектов»
Проект № 2 «Иерархия классов (логические элементы)»
Работа № 63 «Создание формы в RAD-среде»
Работа № 64 «Использование компонентов»
Работа № 65 «Компоненты для ввода и вывода данных»
Работа № 66 «Разработка компонентов»
Проект № 3 «Модель и представление»

ЭОР к главе 7 на сайте ФЦИОР (http://fcior.edu.ru)

- Объектно-ориентированное программирование
- Основные понятия и принципы ООП
- Этапы объектно-ориентированного программирования
- Объектно-ориентированная модель программирования
- Основные принципы объектно-ориентированного программирования: инкапсуляции и полиморфизма, наследования и переопределения.

Самое важное в главе 7

- Сложность и размеры современных программ таковы, что в их разработке принимает участие множество программистов. Объектно-ориентированное программирование — это метод, позволяющий разбить задачу на части, каждая из которых в максимальной степени независима от других.

- Программа в ООП — это набор объектов, которые обмениваются сообщениями.
- Перед программированием выполняется объектно-ориентированный анализ задачи. На этом этапе выделяются взаимодействующие объекты, определяются их существенные свойства и поведение.
- Любой объект — экземпляр какого-то класса. Классом называют группу объектов, обладающих общими свойствами.
- Объекты не могут «узнать» устройство других объектов (принцип инкапсуляции). При описании класса закрытые поля и методы помещаются в секцию **`private`**, а общедоступные — в секцию **`public`**.
- Обмен данными между объектами выполняется с помощью общедоступных свойств и методов, которые составляют интерфейс объектов. Изменение внутреннего устройства объектов (реализации) не влияет на взаимодействие с другими объектами, если не меняется интерфейс.
- Как правило, классы образуют иерархию (многоуровневую структуру). Классы-потомки обладают всеми свойствами и методами классов-предков, к которым добавляются их собственные свойства и методы.
- ООП позволяет обеспечивать высокую скорость и надёжность разработки и модификации больших и сложных программ. В простых задачах применение ООП, как правило, увеличивает длину программы и замедляет её работу.
- Современные программы с графическим интерфейсом основаны на обработке событий, которые вызваны действиями пользователя и поступлением данных из других источников, и могут происходить в любой последовательности.
- Для быстрой разработки программ применяют системы визуального программирования, в которых интерфейс строится без написания программного кода. Такие системы, как правило, основаны на ООП.
- В современных программах принято разделять модель (данные и алгоритмы их обработки) и представление (способ ввода исходных значений и вывода результатов).

Глава 8
Компьютерная графика и анимация

§ 56
Основы растровой графики

Как вы знаете из курса 10 класса, растровый рисунок, как мозаика, состоит из отдельных «квадратиков»-пикселей, каждый из которых закрашен своим цветом. Цвет кодируется как набор чисел, например в модели RGB цвет пикселя задаётся тремя значениями — красной (англ. **R** — *red*), зелёной (англ. **G** — *green*) и синей (англ. **B** — *blue*) составляющими.

В этой главе вы узнаете о том, как можно обрабатывать растровые изображения с помощью современных графических редакторов.

Что такое разрешение?

Любое изображение, в конечном счёте, предназначено для просмотра. При выводе на экран или на печать оно должно иметь определённые размеры, поэтому нужно установить связь между шириной и высотой рисунка в пикселях и его размерами (в сантиметрах или других единицах длины) на экране монитора или на бумаге. Эту связь определяет разрешение, т. е. число пикселей на некотором отрезке изображения (по ширине или высоте). По традиции разрешение измеряется в пикселях на дюйм[1] (англ. **ppi** — *pixels per inch*). Чем больше разрешение, тем выше качество изображения, но тем больше места оно занимает в памяти.

Например, пусть мы хотим вывести на экран изображение размером 10 × 15 см. Каковы будут размеры рисунка в пикселях? Обычно стандартным разрешением для изображения на экране считается[2] 72 ppi или 96 ppi. При разрешении 96 ppi размеры рисунка в пикселях должны быть равны:

высота: $10 \cdot 96 / 2{,}54 \approx 378$ пикселей;
ширина: $15 \cdot 96 / 2{,}54 \approx 567$ пикселей.

---

[1] 1 дюйм = 2,54 см.

[2] Разрешение 96 ppi, например, соответствует размерам экрана 1280 × 1024 пикселя для монитора с диагональю 17 дюймов.

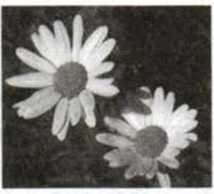
Конечно, нужно учитывать, что фактическое разрешение экрана может отличаться от 96 ppi, оно зависит от размера монитора и режима работы видеокарты (заданного в её настройках количества пикселей по ширине и высоте экрана).

Теперь предположим, что нам нужно напечатать на бумаге стандартную фотокарточку того же размера (10 × 15 см). Печатающие устройства могут обеспечить значительно более высокое разрешение, чем экран. Для получения отпечатков среднего качества (при котором уже практически незаметно, что изображение состоит из пикселей) требуется разрешение около 300 ppi, а для профессиональных работ — 600 ppi и более (до 2400 ppi). При разрешении 300 ppi размеры рисунка в пикселях будут совсем другие:

высота: 10 · 300 / 2,54 ≈ 1181 пиксель;

ширина: 15 · 300 / 2,54 ≈ 1772 пикселя.

На рисунке 8.1 для сравнения показано одно и то же изображение с разным разрешением.


300 ppi

96 ppi

24 ppi

Рис. 8.1

Заметно, что изображение с разрешением 300 ppi смотрится вполне естественно (пиксели не видны), а при разрешении 24 ppi ромашки уже с трудом угадываются в получившемся наборе пикселей.

Для иллюстрации мы будем использовать свободный растровый графический редактор GIMP (www.gimp.org), версии которого существуют для Windows, Linux и Mac OS. Другие популярные редакторы, например Adobe Photoshop, имеют аналогичные возможности.

Если в редактор GIMP загрузить некоторое изображение, с помощью меню **Изображение — Размер изображения** можно посмотреть и изменить его параметры (рис. 8.2).

Рис. 8.2

По умолчанию размер изображения задаётся в пикселях (точках растра). Чтобы увидеть размеры отпечатка (в сантиметрах или миллиметрах), в выпадающем списке справа нужно выбрать соответствующую единицу измерения.

Это диалоговое окно позволяет также менять размеры рисунка и его разрешение. Изменяя размер изображения в пикселях, мы искажаем рисунок. Например, пусть при увеличении размера нужно заменить 5 пикселей на 8. В этом случае программе приходится с помощью математических методов перестроить картинку так, чтобы изображение сохранилось как можно лучше. Алгоритм такой обработки задаётся в поле **Интерполяция**[1]. Кубическая интерполяция считается одной из лучших, однако при использовании этого метода изображение немного размывается. Чтобы сохранить чёткие границы на рисунке, в списке **Интерполяция** нужно выбрать вариант **Никакая**.

При изменении разрешения количество пикселей в рисунке остаётся тем же, поэтому никакой потери качества не происходит. Однако размеры отпечатка изменятся: если увеличить разрешение, при печати изображение уменьшится.

---

[1] Интерполяция — это восстановление промежуточных значений функции.

Цветовые модели

Как вы знаете (см. § 16 учебника для 10 класса) для кодирования цвета пикселей можно использовать разные цветовые модели, причём их выбор зависит от устройства, на которое нужно будет выводить изображение.

Если вы готовите рисунок для просмотра на экране (например, иллюстрацию для веб-сайта), нужно использовать **модель RGB**, в которой цвет пикселя задается тремя числами — красной (англ. **R** — red), зелёной (англ. **G** — *green*) и синей (англ. **B** — *blue*) составляющими. Модель RGB используется для устройств, которые *излучают* свет, например для мониторов.

Если изображение готовится для печати, переходят к **модели CMYK: C** — cyan (голубой), **M** — magenta (пурпурный), **Y** — yellow (жёлтый), **K** — key color (ключевой цвет, чёрный). Модель CMYK более удобна для описания *отражённого* света, который в этом случае видит человек.

Во всех этих моделях цвет кодируется набором чисел, и только устройство вывода определяет, какой именно цвет увидит человек. Поэтому для того, чтобы отпечаток выглядел так же, как изображение на экране, при преобразовании изображения из модели RGB в CMYK нужно использовать **цветовые профили** монитора и принтера, которые определяются с помощью специальных устройств **(калибраторов)**. Наибольшими возможностями для подготовки качественных изображений для профессиональной печати обладает программа Adobe Photoshop.

Существуют и другие цветовые модели. Наиболее интересная из них — **модель HSV**[1] (англ. *Hue* — тон, оттенок; *Saturation* — насыщенность, *Value* — величина), которая ближе всего к естественному восприятию человека. Для кодирования «абсолютного цвета», не зависящего от устройства, применяется **модель Lab** (англ. *Lightness* — светлота, *a* и *b* — параметры, определяющие тон и насыщенность цвета).

Вопросы и задания

1. Что такое разрешение? В каких единицах оно измеряется?
2. Как выбирать разрешение для вывода изображений на монитор и на печать? Почему для печати требуется более высокое разрешение?

---

[1] Или **HSB** (англ. *Hue* — тон, оттенок; *Saturation* — насыщенность, *Brightness* — яркость).

3. От чего зависит фактическое разрешение при выводе изображения на экран?
4. Что произойдёт, если изменить разрешение рисунка, не меняя его размеры в пикселях?
5. Что произойдёт, если изменить размеры рисунка в сантиметрах, сохранив разрешение?
6. Что такое интерполяция? Зачем нужны разные виды интерполяции?
7. В каких случаях используются цветовые модели RGB и CMYK? Какие ещё цветовые модели вы знаете?
8. Вспомните (см. § 16 учебника для 10 класса), что такое цветовой профиль устройства и зачем он нужен.

Подготовьте сообщение

а) «Преобразование цвета между моделями RGB и CMYK»
б) «Цветовая модель HSV»
в) «Цветовая модель Lab»

Задачи

1. Для рисунка размером 200 × 100 пикселей установлено разрешение 300 ppi. Определите его размеры в сантиметрах при выводе на экран и при печати.
2. Определите, каковы должны быть размеры рисунка в пикселях, если нужно напечатать изображение размером 297 мм на 210 мм (формат А4) с разрешением 300 ppi. Сколько мегапикселей (миллионов пикселей) должна содержать чувствительная матрица цифрового фотоаппарата, с помощью которого можно сделать такой снимок?

§ 57
Ввод изображений

Для ввода растровых изображений в компьютер чаще всего применяют цифровые фотоаппараты и сканеры.

Цифровые фотоаппараты

Самый важный элемент цифрового фотоаппарата — это **светочувствительная матрица** — интегральная микросхема, которая состоит из светочувствительных элементов — фотодиодов. Свет, поступивший на фотодиод, преобразуется в электрический сигнал, который с помощью аналого-цифрового преобразователя (АЦП) переводится в числовой код.

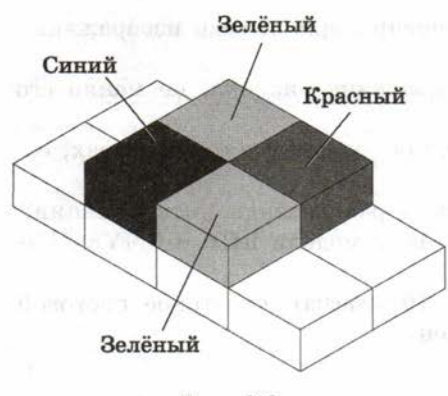


Рис. 8.3

Для того чтобы получить цветное изображение, каждый фотодиод «накрыт» светофильтром определённого цвета (чаще всего используют модель RGB — красные, зелёные и синие светофильтры). Таким образом, фотодиод измеряет яркость только одной составляющей цвета, а остальные компоненты восстанавливаются процессором фотокамеры по соседним пикселям. Во многих фотоаппаратах используют так называемый *фильтр Байера*, состоящий из 25% красных, 25% синих и 50% зелёных фильтров (рис. 8.3 и цветной рис. на форзаце). Такое соотношение объясняется тем, что человеческий глаз более чувствителен к зелёному цвету.

Цифровые фотоаппараты при съёмке сохраняют изображение на встроенной карте памяти в двух основных форматах: RAW и JPEG. Формат RAW — это необработанные данные (от *англ.* raw — «сырой», «сырьё», необработанный), т. е. коды сигналов, полученных каждым элементом чувствительной матрицы фотоаппарата. Снимок в формате RAW содержит наиболее полные данные, на каждый цветовой канал отводится 12–14 битов (в отличие от 8 битов при обычном RGB-кодировании). Существует более сотни различных RAW-форматов для разных моделей фотоаппаратов.

Если фотоаппарат сохраняет снимки в формате JPEG, «сырые» RAW-коды сразу после съёмки обрабатываются процессором с учётом настроек камеры, выбранных пользователем. Результат этой обработки затем и сохраняется на карте памяти, при этом:

- глубина кодирования уменьшается до 8 битов на канал (потеря информации!);
- изображение сжимается с потерями по алгоритму JPEG, в результате некоторые детали снимка безвозвратно теряются, хотя размер файла значительно уменьшается;
- все исходные данные уничтожаются.

Если камера была неверно настроена, полученное изображение будет низкого качества и восстановить его практически невозможно.

Поэтому профессиональные фотографы практически всегда сохраняют фотографии в формате RAW. Это даёт значительно больше возможностей для коррекции (улучшения) с помощью программного обеспечения. Например, часто удаётся увеличить контраст при плохих условиях съёмки. Для того чтобы получить изображение в одном из компьютерных форматов (JPEG, GIF, TIFF, BMP и др.), «сырые» снимки нужно обработать специальной программой, которая называется RAW-конвертором. Одна из известных программ-конверторов — LR *Adobe Photoshop Lightroom*.

Для загрузки изображений с фотоаппарата в компьютер эти два устройства соединяются кабелем через порт USB. После подключения фотоаппарат обнаруживается как новый съёмный диск. Можно также использовать устройство для чтения карт памяти — **кардридер** (при этом карту памяти нужно вынуть из фотоаппарата).

Сканирование

Сканирование — это ввод изображения в компьютер с помощью сканера. Многие растровые графические редакторы позволяют вводить сканированное изображение с помощью собственного меню. Например, в GIMP для этого нужно выбрать пункт меню **Файл – Создать – Сканер/Камера**. После этого запускается программа, обслуживающая сканер, в которой требуется задать режимы сканирования.

В первую очередь требуется определить **тип изображения**:

- чёрно-белое (двухцветное);
- полутоновое (256 оттенков серого, от чёрного до белого);
- цветное.

Второй важный параметр — это **разрешение**. Его нужно выбирать с некоторым запасом, учитывая, что отсканированное изображение потом чаще всего обрабатывается. Если изображение сканируется для вывода на экран (с разрешением 72–96 ppi) и его не нужно увеличивать, при сканировании можно выбрать разрешение 150–200 ppi. Если же вы хотите обработать рисунок и потом напечатать его на принтере, нужно выбирать разрешение не менее 300–400 ppi.

Иногда нужно отсканировать текст, чтобы потом подготовить электронную версию документа. Если документ не нужно редактировать и можно сохранить его в виде рисунка, достаточно установить разрешение 150–200 ppi. Если же требуется распознать

текст документа с помощью специальной программы (типа ABBYY FineReader или CuneiForm), нужно выбирать более высокое разрешение — не менее 300 ppi.

Кадрирование

После сканирования полученное изображение, как правило, нужно **кадрировать**, т. е. выбрать нужные границы изображения, обрезать лишнее. Рассмотрим случай, когда фотография была положена неровно, так что её нужно сначала повернуть, а потом кадрировать (рис. 8.4).

Рис. 8.4

Сначала выполним поворот так, чтобы линия горизонта была строго горизонтальна. Для этого нужно загрузить рисунок в GIMP, выбрать инструмент **Вращение** и щёлкнуть на рисунке. Появится окно, в котором можно установить нужный угол поворота (рис. 8.5).

Рис. 8.5

При изменении угла в основном окне виден результат применения этой операции. Можно также вращать изображение, схватив его мышью.

Теперь выполняем кадрирование[1]. Выберем инструмент **Кадрирование** и выделим прямоугольную область, оставив только нужную часть рисунка. Углы выделенной области можно перетаскивать мышью. При нажатии на клавишу Enter поля будут обрезаны.

Если угол поворота не кратен 90 градусам, растровое изображение искажается. Для построения нового рисунка программа использует математические методы интерполяции, так же как и при изменении его размеров.

Вопросы и задания

1. Какие два основных способа ввода растровых изображений вы знаете?
2. Как вы думаете, когда лучше использовать сканирование, а когда — фотографирование? Приведите примеры.

3. В каких форматах обычно сохраняют снимки цифровые фотоаппараты?
4. Что такое формат RAW? В чём его преимущества и недостатки по сравнению с форматом JPEG?
5. Что такое RAW-конвертер?
6. Как можно загрузить в компьютер изображения, записанные на карте памяти цифрового фотоаппарата?
7. Что такое сканирование?
8. Какие параметры важны при сканировании?
9. Что такое кадрирование? Зачем оно нужно?

Подготовьте сообщение

а) «Форматы RAW: за и против»
б) «Выбор параметров сканирования»

§ 58 Коррекция фотографий

Качество многих фотографий можно значительно улучшить, устранив дефекты, возникшие из-за неправильной настройки фотокамеры. Кроме того, можно исправить некоторые искажения, которые вносит сам фотоаппарат.

---

[1] В Adobe Photoshop кадрирование и поворот выполняются сразу (рамку кадра можно вращать).

Поскольку растровый рисунок хранится как набор чисел, кодирующих цвета пикселей, коррекция фотографий сводится к математической обработке этих данных с помощью специально разработанных алгоритмов.

Исправление перспективы

Оптическая система фотоаппаратов часто даёт искажения, в результате которых вертикальные линии (например, стены домов) становятся наклонными (рис. 8.6, *а*).

а

б

Рис. 8.6

Этот дефект легко исправляется в современных графических редакторах (рис. 8.6, *б*). В GIMP нужно выделить всё изображение (клавиши Ctrl+A или меню **Выделение – Все**) и выбрать инструмент **Перспектива**. После этого углы рисунка, выделенные квадратами, надо перетащить так, чтобы выровнять вертикальные линии.

Гистограмма

Очень полезную информацию для оценки изображения даёт **гистограмма** — график специального вида, который показывает распределение пикселей по яркости (рис. 8.7). Высота вертикальных отрезков определяет количество пикселей, имеющих одинаковую яркость, от самых тёмных (слева) до самых светлых (справа).

Чтобы увидеть гистограмму загруженного изображения, в редакторе GIMP нужно выбрать пункт меню **Окна – Прикрепляющиеся диалоги – Гистограмма**.

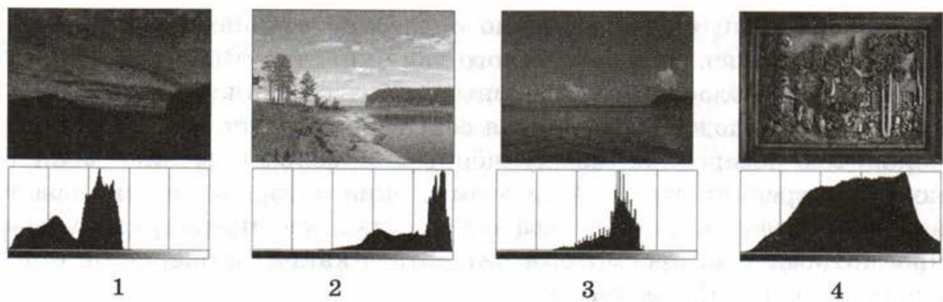


Рис. 8.7

По приведённым на рис. 8.7 гистограммам сразу видно, что:

- фото 1 слишком тёмное, потому что все пиксели сосредоточены в области тёмных тонов (слева);
- фото 2 слишком светлое (нет тёмных тонов);
- фото 3 малоконтрастное (есть средние тона, но нет тёмных и светлых);
- фото 4 содержит пиксели всех уровней яркости[1].

Изображениям 1–3 не хватает контраста. Чтобы их улучшить, нужно «растянуть» гистограмму так, чтобы она занимала весь диапазон тонов, от чёрного до белого. Для этого в GIMP используется меню **Цвет – Уровни**[2]. В появившемся окне нужно отрегулировать положение чёрного, серого и белого движков под гистограммой (рис. 8.8).

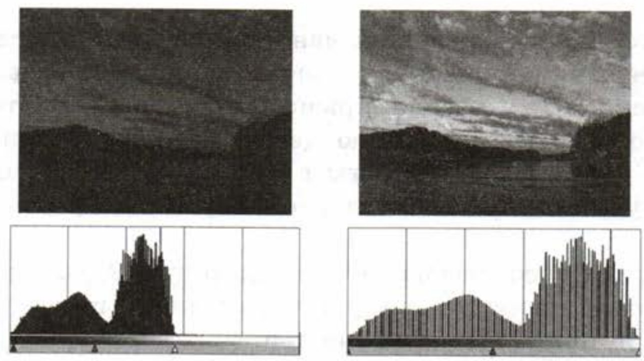


Рис. 8.8

[1] На этой фотографии — картина «Пристань» художника К. А. Гоголева, вырезанная из дерева.

[2] В Adobe Photoshop — меню **Изображение – Коррекция – Уровни.**

Все пиксели слева от чёрного движка становятся чёрными. Те, что оказались справа от белого движка, станут белыми. Таким образом, вся область между чёрным и белым движками растянется на весь диапазон. Передвигая серый движок (по умолчанию он находится посередине между чёрным и белым), можно менять контраст средних тонов. При этом в окне изображения мы сразу видим результат (такой эффект называется **предварительным просмотром**, т. е. просмотром результата какой-то операции до её окончательного применения).

После коррекции уровней внешний вид фотографии значительно улучшается, она становится более контрастной, содержит достаточно большое число и светлых, и тёмных пикселей. Однако гистограмма после коррекции не сплошная, а состоит из отдельных полосок (см. рис. 8.8). Это значит, что пикселей с некоторыми значениями яркости нет совсем (подумайте почему).

Заметим, что над гистограммой в окне **Уровни** расположен выпадающий список, в котором можно выбрать для коррекции один цветовой канал: красный, синий или зелёный. Это даёт возможность настраивать каждый канал отдельно.

Для выравнивания уровней можно также использовать окно регулировки яркости и контраста (меню **Цвет – Яркость – Контраст**). Более сложную тоновую коррекцию выполняют с помощью кривых (**Цвет – Кривые**).

Коррекция цвета

В некоторых изображениях явно видно, что какой-то оттенок явно сильнее, чем нужно, и из-за этого листва деревьев может оказаться синей, а красная крыша — зелёной. В этом случае можно использовать коррекцию цвета, используя свои знания о том, какие цвета имеют объекты в действительности. В программе GIMP нужно выбрать пункт верхнего меню **Цвет – Цветовой баланс**.

С помощью окна, изображённого на рис. 8.9, можно отдельно корректировать цвета тёмных участков (теней), пикселей средней яркости (полутонов) и светлых частей.

Обратите внимание, что невозможно скорректировать один какой-то цвет, оставив все остальные без изменения. Предположим, что для какого-то пикселя мы уменьшили красную составляющую (в модели RGB). Это автоматически означает, что увеличится относительная доля зелёной и синей составляющих, т. е.

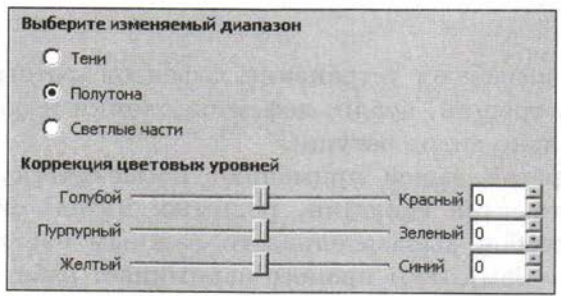


Рис. 8.9

усилится голубой канал (англ. *cyan*). Пары красный–голубой, зелёный–пурпурный (фиолетовый) и синий–жёлтый — это так называемые **дополнительные цвета** (увеличение одного из них вызывает уменьшение парного).

Иногда нужно превратить цветное изображение в чёрно-белое (серое), например для публикации в книге. В этом случае информация о цвете будет потеряна и останется только тон (яркость). В редакторе GIMP для этого используется пункт меню **Цвет – Обесцвечивание.**

Эта операция не так проста, поэтому программа предлагает на выбор несколько методов построения «серого» изображения (рис. 8.10).

Обесцвечивание
Вставленный слой-169 (color.xcf)
Основа оттенков серого:
Светлота
Светимость
Среднее
Предварительный просмотр
Справка | Сбросить | OK | Отменить

Рис. 8.10

Например, при выборе варианта **Светимость** тон пикселя вычисляется по формуле $0{,}21\ R + 0{,}72\ G + 0{,}07\ B$, где R, G и B — значения яркостей красного, зелёного и синего каналов. В этой формуле учитывается, что человеческий глаз наиболее чувствителен к зелёному цвету. Нужно выбирать такой метод обесцвечивания, при котором полученное «серое» изображение выглядит, на ваш взгляд, лучше всего.

Ретушь

Ретушью называют устранение дефектов фотографий — пятен, царапин, трещин, вуали, дефектов съёмки и обработки. Различают несколько видов ретуши.

Для портретов людей применяют **косметическую ретушь** — устранение дефектов (морщин, родимых пятен, складок кожи) и придание особой выразительности важным частям лица (глазам, бровям, губам). Этот процесс напоминает работу художника-визажиста или гримёра.

При обработке старых фотографий надо восстановить первоначальный вид изображения, внося как можно меньше изменений, поэтому говорят о **реставрации.**

Для устранения художественных дефектов применяют **композиционную ретушь** — кадрирование, удаление лишних элементов, добавление элементов, изменение фона, регулировку освещения.

Ретушь, которой раньше занимались специалисты-ретушёры, — сложное и утомительно занятие. Компьютерная ретушь обладает очень большими возможностями и абсолютно безопасна для оригинального изображения. На рисунке 8.11, *а* показана исходная фотография: она малоконтрастная, на лице ребёнка есть пятна, в левом верхнем углу видна граница фотокарточки. С помощью приёмов ретуширования можно исправить её (рис. 8.11, *б*).

а

б

Рис. 8.11

Сначала убирают дефект всего изображения — недостаток контраста, это можно сделать с помощью настройки уровней (см. выше).

Для исправления локальных (местных) дефектов используют специальные инструменты редактора GIMP: **Штамп**, **Осветление/Затемнение**, **Лечащая кисть**, **Размывание/Резкость**.

Инструмент **Штамп** переносит изображение с одного участка на другой. Область-источник задаётся щелчком мышью при нажатой клавише Ctrl[1]. После этого мы как бы рисуем кистью, копируя образец в другое место. В области параметров инструмента (рис. 8.12) можно настроить свойства инструмента, например непрозрачность, форму кисти и её размер (движок **Масштаб**) и др.

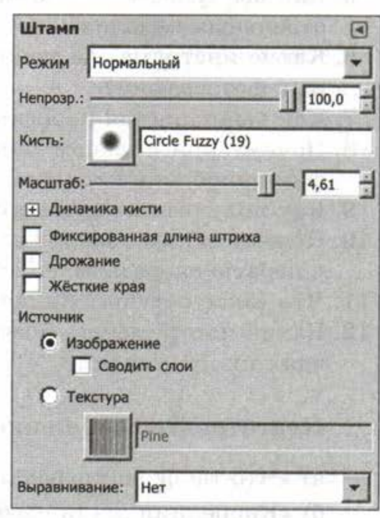


Рис. 8.12

Инструмент **Лечащая кисть** работает практически так же, как и Штамп, но при переносе изображения учитывает окружающие пиксели, поэтому «впечатывание» получается менее заметным (но изображение немного размывается).

При использовании инструмента **Осветление/Затемнение** кисть при рисовании осветляет или затемняет (в зависимости от режима, выбранного в окне параметров) отдельные области.

Инструмент **Размывание/Резкость** позволяет размыть или повысить резкость участков изображения. Размытие может понадобиться, например, для того, чтобы предметы заднего плана не отвлекали внимание зрителей от главного объекта.

Повышение резкости — это некоторая операция с цветом пикселей, которая позволяет чётче выделить границы. Нужно понимать, что значительно увеличить резкость фотографии не удастся, потому что изображение не содержит нужной информации, и программа только пытается её восстановить математическими методами.

Вопросы и задания

1. Почему на многих фотографиях возникает искажение перспективы? Как его убрать?
2. Что такое гистограмма и что она показывает?

[1] В Adobe Photoshop для этого служит клавиша Alt.

3. Как выполняется коррекция уровней?
4. Почему после коррекции уровней гистограмма не сплошная, а состоит из отдельных полосок?
5. Как вы думаете, зачем может понадобиться редактирование уровней отдельных каналов?
6. Какие инструменты можно использовать для коррекции неконтрастных фотографий?
7. Как выполняется коррекция цвета?
8. Почему при сильном уменьшении яркости красного цвета фотография приобретает голубой оттенок?
9. Как получить из цветного изображения чёрно-белое?
10. Почему многие алгоритмы обесцвечивания изображений учитывают в первую очередь зелёный цветовой канал?
11. Что такое ретушь? Какие виды ретуши вы знаете?
12. Какие инструменты можно использовать для ретуши? В каких случаях их применяют?

Подготовьте сообщение

а) «Что такое гистограмма?»
б) «Коррекция цвета изображения»
в) «Использование кривых для коррекции фотографий»
г) «Алгоритмы обесцвечивания изображений»

Задачи

1. Проверьте, как влияет на гистограмму изменение яркости и контраста.
2. Попробуйте обесцветить какое-нибудь изображение несколькими методами и выберите лучший из них (на ваш взгляд).
3. Отсканируйте какую-нибудь старую фотографию с дефектами и отреставрируйте её.

§ 59
Работа с областями

Выделение областей

Часто нужно работать не с целым изображением, а только с некоторой областью, так что все пиксели вне этой области не должны изменяться. Для выделения областей в редакторе GIMP используют несколько инструментов.

Простейшие инструменты выделения — **Прямоугольник** и **Эллипс.** Если при их использовании удерживать нажатой клавишу Shift, будет выделен соответственно квадрат или окружность, а при нажатой клавише Alt происходит выделение от центра, а не с угла области.

С помощью инструмента **Лассо** выделяют область, ограниченную ломаной линией (щелчками мышью в узлах ломаной) или вообще произвольную область (обводя её при нажатой левой кнопке мыши).

Инструменты **Волшебная палочка** и **Выделение по цвету** применяют для выделения областей одного цвета (или близких цветов, если в параметрах инструмента установлен ненулевой порог чувствительности). Выделение начинается в той точке, где выполнен щелчок мышью. Различие между двумя инструментами состоит в том, что **Волшебная палочка** выделяет только связанную область около начальной точки, а **Выделение по цвету** — одноцветные пиксели по всему изображению.

Инструмент **Умные ножницы** используется для выделения областей с чёткими, но неровными границами. Пользователь щёлкает мышью в опорных точках, между которыми программа дорисовывает линию выделения, стараясь следовать границе между двумя цветами. Когда выделение закончено, нужно щёлкнуть мышью в середине области.

Если в момент начала выделения области была нажата клавиша Shift, новая область добавляется к уже выделенной, а при нажатой клавише Ctrl[1] — вычитается из выделенной ранее. С помощью такого приёма можно строить сложные области.

Любой пиксель может быть выделен частично, например на 25%. Это значит, что для такого пикселя эффект, применённый к области (например, заливка) будет ослаблен в 4 раза. Частичное выделение получается при использовании двух параметров: сглаживания (англ. *antialiasing*) и растушёвки (они задаются в окне параметров выбранного инструмента). Выделим три одинаковых круга тремя способами (рис. 8.13):

1) без сглаживания и растушёвки;
2) со сглаживанием;
3) с растушёвкой.

Каждую из этих областей зальём чёрным цветом (пункт меню **Правка – Залить цветом переднего плана**).

---
[1] В редакторе Adobe PhotoShop для этой цели используется клавиша Alt.

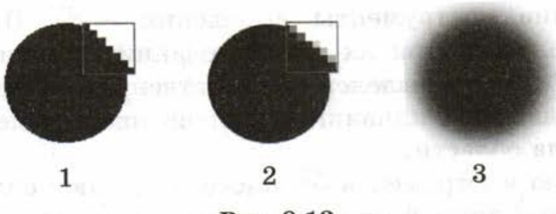


Рис. 8.13

В первом случае видим резкую границу в виде «лесенки»: пиксели или выделены на 100% (они стали чёрными), или вообще не выделены (остались белыми). Во втором случае граница сглаживается за счёт частично выделенных (серых) пикселей. При использовании растушёвки (случай 3) граница размыта.

В меню **Выделение** собраны команды для работы с областью (увеличение, уменьшение, растушёвка и т. д.).

Быстрая маска

Маска — это «накладка», которая скрывает часть объекта. В графических редакторах маска позволяет защитить от изменений некоторые части изображения. Например, когда мы выделяем область, создаётся маска, которая защищает всё, что не выделено.

В простейшем случае каждый пиксель изображения может быть полностью выделен (открыт для изменений) или невыделен (защищён). Кроме того, пиксель может быть частично выделен — есть 256 ступеней выделения, которые кодируются числами от 0 (чёрный цвет, защищенный пиксель) до 255 (белый цвет, полностью выделен, открыт). Таким образом, маска может быть закодирована как чёрно-белое полутоновое изображение.

При выделении сложных областей, которые нужно не раз редактировать, удобно использовать режим **Быстрая маска**. В этом режиме маску можно редактировать с помощью инструментов рисования. Для перехода в режим быстрой маски в редакторе GIMP используют клавиши Shift+Q. Вся закрытая (невыделенная) часть рисунка заливается полупрозрачным красным цветом, а выделенная полностью прозрачна. Рисуя (например, карандашом или кистью) чёрным цветом, мы скрываем область (удаляем из выделения), а при рисовании белым цветом открываем её. Если выбрана кисть с мягкими границами, некоторые пиксели будут выделены частично. В режиме быстрой маски часто исполь-

зуется инструмент **Градиент**, с помощью которого можно получить плавный переход от полного выделения к невыделенной части (маска плавно меняется от белого до чёрного цвета).

Исправление «эффекта красных глаз»

При фотосъемке со вспышкой нередко возникает так называемый «эффект красных глаз» — лучи вспышки отражаются от глазного дна глаз человека (оно имеет красный цвет) и попадают в объектив фотокамеры. Это особенно заметно при съёмке в темном помещении, когда зрачки глаз расширены. На самом деле, без вспышки мы видим тёмный зрачок, который и нужно восстановить.

Существуют различные методы устранения «эффекта красных глаз». Один из них — обесцвечивание (до серого цвета) и затем тонирование, при котором мы как бы смотрим на чёрно-белый рисунок через прозрачное цветное стекло. Цвет этого «стекла» в программе GIMP задаётся с помощью меню **Цвет – Тонирование**: в диалоговом окне можно с помощью ползунков выбрать оттенок **(Тон)**, насыщенность и светлоту (яркость) цвета.

На рисунке 8.14 показан один из вариантов установки движков, при котором выделенный зрачок становится тёмным.

Тонирование изображения
Фон-173 (red-eyes.jpg)

Профили:

Выбрать цвет

Тон: 356

Насыщенность: 6

Освещенность: -24

☑ Предварительный просмотр

Справка | Сбросить | OK | Отменить

Рис. 8.14

Кроме того, можно использовать фильтры — алгоритмы автоматической обработки изображений (см. следующий параграф). Например, в редакторе GIMP в меню **Фильтры** есть фильтр **Улучшение – Удалить эффект красных глаз**, который автоматически выполняет тонирование красных участков в выделенной области.

Вопросы и задания

1. Зачем нужно выделять области? Какие инструменты для этого используют?
2. Какие способы выделения и редактирования сложных областей вы знаете?
3. Что такое частичное выделение пикселя?
4. Что такое сглаживание? В каких случаях без сглаживания получается граница области в виде «лесенки», а в каких — нет?
5. Что такое растушёвка? Чем она отличается от сглаживания?
6. Подумайте, для каких целей можно использовать сильную растушёвку области выделения.
7. Что такое быстрая маска и зачем она нужна?

8. Объясните, почему возникает «эффект красных глаз» и как его исправить.

Задача

Попробуйте исправить «эффект красных глаз» на какой-нибудь фотографии.

§ 60 Фильтры

Что такое фильтры?

Фильтр — это алгоритм автоматической обработки пикселей изображения, который применяется ко всему изображению или к выделенной области.

Фильтры используют некоторые математические преобразования (иногда достаточно сложные), в результате которых цвет (код) каждого пикселя изменяется с учётом кодов соседних пикселей. Большинство фильтров настраивается с помощью диалоговых окон, в которых можно изменять параметры и сразу видеть получаемый результат.

Если эффект оказался слишком слабым и действие фильтра надо повторить, достаточно выбрать первый сверху пункт меню **Фильтр** (в нём будет название последнего примененного фильтра) или нажать клавиши Ctrl+F.

Наоборот, если действие фильтра надо ослабить, можно выбрать пункт меню **Правка – Ослабить.** В этом случае итоговое изображение строится как результат наложения отфильтрованного рисунка (с заданным уровнем непрозрачности) и исходного. Степень непрозрачности устанавливается в диалоговом окне.

Фильтры можно (очень условно) разделить на 2 группы: фильтры для коррекции изображений и фильтры для художественной обработки.

Современные графические редакторы не только содержат большой набор фильтров, но и позволяют пользователю создавать и подключать свои собственные фильтры. Их также называют **плагинами** (от *англ.* plug-in — независимый программный модуль, подключаемый к программе). Плагины для редактора GIMP обычно пишутся на языках Sript-Fu или Python.

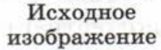
Фильтры для коррекции изображений

Нередко изображение получается нечётким (размытым) из-за того, что автофокус фотоаппарата неверно определил объект, интересующий фотографа, или в момент съемки камера немного сдвинулась. Для того чтобы повысить резкость, применяют специальные фильтры из групп **Улучшение: Нерезкая маска** и **Повышение резкости**. При этом нужно понимать, что эти фильтры не позволяют восстановить потерянную информацию, поэтому в результате их применения мелкие детали не появятся. Во многих случаях результат можно значительно улучшить, однако при сильном размытии качественное изображение получить не удастся.

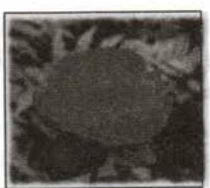
Достаточно часто применяются и фильтры группы **Размывание.** Они выполняют обратное действие — снижают резкость. При обработке фотографий это может понадобиться, например, для того, чтобы не отвлекать внимание зрителя на второстепенные детали заднего плана, оставив чётким только центральный объект. Из этой группы наиболее популярны фильтры **Гауссово размывание** (равномерное) и **Размывание движением** (позволяет создать эффект движения в определённом направлении).

Художественные фильтры

Художественные фильтры предназначены для имитации каких-то эффектов. На рисунке 8.15 показана фотография цветка и результат применения к нему нескольких фильтров редактора GIMP.


Исходное изображение

Фильтр «Кубизм»


Фильтр «Старое фото»

Фильтр «Барельеф»

Рис. 8.15

Изучать многочисленные художественные фильтры лучше всего на практике, экспериментируя с их настройками.

Вопросы и задания

1. Что такое фильтр? Как можно применить фильтр к части изображения?
2. Как усилить или ослабить действие фильтра?
3. Можно ли подключить к графическому редактору GIMP свой фильтр?

§ 61 Многослойные изображения

Зачем нужны слои?

Пусть нам нужно поместить нарисованного человечка на некоторый фон так, как показано на рис. 8.16. Скорее всего, сделать рисунок с первого раза не удастся, и его придётся не раз переделывать, чтобы получился хороший результат.

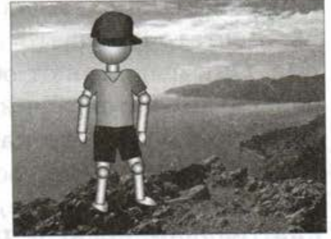


Рис. 8.16

Как вы знаете, растровый рисунок — это просто множество пикселей, каждый из которых имеет свой цвет, независимый от других. Представим себе, что после того, как человечек был нарисован, мы захотели передвинуть его в другое место. К сожалению, сделать это весьма непросто, потому что при рисовании мы меняем цвет пикселей фона и таким образом уничтожаем существующую информацию, которую потом невозможно восстановить.

Такая же проблема возникает при добавлении надписей на рисунок — текст надписи очень сложно изменить, если изображение под ней испорчено.

Как же избежать необратимых изменений? Человек видит в рисунке не пиксели, а знакомые ему объекты: скалы, берег, воду, тело человечка, футболку, шорты, кепку. Поэтому нужно попытаться каждый объект рисовать отдельно, так, чтобы его можно было изменять независимо от других.

Представим себе, что над фоновым рисунком расположено несколько стёкол, на каждом из которых нанесено изображение какого-то объекта (рис. 8.17).

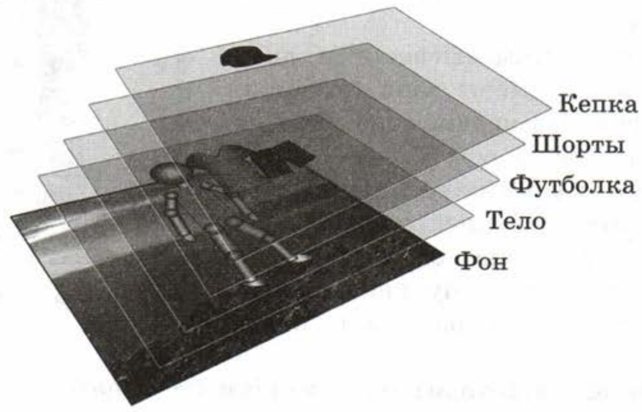


Рис. 8.17

Фактически мы разбили весь рисунок на отдельные **слои** (англ. *layers*), каждый из которых можно изменять и перемещать независимо от других. Однако если посмотреть на эту стопку сверху, мы увидим полный рисунок. Через прозрачные области верхних слоев видны изображения на нижних слоях. Этот принцип широко используется в графических редакторах, такие изображения называются **многослойными**. Применяя многослойные изображения, можно, например, составить портрет человека по описанию («фоторобот») или «примерить» ему новую одежду или причёску.

Не все форматы графических файлов поддерживают слои. Например, популярные форматы BMP, JPEG, GIF и PNG могут хранить только однослойные («плоские») рисунки. Для записи многослойных изображений чаще всего используют форматы PSD (редак-

тор Adobe Photoshop) или XCF (редактор GIMP). Нужно учитывать, что при этом в файле фактически хранится несколько отдельных изображений, поэтому его объём значительно возрастает.

Работа со слоями

Для работы со слоями в редакторе GIMP предназначено специальное окно **Слои** (рис. 8.18). Если этого окна нет на экране, вызвать его можно с помощью меню **Окна – Прикрепляющиеся диалоги – Слои** или комбинации клавиш Ctrl+L.

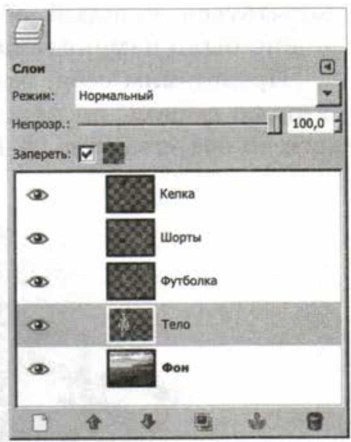


Рис. 8.18

Каждый слой имеет свое имя, двойной щелчок на имени слоя позволяет изменить его. Прозрачные области залиты клетчатым узором.

Обычно самый нижний слой — фоновый, он полностью непрозрачен. При необходимости можно обойтись и без фона, например если нужно получить изображение с прозрачными областями.

Рисование происходит на активном (текущем) слое, который выделен в списке слоёв. Остальные слои при этом не изменяются. Выбрать активный слой можно щелчком мышью в списке слоев.

Кнопки и позволяют переместить активный слой выше или ниже по списку (изменить порядок слоёв).

Если установить флажок **Запереть**, все прозрачные области активного слоя будут сохранены (рисовать можно только там, где уже что-то нарисовано).

Для того чтобы переместить всё изображение слоя, применяют инструмент **Перемещение**. В зависимости от настроек можно перемещать активный слой или слой, выбранный щелчком мышью на поле рисунка.

С помощью кнопки можно создать новый пустой слой выше активного, а кнопка создаёт копию активного слоя (например, чтобы сохранить исходное изображение при экспериментах).

Слой можно временно отключить, щёлкнув на значке в соответствующей строке. Щёлкнув в этом же месте повторно, мы

вновь сделаем слой видимым. Для того чтобы совсем удалить текущий слой, нужно щёлкнуть на кнопке или перетащить слой из списка на эту кнопку.

Слой можно сделать полупрозрачным: для этого движок **Непрозрачность** сдвигается влево. Справа от этого движка показывается непрозрачность слоя в процентах.

Список **Режим** определяет, как слой влияет на изображение, полученное с нижних слоёв. Пусть, для простоты, рисунок содержит два слоя. Тогда цвет пикселя итогового изображения вычисляется по некоторому алгоритму на основе цветов соответствующих пикселей этих двух слоёв. Этот алгоритм и определяется в списке **Режим.** По умолчанию установлен режим **Нормальный** — это значит, что изображение верхнего слоя полностью перекрывает нижний (с учётом прозрачности). Другие режимы позволяют, например, перекрывать только тёмные или только светлые области, затемнять или осветлять рисунок, находить «разность» (различие двух рисунков).

Иногда нужно связать несколько слоёв так, чтобы они перемещались вместе. Для этого напротив каждого слоя нужно щёлкнуть мышью между значком и уменьшенным изображением слоя. В этом месте появится изображение участка цепи (рис. 8.19).

Рис. 8.19

Слои можно объединять, т. е. делать из двух или нескольких слоёв новый слой. Нужно учитывать, что, если изображения на этих слоях перекрываются, разделить их будет практически невозможно. Существуют три варианта объединения:

- объединение текущего слоя с предыдущим (нижележащим);
- объединение всех видимых слоёв;
- сведение изображения (объединение всех слоёв в один фоновый слой).

В редакторе GIMP эти операции выполняются с помощью контекстного меню слоя, которое появляется при щелчке правой кнопкой мыши на нужном слое в окне **Слои**.

Текстовые слои

В простых графических редакторах текст, размещённый на поле рисунка, «встраивается» в изображение и сразу становится

набором пикселей. Такой текст нельзя редактировать, перемещать и т. п. В более совершенных программах надписи хранятся на отдельных слоях. Большинство используемых сейчас компьютерных шрифтов — *векторные*, в них буквы задаются узловыми точками и соединяющими их отрезками или кривыми. Это позволяет многократно изменять содержание и оформление текста (в том числе гарнитуру и размер шрифта), не теряя качество изображения. Например, можно легко исправить опечатку, замеченную не сразу.

В редакторе GIMP инструмент **Текст** обозначается кнопкой **A**. После его выбора нужно щёлкнуть мышью в том месте, где будет левый верхний угол надписи и ввести текст в появившемся окне (рис. 8.20). Свойства текста настраиваются на панели свойств инструмента в нижней части Панели инструментов.

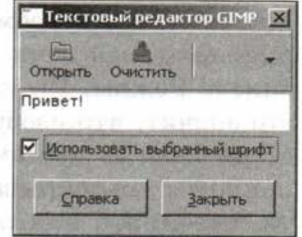


Рис. 8.20

В окне **Слои** появляется новый слой с текстом, причём вместо уменьшенного изображения содержимого вы увидите значок T. Это означает, что текст в этом слое хранится в векторном формате. Можно превратить текстовый слой в обычный (растровый), выбрав в меню пункт **Слои – Удалить текстовую информацию.** После этого текст будет редактироваться только как точечный рисунок.

Маска слоя

При создании сложных изображений желаемый результат практически никогда не получается с первого раза. Поэтому важно применять по возможности неразрушающие методы обработки, при которых информация не теряется и всегда можно вернуться к исходным данным. Один из таких приёмов — маска слоя, которая позволяет сделать часть изображения на слое невидимой (или полупрозрачной), ничего не удаляя.

Маска слоя — это полутоновое («серое») изображение, связанное с данным слоем. Чёрные области в маске закрывают рисунок на слое, а белые открывают. Серые тона — это частично открытые (полупрозрачные) области. Фактически маска слоя позволяет установить разную прозрачность для разных участков слоя. При этом рисунок на слое полностью сохраняется, что позволяет вернуться к исходному варианту в случае неудачи.

Чтобы добавить или удалить маску слоя, используют команду **Добавить маску слоя** из контекстное меню окна **Слои.** Если у слоя есть маска, в списке слоёв появляется второй значок: ▒█. Если выделен левый значок, вы меняете изображение слоя, а если правый — маску слоя. При редактировании маски используются только оттенки серого цвета. Чтобы увидеть маску, в контекстном меню слоя нужно выбрать пункт **Показать маску.** Действие маски можно временно отключить с помощью команды **Скрыть маску** из контекстного меню.

Вопросы и задания

1. Что такое «неразрушающие методы обработки»?
2. Зачем используют слои? Что это даёт?
3. Какие форматы файлов используют для хранения многослойных изображений?
4. Какие операции можно выполнять со слоями?
5. Что такое фоновый слой?
6. Как обозначаются прозрачные области слоя?
7. Какие свойства слоя определяются с помощью выпадающего списка **Режим**?
8. Каким образом можно объединить слои?
9. Какие особенности имеют текстовые слои?
10. Что такое маска слоя? Зачем она нужна?

§ 62
Каналы

Цветовые каналы

При RGB-кодировании с глубиной 24 бита на пиксель цвет каждого пикселя хранится как набор из трёх чисел (от 0 до 255), обозначающих яркости красной, зелёной и синей составляющих. Возьмём одну из трех цветовых составляющих (например, красную) и построим новое изображение, в котором каждому пикселю соответствует код от 0 до 255. Будем считать, что 0 обозначает чёрный цвет, 255 — белый, а промежуточные значения — оттенки серого цвета. Тогда получается, что мы построили полутоновое («серое») изображение, которое совпадает по размерам с исходным рисунком и показывает «вклад» выбранного цвета. Такое вспомогательное изображение называют каналом.

Канал — это чёрно-белое полутоновое изображение, которое показывает степень влияния какого-то эффекта на изображение.

На рисунке 8.21, *а* (и цветном рис. на форзаце) изображён светофор со всеми зажжёнными лампами, а на рис. 8.21, *б–г* — цветовые каналы модели RGB, показывающие влияние красного, зелёного и синего цветов.

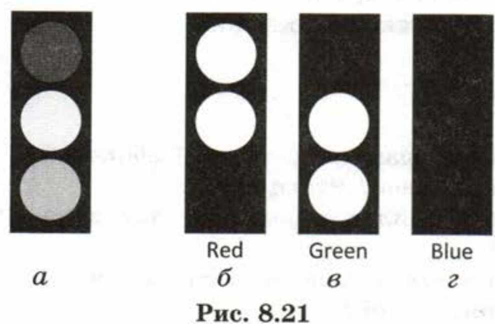


Рис. 8.21

Очевидно, что для области красного цвета будет активным только красный канал, в котором эта область закрашена белым цветом. Для области зелёного цвета также «открыт» только один канал — зелёный. Жёлтый цвет получается «сложением» красного и зелёного, поэтому в обоих этих каналах жёлтой области соответствует белый цвет. Синего цвета на рисунке нет вообще, поэтому синий канал залит чёрным цветом.

В редакторе GIMP цветовые каналы модели RGB можно увидеть в окне **Каналы** — рис. 8.22 (меню **Окна – Прикрепляющиеся диалоги – Каналы**). Любой канал можно отключить, щёлкнув на значке 👁 в соответствующей строке (повторный щелчок в этом месте снова включает канал).

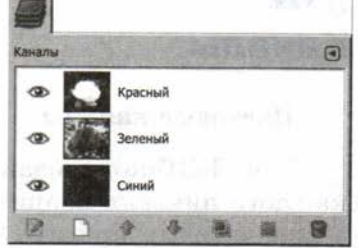


Рис. 8.22

По умолчанию выделены (синим фоном) все три цветовых канала, т. е. все инструменты рисования и коррекции применяются к ним одновременно. При желании можно выделить какой-то один канал и редактировать только его.

У изображений с прозрачными и полупрозрачными областями появляется четвёртый канал — так называемый **альфа-канал**, определяющий прозрачность (рис. 8.23). Чёрные пиксели альфа-

канала определяют прозрачное изображение, белые — полностью непрозрачное. Чтобы построить такое изображение, нужно выбрать команду **Добавить альфа-канал** из контекстного меню окна **Слои.** При этом фоновый слой превращается в обычный слой, на нём теперь можно делать прозрачные участки (например, удалив выделенную область или используя инструмент **Ластик**).

Рис. 8.23

Редактор GIMP не позволяет редактировать изображение в других цветовых моделях (например, в модели CMYK, которая используется при печати). Однако с помощью меню **Цвет – Составляющие – Разобрать** можно построить новое многослойное полутоновое изображение, в котором каждый слой будет представлять собой канал выбранной цветовой модели. После этого можно отдельно редактировать каждый канал и затем обратно «собрать» полное изображение с помощью меню **Цвет – Составляющие – Собрать.** Например, чтобы избежать искажения цветов при повышении резкости изображения, рекомендуется применять эту операцию только к каналу яркости (V-каналу) в модели HSV.

Сохранение выделенной области

Когда вы выделяете какую-то область рисунка, создаётся маска — временный канал, в котором чёрный цвет обозначает невыделенные пиксели, а белый — выделенные. Эту область можно запомнить как новый канал (**Выделение – Сохранить в канале**).

Каналы, полученные в результате сохранения выделенных областей (на рис. 8.24 — канал **Цветок**), располагаются в нижней части окна **Каналы** (ниже цветовых каналов). Чтобы снова превратить такой канал в выделение, нужно выделить его в списке и щёлкнуть на кнопке.

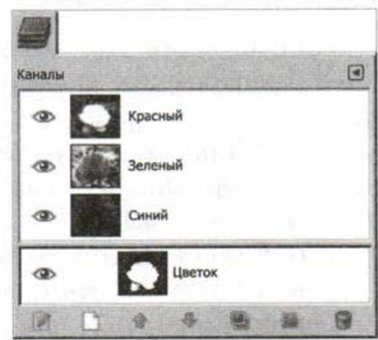


Рис. 8.24

Вопросы и задания

1. Что такое канал?
2. Что такое альфа-канал?
3. Можно ли сказать, что маска — это канал? Ответ обоснуйте.
4. Сколько ступеней выделения можно применять, если использовать 16 битов на канал?
5. Как можно сделать изображение с прозрачными областями?
6. Как разбить изображение на отдельные каналы в цветовых моделях CMYK и HSV?
7. Как собрать изображение из отдельных каналов?
8. Как сохранять выделенные области в каналах и использовать их повторно?

Подготовьте сообщение

а) «Редактирование изображений в модели CMYK»
б) «Редактирование изображений в модели HSV»
в) «Редактирование изображений в модели Lab»

Задача

Разбейте какое-нибудь размытое изображение на составляющие модели HSV, примените операцию повышения резкости к каналу V и соберите изображение заново. Сравните результат с тем, что получается при применении той же операции повышения резкости ко всем каналам.

§ 63
Иллюстрации для веб-сайтов

Иллюстрации для веб-сайтов должны быть сохранены только в тех графических форматах, которые умеют отображать браузеры:

- **JPEG** (англ. *Joint Photographic Experts Group* — объединённая группа фотографов-экспертов, файлы с расширением jpg или jpeg);
- **GIF** (англ. *Graphics Interchange Format* — формат для обмена изображениями, файлы с расширением gif);
- **PNG** (англ. *Portable Network Graphics* — переносимые сетевые изображения, файлы с расширением png).

Все эти форматы предназначены только для «плоских» (однослойных) изображений, поэтому все слои многослойных изображений при сохранении приходится сводить в один слой. При этом теряется информация (снова разделить слои практически невозможно), поэтому рекомендуется всегда оставлять на всякий случай исходные многослойные файлы (в форматах PSD или XCF).

В форматах JPEG, GIF и PNG используется сжатие данных для того, чтобы уменьшить объём файлов и таким образом ускорить загрузку веб-страницы. Если увеличивать степень сжатия, то качество рисунка ухудшается, и наоборот. Поэтому при сохранении нужно установить максимальную степень сжатия, при которой изображение выглядит приемлемо.

При выборе формата для конкретного изображения нужно учитывать, что:

- в формате JPEG можно хранить только **полноцветные изображения** (с глубиной цвета 24 бита на пиксель); для уменьшения объёма файла используется сжатие с потерями, которое приводит к размытию границ объектов, появлению пятен вокруг них и одноцветных квадратов размером 8 × 8 пикселей (так называемые *артефакты JPEG*);
- формат JPEG не поддерживает прозрачность;
- в формате GIF можно хранить только изображения **с палитрой** (от 2 до 256 цветов);
- анимация поддерживается только в формате GIF;
- в форматах GIF и PNG используется сжатие без потерь (рисунок при сжатии не искажается); для уменьшения объёма файла можно уменьшать количество цветов в палитре;
- полупрозрачные изображения можно сохранять только в формате PNG, где для каждого пикселя может храниться дополнительный байт, задающий прозрачность (альфа-канал).

В таблице на рис. 8.25 показаны изображения, полученные при сохранении одного и того же рисунка в разных форматах, а также указаны области применения каждого формата.

Формат	Примеры			Применение
JPEG	Качество 100, 4743 байта	Качество 20, 983 байта	Качество 0, 518 байтов	Фотографии, непрозрачные изображения с размытыми границами объектов и плавными переходами цветов
GIF	256 цветов, 6165 байтов	16 цветов, 1783 байта	8 цветов, 1184 байта	Рисунки с небольшим количеством цветов; мелкие рисунки с чёткими границами; рисунки с прозрачными областями; анимация
PNG	Режим RGB, 7283 байта	16 цветов, 10440 байтов	8 цветов, 1061 байт	Высококачественные изображения, рисунки с прозрачными и полупрозрачными областями

Рис. 8.25

Чтобы при сохранении рисунка вручную задать количество используемых цветов, необходимо преобразовать его к **индексированному режиму**, т. е. закодировать с палитрой. Для этого используется пункт меню **Изображение – Режимы – Индексированное.** Если исходное изображение имеет глубину цвета 24 бита на пиксель (режим RGB), при таком преобразовании происходит потеря информации о цвете. Поэтому лучше работать с копией рисунка, сохранив полноцветный оригинал в отдельном файле. Заметим, что формат PNG обеспечивает лучшее сжатие, чем GIF.

Как видно из таблицы на рис. 8.25, кодирование с палитрой (форматы GIF и PNG) плохо подходит для хранения изображений с плавными переходами цветов **(градиентами).** Это связано с тем, что уменьшается количество используемых цветов и переходы становятся более резкими. Чтобы несколько улучшить результат, при переходе к индексированному изображению можно включить **размывание цвета** (англ. *dithering* — растрирование). Суть этого подхода состоит в том, что области, залитые в исходном рисунке

«отброшенными» цветами, строятся как мозаика из пикселей тех цветов, которые остались в палитре. На рисунке 8.26 показан результат применения размывания при использовании 8-цветной палитры (редактор GIMP, размывание Флойда–Стейнберга). Объём файла увеличился с 1184 до 1664 байтов из-за того, что алгоритм сжатия LZW, используемый в формате GIF, хуже сжимает разноцветные области, чем строки одного цвета.

Рис. 8.26

Вопросы и задания

1. Какие форматы используются для хранения изображений на веб-страницах?
2. Опишите особенности и области применения каждого из форматов.
3. Определите лучший формат для сохранения:
   а) фотографии;
   б) изображения с чёткими границами областей;
   в) изображения с прозрачными областями;
   г) изображения с полупрозрачными областями;
   д) анимации.
4. Что такое индексированное изображение?
5. Что происходит при переходе от режима RGB к индексированному изображению? Как выбрать количество цветов в палитре?
6. Что такое размывание цвета и зачем оно используется?

Подготовьте сообщение

«Оптимизация изображений для веб-страниц»

§ 64 Анимация

Анимация (англ. *animation* — одушевление) — это «оживление» изображения. При анимации несколько рисунков (кадров) сменяют друг друга через заданные промежутки времени.

Для создания анимации в графических редакторах обычно используются многослойные изображения, каждый слой соответствует одному кадру. В простейшем случае слои просто меняют друг друга: сначала виден только первый слой, через заданное время он скрывается и показывается второй слой и т. д. Такой метод называется **заменой** (англ. *replace*) — рис. 8.27.

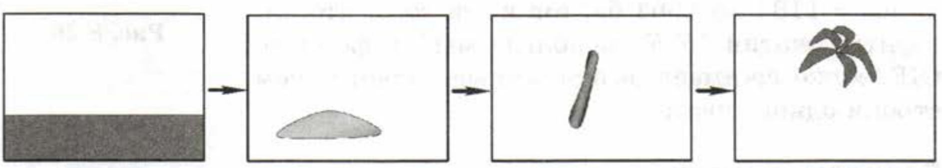


Рис. 8.27

Можно использовать и другой подход: новый слой «накладывается» на существующее изображение — это метод **объединения** (англ. *combine*). Для того же набора слоёв, что и в предыдущем примере, анимация такого типа показана на рис. 8.28.

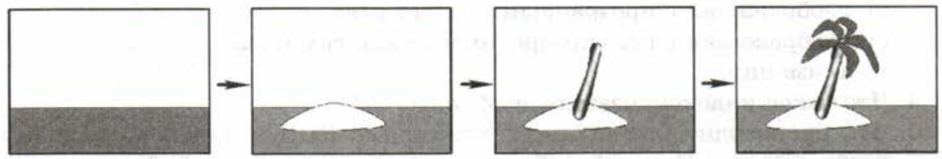


Рис. 8.28

Тип анимации для каждого слоя устанавливается отдельно, поэтому в одной анимации можно совмещать оба метода.

Анимация строится за два шага:

1) готовятся изображения для всех слоёв;
2) устанавливается время задержки и вид анимации (для каждого слоя отдельно).

Чтобы готовую анимацию можно было просматривать без графического редактора (например, на веб-странице), нужно сохранить её в файле формата GIF (т. е. **экспортировать** — записать в другом формате с преобразованием).

В редакторе GIMP при сохранении файла с несколькими слоями в формате GIF появляется диалоговое окно, в котором нужно выбрать вариант **Сохранить как анимацию** и нажать кнопку **Экспорт** (рис. 8.29).

Рис. 8.29

После этого в следующем окне нужно настроить параметры сохранения анимации (рис. 8.30).

Сохранить как GIF
Параметры GIF
Черезстрочность
Комментарий GIF: Created with GIMP
Параметры анимированного GIF
Бесконечный цикл
Если задержка между кадрами не указана, она равна: 100 миллисекунд
Расположение кадра, если не указано: Наложение слоев (об'единение)
Использовать указанную задержку в дальнейшем
Использовать указанное расположение в дальнейшем
Справка
Сохранить
Отменить

Рис. 8.30

Если установлен флажок **Бесконечный цикл**, то анимация будет повторяться бесконечно, иначе она выполнится только один раз. Здесь же выбирается задержка между кадрами и вид анимации (список **Расположение кадра**).

Если теперь открыть сохранённый таким образом GIF-файл с анимацией, в названиях слоёв-кадров мы увидим в скобках время задержки (например, **100ms** обозначает 100 миллисекунд) и метод анимации (**combine** или **replace** — рис. 8.31). Изменив названия слоёв, можно задать задержку и метод анимации для каждого слоя отдельно.

Чтобы просмотреть анимацию, не выходя из редактора GIMP, нужно выбрать фильтр **Анимация** – **Воспроизвести** в меню **Фильтры**.

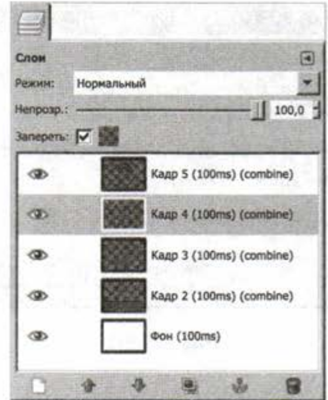


Рис. 8.31

В группе **Анимация** есть фильтр **Оптимизация,** с помощью которого можно существенно уменьшить объём GIF-файла с анимацией. Это особенно важно, если анимация размещается на веб-странице. Оптимизация основана на том, что многие пиксели разных кадров совпадают, поэтому можно удалить все повторяющиеся части и хранить их только один раз.

Вопросы и задания

1. Что такое анимация?
2. Какие два метода анимации вы знаете?
3. Что такое экспортирование документа? Зачем оно применяется?
4. Как в редакторе GIMP установить время задержки и способ анимации для каждого кадра отдельно?
5. На чем основана оптимизация файлов с анимацией?

Подготовьте сообщение

«Анимация на веб-страницах: за и против»

§ 65 Контуры

Современные растровые графические редакторы (Adobe Photoshop, GIMP) могут работать с векторными изображениями — **контурами** (англ. *path*). Контур хранится в памяти не как набор пикселей, а как *кривая Безье*, которая задаётся опорными

точками (на рис. 8.32 — точки *А*, *Б*, *В*, *Г* и *Д*) и координатами «рычагов» (управляющих линий), связанных с каждой точкой (см. § 16 из учебника для 10 класса).

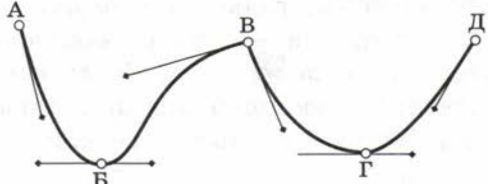


Рис. 8.32

Если оба управляющих рычага находятся на одной прямой, получается сглаженный узел (узлы *Б* и *Г* на рисунке), если нет, то угловой узел (например, узел *В*).

С помощью контуров удобно выделять области рисунка, которые должны иметь строго определённые «правильные» границы. Контуры сохраняются в файле (в форматах PSD и XCF), после создания их можно многократно изменять.

Для создания нового контура в редакторе GIMP надо выбрать инструмент **Контуры** и щелчками мышью обозначить узлы. Чтобы построить гладкий узел, нужно «вытащить» из него управляющую линию, не отпуская левую кнопку мыши. Контур можно замкнуть, щёлкнув на первом узле при нажатой клавише Ctrl.

Контур может состоять из нескольких несвязанных частей. Чтобы начать новую часть контура, нужно при добавлении узла нажать клавишу Shift. Чтобы объединить две части, нужно выделить конечный узел одной из них и при нажатой клавише Ctrl щелкнуть на узле, с которым его нужно соединить.

После создания контур можно изменять, перетаскивая узлы и управляющие рычаги. Если при этом удерживать клавишу Shift, оба рычага будут расположены на одной прямой и получается гладкий узел.

Новые узлы добавляются на соединяющие линии при нажатой клавише Ctrl. Для удаления узлов и соединяющих линий нужно удерживать одновременно Ctrl и Shift.

Чтобы выделить одновременно несколько узлов, нужно удерживать клавишу Shift. При нажатой клавише Alt контур можно перемещать как одно целое.

Флажок **Многоугольник** на панели свойств инструмента **Контуры** позволяет работать с многоугольниками, т. е. сегменты не будут искривляться.

Новый контур появляется в окне **Контуры** — рис. 8.33 (меню **Окна – Прикрепляющиеся диалоги – Контуры**). Это окно аналогично окну **Слои**, с которым вы уже знакомы. Поскольку контуры в GIMP предназначены, главным образом, для выделения областей, основные операции — это преобразование контура в выделенную область (кнопка) и наоборот (кнопка). Когда область с растушёванной границей преобразуется в контур, растушёвка пропадает, граница становится резкой.

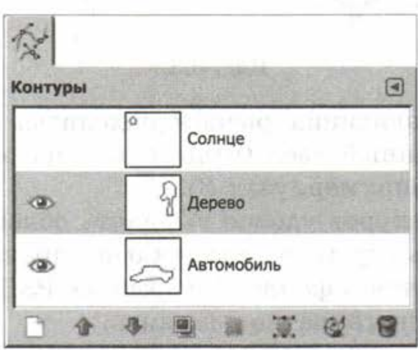


Рис. 8.33

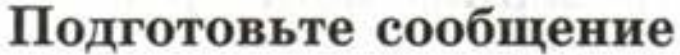
Кроме того, можно расположить текст вдоль контура. В редакторе GIMP для этого надо выделить контур в окне **Контуры**, добавить новый текст (текстовый слой) с помощью инструмента **Текст** и щёлкнуть по кнопке **Текст по контуру** в окне параметров инструмента **Текст**. При этом строится новый контур по форме букв, размещённых вдоль контура. Затем можно залить эту область цветом (например, с помощью меню **Правка – Залить цветом переднего плана**), а текстовый слой удалить, он больше не нужен. Конечно, редактировать такой текст уже нельзя, потому что он хранится как растровая картинка.

Контуры, созданные в редакторе GIMP, можно сохранить на диске в виде файлов формата SVG (англ. *Scalable Vector Graphics* — масштабируемые векторные изображения). Для этого используют команду **Экспортировать контур** из контекстного меню **Контуры**, которое можно вызвать щелчком правой кнопкой мыши в списке контуров или с помощью кнопки . Вместе с тем контуры, созданные в других программах (например, в Inkscape) и сохранённые в формате SVG, можно загружать в GIMP с помощью команды **Импортировать контур** того же меню.

Вопросы и задания

1. Что такое контур? Из каких элементов состоит контур?
2. Как регулируется угол наклона касательной в узлах контура?
3. Что такое гладкий узел, угловой узел?
4. В каких форматах контуры сохраняются вместе с основным изображением?
5. Как можно редактировать контур?
6. Как превратить контур в выделение, и наоборот?
7. Как вы думаете, почему при преобразовании выделенной области в контур пропадает растушёвка? Как сохранить область с растушёвкой для повторного использования? Приведите примеры.

8. Как расположить текст вдоль контура? Почему после этого текст нельзя редактировать?



Подготовьте сообщение

«Использование контуров в практических задачах»

Практические работы к главе 8

Работа № 67 «Ввод и кадрирование изображений»
Работа № 68 «Коррекция фотографий»
Работа № 69 «Работа с областями»
Работа № 70 «Работа с областями»
Работа № 71 «Многослойные изображения»
Работа № 72 «Многослойные изображения»
Работа № 73 «Каналы»
Работа № 74 «Иллюстрации для веб-сайтов»
Работа № 75 «GIF-анимация»
Работа № 76 «Контуры»

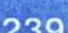
ЭОР к главе 8 на сайте ФЦИОР (http://fcior.edu.ru)

- Растровые редакторы

- Размещение графики на интернет-странице

Самое важное в главе 8

- Растровое изображение — это набор пикселей. Цвет пикселя задаётся в виде числового кода.



- Качество растрового изображения определяется разрешением и глубиной цвета. Чем выше разрешение и глубина цвета, тем лучше качество.
- Файл, в котором хранится растровое изображение, содержит не только коды пикселей, но и служебную информацию: размеры рисунка, цветовую палитру и др.
- Редактирование растрового изображения — это изменение кодов, задающих цвета пикселей, с помощью математических операций.
- Ретушь — это исправление дефектов изображения.
- Фильтр — это алгоритм автоматической обработки пикселей изображения, который применяется ко всему изображению или к выделенной области. Различают фильтры для коррекции изображений и художественные фильтры.
- В многослойных документах на каждом слое может строиться отдельное изображение, которое редактируется независимо от других. Каждый слой может разными способами накладываться на изображение, полученное с предыдущих слоёв.
- Канал — это чёрно-белое полутоновое изображение, которое показывает степень влияния какого-то эффекта на изображение. Например, степень прозрачности изображения записывается в так называемый альфа-канал.
- Анимация — это быстрая смена отдельных изображений, создающая у человека иллюзию движения. Анимация строится на основе многослойных изображений.
- Современные графические форматы позволяют хранить векторные и растровые элементы изображения в одном файле.

Глава 9
Трёхмерная графика

§ 66
Введение

Что такое трёхмерная графика?

Раньше, говоря о компьютерной графике, мы имели в виду двумерные («плоские») изображения. Невозможно «повернуть» автомобиль, изображённый на таком рисунке, и посмотреть на него с другой стороны. В то же время реальный автомобиль — это трёхмерный объект, поэтому при решении многих задач его «плоской» модели (рисунка, фотографии) недостаточно.

Трёхмерная графика (3D-графика, от англ. *3-Dimensions* — 3 измерения) — это раздел компьютерной графики, который занимается созданием моделей и изображений трёхмерных объектов.

В программах для работы с 3D-графикой строятся трёхмерные (пространственные) модели объектов, в которых каждая точка имеет три координаты (а не две, как на «плоском» рисунке). Затем пользователь может выбрать в пространстве точку наблюдения и получить плоское изображение, т. е. построить проекцию трёхмерной сцены на плоскость. Многие программы позволяют создавать анимацию, показывающую движение трёхмерных объектов в пространстве.

3D-модели применяются не только для построения двумерных изображений. Их используют для различных вычислений, например для расчёта прочности деталей. В последние годы активно разрабатываются 3D-принтеры, которые позволяют методом послойной печати построить объёмный физический объект (чаще всего из пластика) по его трёхмерной модели.

Перечислим важнейшие области применения трёхмерной графики:

- компьютерное проектирование машин и механизмов (САПР — системы автоматизированного проектирования[1]);
- компьютерные тренажёры и обучающие программы;
- построение трёхмерных моделей в науке, промышленности, медицине;
- дизайн зданий и интерьера (внутренней обстановки);
- компьютерные эффекты в кино и телевидении; существуют даже полнометражные фильмы, которые полностью созданы с помощью трёхмерной графики и анимации;
- телевизионная реклама;
- интерактивные игры.

Создание изображений с помощью 3D-графики включает несколько этапов:

1) **моделирование** — создание трёхмерных объектов, персонажей;
2) **текстурирование** (раскраска) — наложение на модели рисунков (текстур), которые имитируют реальный материал (дерево, мрамор, металл, кожу и пр.);
3) **освещение** — установка и настройка источников света;
4) **анимация** — описание изменения объектов во времени (изменение положения, углов поворота, свойств);
5) **съёмка** — установка камер (выбор точек съёмки), перемещение камер по сцене;
6) **рендеринг** (визуализация) — построение фотореалистичного изображения или анимации.

Среди профессиональных программ 3D-моделирования наиболее популярны продукты фирмы Autodesk (www.autodesk.com): *3D Studio MAX*, *Maya* и *AutoCAD*, а также программа *Cinema4D* фирмы MAXON (www.maxon.net). Мы будем использовать для иллюстраций свободно распространяемый пакет *Blender* (www.blender.org), версии которого существуют для операционных систем Windows, Linux, Mac OS и других.

Для работы с программами трёхмерной графики нужен компьютер с мощным процессором и большим объёмом оперативной и дисковой памяти. Построение качественных фотореалистичных изображений (которые выглядят как фотографии) занимает огромное время, иногда несколько часов расчётов на один кадр.

[1] В английском языке — **CAD-системы** (CAD — Computer Aided Design, проектирование с помощью компьютера).

Во многих программах есть возможность **сетевого рендеринга**, когда для расчёта изображения используются мощности нескольких компьютеров, объединённых в сеть (англ. *render farm* — рендер-фермы).

Проекции

Хотя программы трёхмерной графики предназначены для создания трёхмерных моделей объектов, пользователь видит только плоское (двухмерное) изображение на мониторе или бумажном отпечатке, т. е. **проекцию**. На рисунке 9.1 показаны четыре проекции модели головы обезьянки Сюзанны (объект **Monkey**), которая включена в набор стандартных объектов программы Blender. Вы видите три стандартные проекции этой модели (виды спереди, сверху и справа) и одну произвольную проекцию (проекцию пользователя).

Рис. 9.1

Программа Blender позволяет видеть четыре проекции одновременно или оставить только одну проекцию пользователя, которая занимает всю рабочую область. Для переключения между этими режимами используется комбинация клавиш Ctrl+Alt+Q.

Обычно работают с одним видом, который занимает всю рабочую часть окна. Для быстрого перехода к стандартным проекциям (видам спереди, сверху, справа и др.) используется меню **Вид (View)** или дополнительная цифровая клавиатура (англ. *numpad*), расположенная в правой части стандартной клавиатуры (рис. 9.2).

Рис. 9.2

Далее для обозначения этих клавиш будем использовать «приставку» *Num*, например *Num*1 обозначает клавишу «1» на дополнительной цифровой клавиатуре.

Существует два типа проекций: **перспективные** и **ортогональные** (их также называют **прямоугольные** или **ортографические**). На рисунке 9.3 показаны перспективная и ортогональная проекции куба.

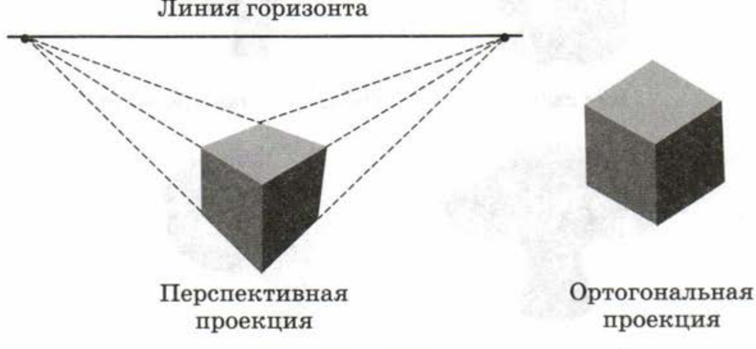


Рис. 9.3

Наш взгляд привык к **перспективе**: удалённые предметы кажутся меньше по размеру, параллельные линии «сходятся» в бесконечной точке (вспомните, как выглядит уходящее вдаль шоссе). Однако при трёхмерном моделировании такие проекции не совсем удобны, потому что искажают форму и размеры объектов.

Для **ортогональной проекции** всё по-другому: размеры не зависят от расстояния до предмета, а параллельные грани остаются параллельными и на проекции. Ортогональные проекции очень полезны, потому что делают трёхмерную сцену проще и позволяют оценить истинные размеры объектов.

В редакторе Blender тип проекции показывается в левом верхнем углу рабочего окна. Например, надпись **Top Ortho** означает «вид сверху» (англ. *top view*), ортогональная проекция (англ. *orthograpgic*). Надпись **Front Persp** означает «вид спереди» (англ. *front view*), перспективная проекция (англ. *perspective*).

Чтобы переключиться с ортогональной проекции на перспективную или наоборот, нужно нажать кнопку «5» на дополнительной клавиатуре (*Num*5).

Вращая колёсико мыши, пользователь может уменьшать и увеличивать масштаб изображения в окне, над которым находится курсор мыши (размеры самого объекта при этом не меняются). Для вращения произвольной проекции нужно перемещать мышь при нажатой средней кнопке (колёсике). Нажав одновременно колёсико мыши и клавишу Shift, можно перемещать изображение в окне, не поворачивая его.

Для вращения и перемещения удобно использовать клавиши-стрелки на дополнительной цифровой клавиатуре (*Num*2, *Num*4, *Num*6 и *Num*8): в обычном режиме они вращают сцену, а при нажатой клавише Ctrl перемещают точку наблюдения.

Вопросы и задания

1. Как строится двумерное изображение трёхмерной модели?
2. В каких задачах необходимо использование трёхмерных моделей?
3. Как вы думаете, в каком виде хранится в памяти информация о трёхмерных объектах?
4. На каких этапах создания изображений в программах 3D-моделирования используется векторная и растровая графика?
5. Объясните, что такое моделирование, текстурирование, рендеринг.
6. Вспомните, что такое свободное программное обеспечение. В чём его достоинства и недостатки?
7. Что такое кроссплатформенное программное обеспечение? Относится ли программа Blender к этому типу программ?
8. Объясните, почему для работы с программами трёхмерной графики требуются мощные компьютеры.
9. Что такое проекции? Зачем они нужны?
10. Чем отличаются перспективные и ортогональные проекции? Когда их удобно использовать?

Подготовьте сообщение

«Программы для 3D-моделирования»

Задача

Загрузите в программу любую трёхмерную модель и научитесь ориентироваться в трёхмерном пространстве: переходить от одной проекции к другой, менять точку наблюдения, наблюдать разные стороны объектов.

§ 67 Работа с объектами

Примитивы

Построение трёхмерных моделей обычно начинается с **примитивов** — простейших объектов. К ним относятся плоскость (точнее, её прямоугольная часть), куб, сфера, цилиндр, конус, тор и некоторые другие (рис. 9.4).

Куб Сфера Цилиндр Конус Тор

Рис. 9.4

При создании объекта ему автоматически присваивается имя. Например, первый куб будет называться *Cube*, второй — *Cube.001* и т. д. Это имя можно изменить на панели свойств объекта, которая расположена в правой части окна программы Blender (рис. 9.5).

Рис. 9.5

Выделение объектов

Для того чтобы работать с объектом, например изменять его свойства, предварительно нужно выделить его. В программе Blender для этого используется *правая* кнопка мыши (а не левая, как в других программах).

При нажатой клавише Shift можно выделить правой кнопкой мыши несколько объектов одновременно. Если повторно щёлкнуть на объекте, выделение снимается.

С помощью клавиши *A* (от англ. *all* — всё) снимается выделение со всех объектов, а если ни один объект не был выделен, все они выделяются.

После нажатия клавиши *B* (от англ. *border select* — выделить рамкой) можно с помощью мыши обвести прямоугольной рамкой все объекты, которые нужно выделить. Если объект хотя бы частично попал в рамку, он будет выделен. Кроме того, можно нажать левую кнопку мыши и, удерживая нажатой клавишу Ctrl, «обвести» все нужные объекты произвольным контуром.

Для мелких объектов удобно использовать **круговое выделение**. Этот инструмент включается при нажатии клавиши *C*: появляется окружность, размер которой регулируется колёсиком мыши. Затем щелчком (или «протаскиванием») левой кнопкой мыши выделяются все элементы, попадающие внутрь окружности. Если в этом режиме случайно выделен лишний объект, выделение можно снять, нажав на колёсико мыши.

Существуют и другие, более сложные способы выделения объектов, которые доступны через меню **Select**.

С помощью клавиши *Num.* (точка на цифровой клавиатуре) можно приблизить выделенный объект, а клавиша *Num/* временно скрывает остальные объекты (кроме выделенных).

Преобразования объектов

Любой объект можно перемещать, вращать и масштабировать (изменять размеры). Эти операции называются **преобразованиями** объектов или трансформациями (англ. *transformations*).

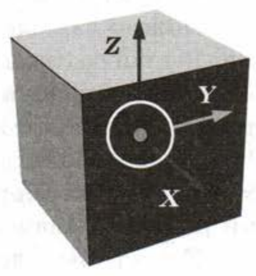


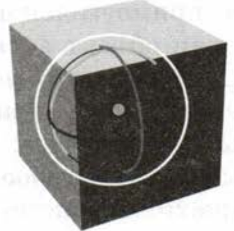
Рис. 9.6

В опорной точке (англ. *origin* — начало координат) выделенного объекта появляется так называемый манипулятор с тремя стрелками, параллельными осям координат (рис. 9.6). Ось *Z* (синяя стрелка в Blender) на-

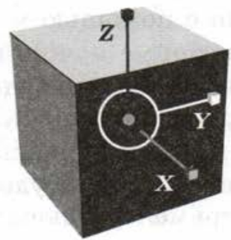
правлена вверх, а оси X (красная стрелка в Blender) и Y (зелёная стрелка в Blender) находятся в горизонтальной плоскости.

За эти стрелки объект можно перетаскивать левой кнопкой мыши, меняя его положение только по одной выбранной оси. Центральный круг дает возможность произвольно перемещать объект в плоскости проекции.

Объекты можно вращать как в плоскости проекции (нажав клавишу R, от *англ.* rotate — вращение), так и вокруг одной выбранной оси (для этого после нажатия клавиши R нужно нажать клавишу с названием оси — X, Y или Z). Можно использовать специальный манипулятор вращения, позволяющий вращать фигуру за выделенные части окружностей (рис. 9.7)



Манипулятор вращения



Манипулятор изменения размеров

Рис. 9.7

Чтобы изменить размеры объекта, нужно нажать клавишу S (от англ. *scale* — изменить масштаб) и перемещать мышь. Чтобы изменять размер только по одной из осей, нужно после нажатия клавиши S нажать клавишу с названием оси. Манипулятор изменения размеров содержит небольшие кубики на концах указателей осей (см. рис. 9.7). Перетаскивая один из них, можно менять соответствующий размер.

Координаты, углы поворота и размеры объектов можно задавать с клавиатуры в числовой форме. В программе Blender для этого используют панель **Преобразование** (Transform) — рис. 9.8, которая появляется при нажатии клавиши N. Числовые свойства можно также изменять с помощью мыши (перетаскивая влево и вправо стрелки рядом со значениями).

Рис. 9.8

Копия выделенного объекта создаётся с помощью клавиш Shift+D. Затем объект-копию

нужно переместить мышью в нужное место и зафиксировать щелчком левой кнопкой. Если предварительно нажать клавишу *X*, *Y* или *Z*, объект-копия будет перемещаться только вдоль указанной оси.

Удалить выделенный объект со сцены можно с помощью клавиши Delete.

Системы координат

По умолчанию используется **глобальная («мировая») система координат**, начало которой находится в точке с координатами (0,0,0) пространства сцены, она не зависит от положения объекта. Иногда бывает удобно перейти к **локальной системе координат**, которая связана с объектом. Её центр находится в опорной точке объекта, а оси меняют направление при его вращении, показывая, где у объекта «верх» (локальная ось $Z_{лок}$), «право» (локальная ось $X_{лок}$) и т. д. На рисунке 9.9 показаны направления осей глобальной и локальной систем координат для повёрнутого конуса.

Глобальная система координат — Локальная система координат

Рис. 9.9

В программе Blender можно выбрать нужную в данный момент систему координат с помощью выпадающего списка в нижней части рабочего окна. Именно эти оси будут использоваться при перемещении, вращении и изменении размеров объекта.

Слои

Трёхмерная сцена в Blender может состоять из нескольких слоёв. В рабочем окне видны только те объекты, которые расположены на активном слое. Активный слой можно выбрать щелчком мышью на специальной панели в нижней части рабочего окна (рис. 9.10).

Рис. 9.10

Всего можно использовать 20 слоёв, в данном случае (см. рис. 9.10) какие-то объекты есть на слоях 1, 2 и 4. Активный слой — слой 2, его клетка выделена тёмно-серым фоном. Щёлкая мышью на клетках панели при нажатой клавише Shift, можно сделать активными несколько слоёв, чтобы показать все связанные с ними объекты.

Объект может принадлежать не только одному, но и нескольким слоям. Для этого их нужно выделить в аналогичном элементе управления на панели свойств объекта. Здесь же можно перевести объект с одного слоя на другой. Для этих операций можно также использовать всплывающее окно, которое появляется при нажатии клавиши *M*.

Связывание объектов

Представим себе, что мы построили трёхмерную модель стола, на котором размещены тарелки, чашки и столовые приборы. Теперь нужно передвинуть стол в другое место сцены и немного развернуть его. При этом точно так же должны переместиться и все объекты, стоящие на столе. Чтобы проще решить эту задачу, нужно «привязать» к столу все стоящие на нем предметы, т. е. на панели свойств установить для них так называемый **родительский объект** (англ. *parent* — родитель). Можно поступить иначе: выделить, удерживая клавишу Shift, все нужные объекты (тарелки, чашки и т. д.), последним выделить родительский объект (стол) и нажать клавиши Ctrl+*P*.

После этого при всех преобразованиях объекта-родителя (перемещении, вращении, масштабировании) те же самые преобразования применяются и к объекту-потомку. В то же время объект-потомок по-прежнему можно перемещать независимо от родителя, т. е. мы можем свободно двигать чашку по столу.

Обратите внимание на отличие группы в Blender от аналогичных средств в большинстве программ, работающих с графикой: объекты в такой связанной группе неравноправны — среди них есть один родительский объект и несколько потомков.

Рис. 9.11

Всю структуру сцены (иерархию объектов) можно посмотреть в специальном окне **Структура проекта** (Outliner) — рис. 9.11. Здесь видно, например, что объект Cube — родительский для объекта Cube.001.

Справа от имени объекта в окне **Структура проекта** показаны три значка: щелчок мышью на изображении глаза скрывает объект; если щёлкнуть на значке , то объект будет нельзя выделить (иногда это не нужно), а щелчком на значке можно запретить показ объекта при рендеринге.

Вопросы и задания

1. Что такое примитивы? Зачем они нужны?
2. Как вы думаете, зачем каждому объекту сцены присваивается уникальное имя?
3. Как выделить одновременно несколько объектов?
4. Какие преобразования объектов вы знаете?
5. Как можно применить преобразования только по одной оси?
6. Что такое манипуляторы? Как их использовать?
7. Какие системы координат применяются при трёхмерном моделировании? Чем они различаются и когда используются?
8. Зачем нужны слои?
9. В каких случаях удобно использовать связь объектов «родитель – потомок»?

Задача

В программе трёхмерного моделирования научитесь создавать различные типы примитивов и применять к ним преобразования.

§ 68 Сеточные модели

Как строятся объекты?

По умолчанию объекты в Blender изображаются как объёмные твердые тела (англ. *solid* — сплошной). При этом не сразу понятно, как же программа может быстро перестраивать изображение

при изменении точки наблюдения. Для того чтобы понять внутреннее устройство объектов, нужно с помощью списка переключиться в другой режим, который называется **Каркас** (англ. *wireframe*). Мы увидим, что каркас куба включает (рис. 9.12):

- 8 вершин: *A*, *B*, *C*, *D*, *E*, *F*, *G* и *H*;
- 12 рёбер, соединяющих вершины: *AB*, *AD*, *BC*, *CD*, *EF*, *EH*, *FG*, *GH*, *AE*, *BF*, *CG* и *DH*.

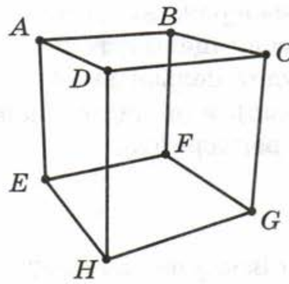


Рис. 9.12

Вершина (англ. *vertex*) — это точка в трёхмерном пространстве, которая задается тремя координатами. **Ребро** (англ. *edge*) — это отрезок, соединяющий две вершины. Рёбра ограничивают **грани** (англ. *face*) — участки поверхностей. У куба 6 граней: *ABCD*, *EFGH*, *ABFE*, *CDHG*, *ADHE* и *BCGF*.

Такие модели называются **сеточными** (англ. *mesh*), потому что они представляют собой поверхности, которые строятся на сетке из рёбер. Чаще всего используются треугольные и четырёхугольные грани, однако последние версии Blender могут работать с гранями, состоящими из большего числа рёбер, — так называемыми *N*-угольниками или **полигонами**. Поэтому часто применяют выражение *«полигональная модель»* (от англ. *polygon* — многоугольник, полигон). Сфера и другие криволинейные поверхности тоже строятся из плоских граней, но их значительно больше, чем у куба.

Трёхмерные модели хранятся в векторном формате, для построения поверхности достаточно запомнить координаты вершин каркаса. По ним можно рассчитать координаты всех точек рёбер и граней. Каждая грань обрабатывается отдельно, поэтому чем больше граней, тем большее время требуется для расчётов.

Редактирование сетки

Для изменения положения элементов сетки нужно перейти в режим редактирования (англ. *Edit mode*). В программе Blender для этого используется клавиша Tab.

В режиме редактирования можно работать с вершинами, рёбрами и гранями. Нужный тип объектов выбирается с помощью показанного на рис. 9.13 элемента управления, который расположен в нижней части рабочего окна, или с помощью всплывающего меню, которое вызывается нажатием клавиш Ctrl+Tab. На рисунке 9.13 первая кнопка выделена тёмным фоном, это означает, что включён режим работы с вершинами.

Рис. 9.13

При нажатой клавише Shift можно включить несколько режимов выделения сразу, например выделять рёбра и грани.

Для выделения элементов сетки используются те же методы, что и при выделении объектов. Затем их можно перемещать, вращать и масштабировать. Очень удобно использовать *круговое выделение* (см. § 67).

По умолчанию при работе с рёбрами и гранями выделяются не только видимые, но и невидимые элементы, расположенные на задней поверхности объекта. Чтобы этого не происходило, нужно ограничить выделение только видимыми элементами, щёлкнув на кнопке в нижней части рабочего окна.

В режиме работы с вершинами щелчок левой кнопкой мыши при нажатой клавише Ctrl создаёт новую вершину, которая соединяется с уже выделенной вершиной. Если выделить две вершины (используя клавишу Shift) и нажать клавишу *F*, между ними строится новое ребро.

Чтобы создать новую грань, нужно выделить все вершины замкнутого многоугольника и нажать клавишу *F*.

Для доступа к другим операциям с элементами сеточной модели можно использовать всплывающие меню, которые появляются при нажатии клавиш Ctrl+*V* (меню для вершин), Ctrl+*E* (меню для рёбер) и Ctrl+*F* (меню для граней).

В Blender существует особый режим **пропорционального редактирования** (англ. *proportional editing*). В этом режиме переме-

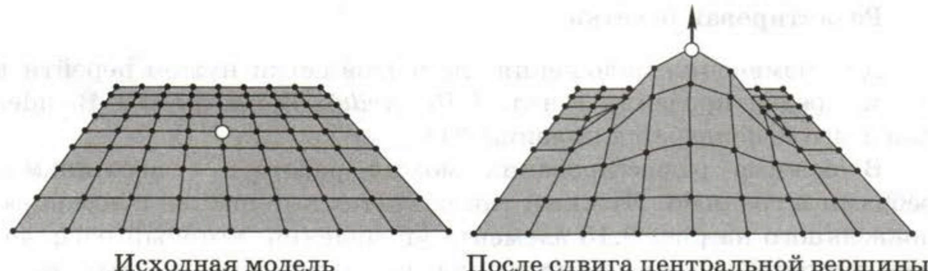


Рис. 9.14

щаемая вершина, ребро или грань увлекает за собой соседние. В примере, показанном на рис. 9.14, перемещалась вверх только центральная вершина сетки.

Деление рёбер и граней

Часто требуется разделить ребро или грань на несколько частей. Для этой цели проще всего использовать инструмент **Подразделять** (Subdivide), который делит выделенные рёбра или грани на несколько равных частей. Примеры его использования показаны на рис. 9.15.

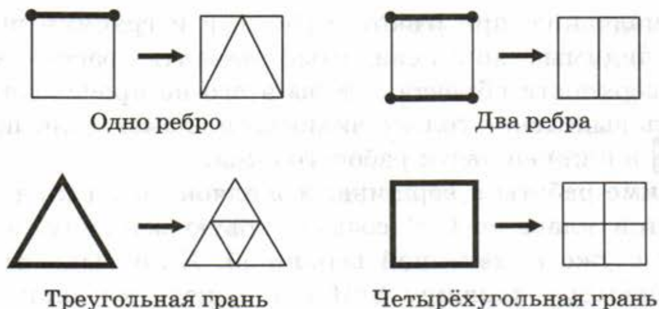


Рис. 9.15

Существует также инструмент **Нож** (Knife), который позволяет «разрезать» выделенные рёбра. Для этого нужно при нажатой клавише *K* нажать и не отпускать левую кнопку мыши, после чего курсор становится похожим на нож. Теперь остаётся провести мышь через точки деления рёбер. Если нажимать Shift+*K* вместо *K*, рёбра, через которые проходит нож, делятся ровно пополам.

Одиночное сечение Множественное сечение

Рис. 9.16

Инструмент **Разрезать петлёй со сдвигом** (Loop Cut and Slide), который вызывается нажатием клавиш Ctrl+R, позволяет рассечь грани по контуру вокруг объекта и сдвинуть сечение в нужное место (рис. 9.16). Колёсиком мыши можно увеличивать число сечений.

Выдавливание

Один из самых полезных инструментов при работе с сеточными моделями — **Выдавить участок** (Extrude). Выдавливание состоит в том, что выбранная грань перемещается вдоль нормали (перпендикуляра к этой грани) и вокруг неё создаются новые грани. На рисунке 9.17 показаны два типа выдавливания центральной верхней грани куба.

Выдавливание наружу Выдавливание внутрь

Рис. 9.17

Для того чтобы выполнить выдавливание, нужно выделить грань, нажать клавишу E и мышью переместить грань в нужное положение.

Выдавливание можно применять также к выделенным рёбрам и вершинам. Чтобы они перемещались только по одной оси, нужно после клавиши E нажать клавишу с названием этой оси (X, Y или Z).

Сглаживание

Вы уже знаете, что модель любой поверхности в программах трёхмерного моделирования строится из отдельных плоских граней. Однако во многих случаях реальный объект моделирования — гладкий, и на картинке нужно получить сглаженную поверхность (рис. 9.18).

Без сглаживания

Со сглаживанием

Рис. 9.18

Для сглаживания стыков между гранями используют инструмент **Сгладить** (Smooth), который может применяться как ко всему объекту, так и к отдельным граням. При этом важно, что геометрия объекта (количество граней, способ разбивки на грани) не изменяется. Обратите внимание на контур объекта — он остался угловатым. Программа выполняет сглаживание («скругление») граней только при выводе изображения на экран, используя специальные алгоритмы затенения.

Есть и другой способ сглаживания — дополнительная разбивка на более мелкие грани, мы рассмотрим его в следующем параграфе.

Вопросы и задания

1. Что такое сеточная модель? Из каких элементов она состоит?
2. Подумайте, какими достоинствами и недостатками обладают сеточные модели.
3. В каком формате (растровом или векторном) хранится информация о сеточной модели?
4. Как связано количество граней модели и время расчёта изображения сцены? Какие рекомендации вы можете дать в связи с этим?
5. Расскажите о приёмах, которые можно использовать для редактирования сетки.
6. Что такое выдавливание?
7. Зачем нужно сглаживание?

Задачи

1. Исследуйте сеточную модель куба и научитесь изменять её.
2. Исследуйте сеточные модели двух типов сфер, которые в Blender называются **UV-сфера** (UV-sphere) и **Икосаэдр** (Icosphere). Чем они различаются?

§ 69 Модификаторы

Что это такое?

Модификатор — это преобразование объекта, которое выполняется автоматически при выводе проекции на экран или построении готового изображения (*рендеринге*). При этом геометрия объекта не меняется, т. е. это неразрушающая операция (действие модификатора всегда можно отменить). Как правило, модификатор имеет настройки, которые можно менять в диалоговом режиме.

Для любого объекта можно использовать несколько модификаторов. Они действуют последовательно, т. е. первый модификатор (верхний в списке) «работает» с исходной сеточной моделью, второй — с результатом работы первого и т. д. Все применённые модификаторы образуют **стек модификаторов**. В программе Blender вершина стека (последний применяемый модификатор) находится в конце списка. Порядок применения модификаторов можно изменять.

Каждый раз, когда пользователь изменяет вид сцены (например, поворачивая или перемещая объекты), для построения проекции модификаторы применяются заново к изменённой сеточной модели, а это требует значительного времени и ресурсов компьютера. Однако исходная сеточная модель остаётся довольно простой, и её легко редактировать.

Когда модель полностью готова, можно раз и навсегда применить модификатор, т. е. внести соответствующие изменения в сеточную модель. Нужно учитывать, что во многих случаях (например, при сглаживании) новая модель будет содержать значительно большее число граней, занимать больше места на диске и требовать больше времени для рендеринга. Кроме того, редактировать её будет намного сложнее.

Далее мы познакомимся с некоторыми часто используемыми модификаторами программы Blender. Про остальные вы можете узнать в справочной системе.

Сглаживание

В природе редко встречаются чёткие и ровные углы. Поэтому для сеточных моделей животных, растений, людей, сказочных персонажей обязательно используют сглаживание.

В предыдущем параграфе уже рассматривался один вариант сглаживания — с помощью инструмента **Сгладить** (Smooth). Подобную операцию можно выполнить с помощью модификатора **Подразделение поверхности** (Subsurf, англ. *subdivision surface* — разбиение поверхностей).

У модификатора **Подразделение поверхности** есть настройки, с помощью которых можно регулировать степень сглаживания (рис. 9.19).

Без сглаживания (6 граней)

Уровень 1 (24 грани)

Уровень 4 (1536 граней)

Рис. 9.19

Симметрия

При моделировании симметричных объектов, например мордочки животного, хочется автоматически поддерживать одинаковую форму левой и правой частей. Для этого удобно применять модификатор **Отражение** (Mirror).

Основной объект в примере на рис. 9.20 — это правая половина модели головы обезьянки Сюзанны, которая включена в Blender как тестовый объект[1]. При использовании модификатора **Отражение** все изменения, которые выполняются с правой частью, автоматически применяются и к левой.

Если применить модификатор, щёлкнув на кнопке **Применить** (Apply), будет построена новая симметричная сеточная модель,

---

[1] В других программах в качестве тестового объекта используют чайник.

Половина головы С модификатором Отражение

Рис. 9.20

в которой левая и правая части независимы друг от друга (т. е. при изменении одной половины вторая уже не будет изменяться).

Логические операции

С помощью модификатора **Логический** (англ. *boolean*) можно строить объединение, пересечение и «разность» двух объектов. На рисунке 9.21 показаны четыре возможные «логические» операции, которые можно применить к кубу и сфере.

Объединение (Union) Пересечение (Intersection) «Куб минус сфера» (Difference) «Сфера минус куб» (Difference)

Рис. 9.21

Легко заметить, что объединение и пересечение можно сопоставить логическим операциям «ИЛИ» и «И», которые вы изучали в 10 классе.

Модификатор применяется к одному объекту, а второй указывается в параметрах модификатора в поле **Объект** (Object). На рисунке 9.22 показаны настройки модификатора **Логический** для случая, когда нужно построить «разность» сферы (объект с именем **Sphere**) и куба (объект **Cube**). Список **Операция** (Operation) позволяет выбрать нужную операцию (здесь — операция **Разность**

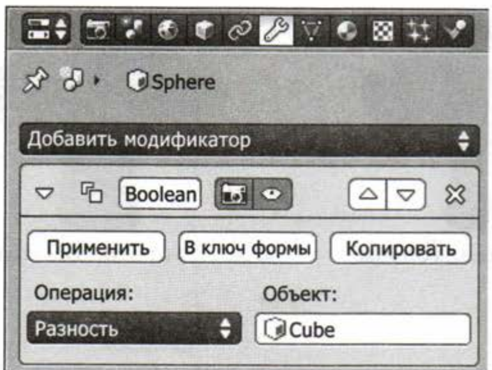


Рис. 9.22

(Difference)). При этом куб никак не меняется, а вместо сферы, к которой применён модификатор, появляется объект «сфера минус куб», т. е. часть шара, которая не входит в куб.

Если после этого сдвинуть куб, полученный объект-«разность» изменится, потому что он каждый раз строится заново с учётом текущего положения сферы и куба. Чтобы новый объект стал независимым, нужно применить модификатор с помощью кнопки **Применить** (Apply). Теперь изменения куба не будут на него влиять.

Сетка объекта, полученного в результате логической операции, значительно усложняется в местах стыковки исходных тел, поэтому потом её довольно сложно редактировать.

Массив

Модификатор **Массив** (Array) позволяет создать несколько копий (**клонов**) основного объекта, которые смещены друг относительно друга на одинаковое расстояние по осям *X*, *Y* и *Z*. Например, так можно из одной модели солдата построить модель целого взвода, стоящего в колонну или шеренгу. Если в том же случае повторно использовать модификатор **Массив** и задать смещение по другой оси, мы сможем построить солдат в несколько колонн.

Копии (клоны) сохраняют связь с основным объектом, при любом его изменении клоны также меняются. Если применить модификатор **Массив** (с помощью кнопки **Применить**), связь копий с исходным объектом разрывается, после этого их можно редактировать независимо друг от друга.

Деформация

Для изменения формы объектов часто удобен модификатор **Решётка** (Lattice). Исходный объект (на рис. 9.23 — сфера) помещается внутрь специальной «клетки» — объекта **Решётка.** Объект **Решётка** — это вспомогательная сетка, которая не показывается при рендеринге, т. е. не влияет на итоговую картинку.

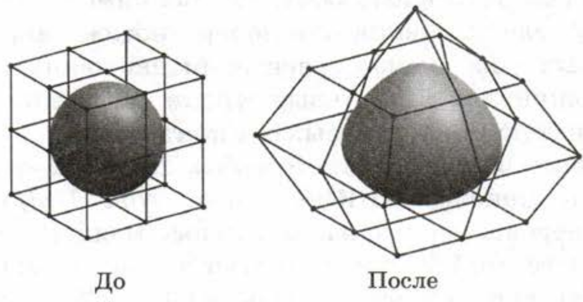


Рис. 9.23

Затем к исходному объекту применяется модификатор **Решётка.** В результате объект и «клетка» оказываются связанными, так что при изменении формы клетки (например, при перемещении её вершин и ребер) меняется и форма основного объекта.

С помощью решётки удобно менять форму объекта при анимации, не меняя его сеточную модель. Например, мяч при отскоке от пола немного сплющивается, а потом опять принимает нормальную форму.

Вопросы и задания

1. Что такое модификатор?
2. Почему модификаторы — это неразрушающий метод редактирования?
3. Что означает «применить модификатор»? Какие достоинства и недостатки имеет этот приём?
4. Что такое стек модификаторов? Как влияет расположение модификаторов в стеке на окончательный результат?
5. Опишите действия модификаторов, рассмотренных в тексте параграфа.

Задачи

1. Научитесь использовать модификаторы, рассмотренные в тексте параграфа.

*2. Найдите документацию по другим модификаторам и научитесь их применять.

§ 70
Кривые

Основные понятия

Кривые используются в программах трёхмерной графики как вспомогательные векторные объекты, например для того, чтобы задать форму тела вращения или поперечное сечение трубы. С помощью кривых определяют траектории движения объектов при анимации, искривляют текстовые строки, моделируют провода и нитки. Замкнутую кривую называют **контуром.**

В программе Blender можно строить два типа кривых — **кривые Безье** и **кривые NURBS** (англ. *Non Uniform Rational B-Splines* — неравномерные рациональные В-сплайны). В этом параграфе мы рассмотрим только кривые Безье как самые простые. Как вы знаете, они состоят из **узлов** (опорных точек) и **сегментов** (соединяющих их линий). Кривизну сегментов определяют направляющие линии (касательные) в каждом узле, положение которых можно регулировать с помощью «рычагов», их концы обозначены на рис. 9.24 белыми кружками.

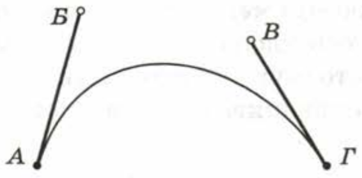


Рис. 9.24

Для построения каждого сегмента используются только 4 точки, показанные на рис. 9.24: два узла, определяющие сегмент (*А* и *Г*), и концы рычагов (*Б* и *В*).

Кривая на этом участке выходит из точки *А* в направлении точки *Б*, затем изгибается к точке *В* и, наконец, входит по касательной в точку *Г*.

В программе Blender различаются четыре типа узлов (рис. 9.25):

- **векторные узлы** (англ. *vector* — векторный), предназначенные для рисования ломаных (касательные направлены в соседнюю точку);
- **гладкие узлы** (англ. *aligned* — выровненный), в которых обе направляющие лежат на одной прямой;

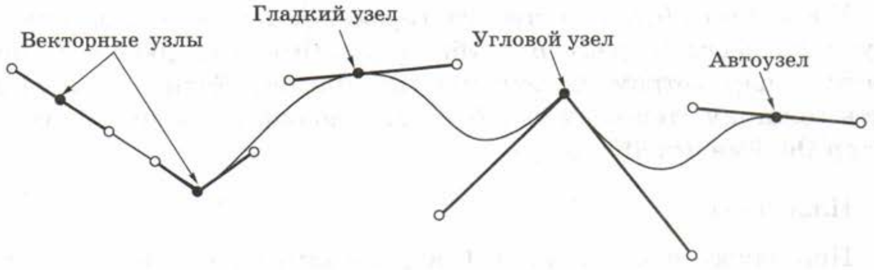


Рис. 9.25

- **угловые,** или **свободные узлы** (англ. *free* — свободный), в которых каждую направляющую можно настраивать независимо от другой;
- **автоузлы** (англ. *auto* — автоматический) — гладкие узлы, в которых положение направляющих выбирается программой.

При создании **кривой Безье** в Blender (меню **Добавить – Кривая Безье** (Add – Curve – Bezier)) строится стандартная кривая с двумя гладкими узлами, находящаяся в плоскости XY. Кривую в целом можно редактировать как обычный объект (перемещать, вращать, изменять размеры).

Для настройки отдельных узлов нужно перейти в режим редактирования (клавиша Tab), в котором можно перемещать сами узлы и их управляющие рычаги. Создать новый узел между двумя выделенными узлами можно с помощью всплывающего меню **Специальные – Подразделить** (Specials – Subdivide), которое появляется при нажатии клавиши W. Меню для выбора типа узла вызывается клавишей V.

Кривые часто используют как вспомогательные объекты для построения более сложных фигур, поэтому при рендеринге их не видно. Однако когда мы моделируем с помощью кривой например, кабель, трубу, нитку и т. п., она должна присутствовать на готовом изображении. Для этого нужно изменить её свойства особым образом:

- сделать кривую трёхмерной (включить режим **3D**);
- в списке **Заполнение** (Fill) выбрать вариант **Полностью** (Full); другие варианты позволяют построить половину или четверть трубы;
- применить **Скос (Bevel)**.

После этого будет построена трубка, её диаметр определяется глубиной скоса (параметр **Глубина** (Depth)), а гладкость поверхности — параметром **Разрешение** (Resolution). Если нужно увеличить толщину стенок, к такой трубке можно применить модификатор **Объёмность** (Solidify).

Пластины

При нажатии клавиш Alt+C кривая замыкается (или размыкается, если она была замкнута). Контур (замкнутую кривую) можно превратить в пластинку, которая будет показана при рендеринге картинки. Для этого в свойствах объекта нужно установить режим **2D** (англ. *2-dimensions*, двумерная, или плоская кривая) и ненулевой параметр **Выдавить** (Extrude), который определяет толщину пластинки. Параметр **Скос** задаёт фаску, которая применяется ко всем граням (рис. 9.26).

Рис. 9.26

Эти операции фактически представляют собой модификаторы, применяемые к контуру. Это значит, что контур по-прежнему можно свободно редактировать. Чтобы сделать из такого объекта сеточную модель, нужно нажать клавиши Alt+*C* и выбрать во всплывающем меню команду **Полисетка из кривой** (Mesh from Curve). Если теперь выделить объект и перейти в режим редактирования узлов, мы увидим «обычную» сетку из узлов, рёбер и граней.

Профили

Часто нужно смоделировать кабель, трубу или брусок, имеющий заданный профиль сечения. В программах трёхмерной графики для этого обычно используются две кривые: одна задаёт профиль сечения, а вторая — путь. На рисунке 9.27 показаны кривые, с помощью которых построена модель балки с сечением в форме тавра.

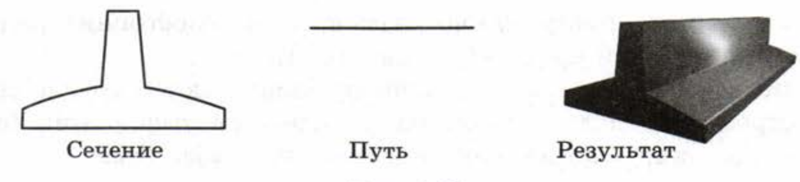


Рис. 9.27

Для того чтобы связать два контура в программе Blender, нужно выделить контур-путь и перейти на страницу данных объекта (Object Data). Затем в поле **Форма скоса** (Bevel object) из списка существующих контуров выбирается имя контура, задающего профиль. В некоторых программах, например в 3ds Max, такой приём называется **«лофтинг»** (англ. *lofting*).

Объект, который мы видим, строится только при выводе на экран. В памяти он хранится как две кривые, а не как сеточная модель. Поэтому можно как угодно изменять форму пути и профиля, но нельзя работать с отдельными вершинами, рёбрами и гранями.

К такому объекту можно применять модификаторы, например **Подразделить** (Subsurf) для получения гладкой поверхности. Можно также преобразовать его в сеточную модель с помощью клавиш Alt+C.

Тела вращения

В жизни нас окружает множество объектов, которые могут быть построены как тела вращения: тарелки, стаканы, бокалы, вазы и т. п. Для их моделирования также можно использовать профили, но в данном случае путь — это окружность. На рисунке 9.28 показано, как построить трёхмерную модель тарелки.

При создании профиля на окружности оказывается опорная точка кривой, определяющей сечение (начало локальных коорди-

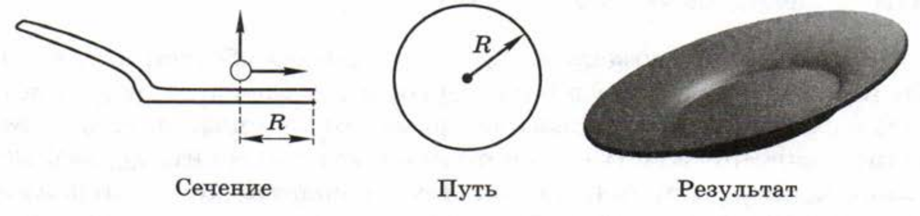


Рис. 9.28

нат), поэтому эту точку нужно размещать на расстоянии радиуса окружности от края кривой (см. рис. 9.28).

Отметим, что в программе Blender существуют и другие способы построения тел вращения, например инструмент **Spin** (англ. *spin* — вращение), применяемый к сеточным моделям.

Вопросы и задания

1. Зачем используются кривые в программах трёхмерного моделирования?
2. Из каких элементов состоит кривая Безье? Как изменять форму такой кривой?
3. Какие типы узлов используются при построении кривых Безье? Как изменить тип узла?
4. Как сделать видимую нить или трубу с помощью кривых?
5. Как создать пластину?
6. Как используются кривые для моделирования объектов с заданным профилем сечения?
7. Что нужно сделать, чтобы можно было изменять положение отдельных вершин такой модели?
8. Как смоделировать тело вращения?

Задачи

1. Постройте трёхмерную модель логотипа программы Blender, используя описанный метод создания пластин.
2. Постройте трёхмерную модель балки с сечением в форме двутавра (рисунок справа).
3. Постройте трёхмерную модель чашки или вазы.

§ 71
Материалы и текстуры

В природе нет объектов, которые были бы абсолютно гладкими и ровно покрашены в один серый цвет. Сцена не будет смотреться реалистично, пока мы не применим **материалы**, т. е. не зададим какие-то свойства, по которым можно отличить дерево, металл, мрамор, кирпич, песок. Без использования материалов невозможно сделать тела блестящими, прозрачными, светящимися и т. п.

Отражение света

Для того, чтобы разобраться со свойствами материалов, нужно понять, как мы видим окружающие предметы. Какой-то источник света (солнце, лампочка и т. п.) излучает световые волны, которые попадают на объекты. При этом часть волн поглощается материалом, а остальные отражаются от поверхности. Глаз воспринимает попадающие в него отражённые световые волны, длины их волн определяет видимый цвет предмета. Например, если источник излучает «белый» свет (включающий волны всех частот видимого светового диапазона), а мы видим зелёную поверхность, это означает, что материал поглощает все волны, кроме тех, которые соответствуют зелёному цвету.

Из курса физики вам известен закон отражения света, согласно которому угол отражения равен углу падения. Такое отражение называется **зеркальным** (*англ.* specular), при этом предполагается, что поверхность идеально ровная (рис. 9.29). Однако на самом деле любой материал имеет шероховатости, различающиеся по размеру. Поэтому лучи света отражаются от большинства предметов во всех направлениях, такое отражение называется **рассеянным**, или **диффузным** (англ. *diffuse*). Именно диффузное отражение определяет цвет объекта, который мы видим (см. рис. 9.29).

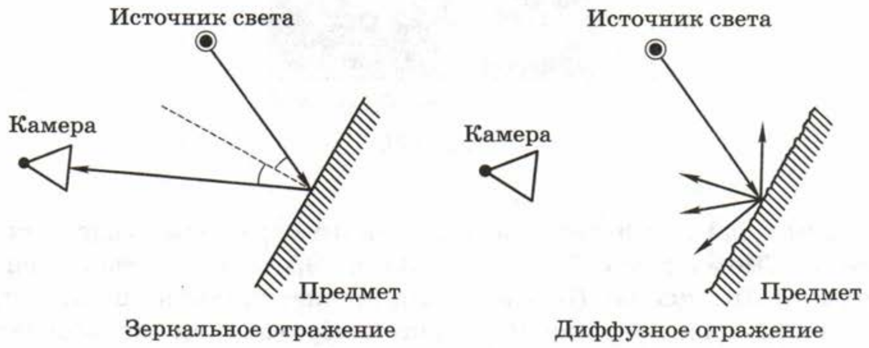


Рис. 9.29

Тип отражения зависит от степени шероховатости. Если размеры неровностей соизмеримы с длиной световой волны, происходит зеркальное отражение. Если неровности значительно больше длины волны, происходит рассеяние света (диффузия) и в глаз (или в съёмочную камеру) попадают лучи, отражённые от всех точек поверхности.

Простые материалы

Для объектов трёхмерной сцены можно построить сколько угодно различных материалов, выбирая для каждого нужные свойства. Материалы можно использовать повторно, т. е. материал одного объекта можно применять к другим объектам.

При настройке материала в первую очередь задаются два цвета — цвет диффузного (рассеянного) отражения и цвет зеркального отражения (цвет бликов). В программе Blender для изменения свойств материала используется страница свойств **Материал** (Material). В окне **Предпросмотр** (Preview) показан один из тестовых объектов, к которому применён выбранный материал (рис. 9.30).

Рис. 9.30

Цвета для диффузного и рассеянного отражения задаются на панелях **Диффузный** (Diffuse) и **Блик** (Specular) соответственно, параметр **Интенсивно** (Intensity) определяет яркость цвета. В правой части каждой панели можно выбрать один из **шейдеров** (*англ.* shader) — так называют алгоритмы, с помощью которых рассчитывается цвет каждой точки изображения. По умолчанию в Blender используются алгоритмы **Ламберт** (Lambert) для диффузного отражения и **Кук-Торренс** (CookTorr) для зеркального. Включив флажок **Градиентная карта** (Ramp), можно задать **градиент** — плавный переход между двумя или несколькими цветами. Параметр **Жёсткость** (Hardness) определяет размытость бликов, чем он больше, тем более резкая граница у блика.

С помощью панели **Прозрачность** (Transparency) можно сделать материал полупрозрачным. Такой материал пропускает часть падающих на него лучей света, например красное стекло пропускает только красные лучи, а остальные поглощает.

Панель **Отражение** (Mirror) позволяет получить на предмете отражения окружающих объектов.

Многокомпонентные материалы

Разным граням (полигонам) одного объекта можно присвоить разные материалы. В этом случае объект должен быть связан с несколькими материалами, список которых находится на странице свойств **Материал** (Material). На рисунке 9.31 показан случай, когда для объекта используются три материала с именами **Red**, **Green** и **Blue**. В данном примере выбран и редактируется материал **Red**.

Рис. 9.31

Если перейти в режим редактирования (когда можно работать с отдельными гранями), появляются три кнопки:

- **Присвоить** (Assign) — присвоить выбранный материал выделенным граням;
- **Выделение** (Select) — выделить грани, для которых установлен выбранный материал;
- **Снять выделение** (Deselect) — отменить выделение граней, для которых установлен выбранный материал.

Текстуры

При создании фотореалистичных изображений часто используются не простые одноцветные материалы, а так называемые **текстуры** — точечные (растровые) изображения, которые накладываются на поверхность для изменения окраски или имитации рельефа (рис. 9.32).

Текстура обычно относится к какому-то материалу, причем с каждым материалом можно связать несколько текстур (так же, как с одним объектом можно связать несколько материалов).

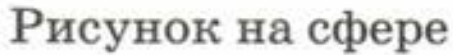


Рисунок на сфере Имитация рельефа

Рис. 9.32

Текстуры можно разделить на два типа: готовые изображения и так называемые **процедурные текстуры,** которые строятся по различным математическим алгоритмам. В окне свойств есть страница **Текстуры** (Texture), где для материала, выбранного на странице **Материал** (Material), можно задать одну или несколько текстур.

При создании новой текстуры по умолчанию выбирается тип **Облака** (Clouds). Это одна из стандартных процедурных текстур. Чтобы загрузить текстуру из файла на диске, нужно выбрать тип **Изображение или фильм** (Image or Movie), а затем, щёлкнув на кнопке **Открыть** (Open), выбрать нужный файл на диске.

В простейшем случае для наложения рисунка на объект текстура загружается в цветовой канал **Диффузный** (Diffuse), который определяет «нормальный» цвет объекта. Кроме того, текстуры можно использовать для канала **Блик** (Specular), альфа-канала (прозрачность), рельефа. Эти режимы задаются на панели **Влияние** (Influence), где нужно отметить параметры, на которые влияет текстура. Кроме того, степень воздействия текстуры можно регулировать, например смешивать установленный для материала цвет и текстуру в некоторой пропорции.

Для того чтобы наложить рисунок на поверхность, каждой точке этой поверхности нужно сопоставить определённый пиксель рисунка. Это фактически означает переход к другой системе координат. Способ такого преобразования координат задаётся на панели **Отображение** (Mapping). В списке **Координаты** (Coordinates) по умолчанию установлен вариант **Сгенерировать** (Generated), т. е. построить автоматически (рис. 9.33). В списке **Проекция** (англ. *Projection*) выбирается форма тела: **Плоское** (Flat), **Куб** (Cube), **Сфера** (Sphere) или **Трубка** (Tube).

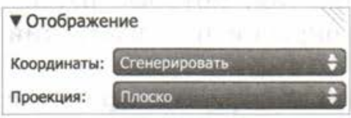


Рис. 9.33

UV-проекция

Пользователь может вручную определить, как именно точки рисунка будут проецироваться на поверхность. Для этого применяется так называемая **UV-проекция**, или **UV-развёртка** (UV unwrap).

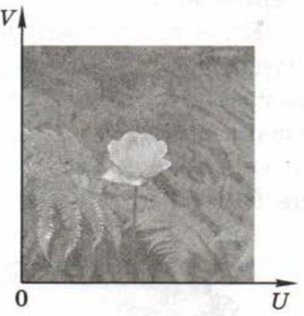


Рис. 9.34

Чтобы вручную задать способ наложения текстуры на грани объекта, используется специальная система координат *UV*, которая похожа на стандартную прямоугольную систему *XY* на плоскости. Оси *U* и *V* аналогичны осям *X* и *Y*, но относятся не к трёхмерной модели, а к текстуре (рис. 9.34). Построить *UV*-проекцию означает сопоставить каждой точке (x, y, z) поверхности объекта некоторую точку текстуры (u, v).

Для каждой грани можно настраивать *UV*-проекцию отдельно с помощью специального окна **Редактор UV-изображений** (UV/Image Editor). Для этого в окне трёхмерной проекции **(3D View)** надо перейти в режим редактирования сетки (клавиша Tab) и выделить нужную грань. Затем следует выбрать пункт меню **Полисетка – UV-развёртка – Развернуть** (Mesh – UV Unwrap – Unwrap), и в окне редактора UV-проекций появляется плоская сетка, повторяющая форму грани (рис. 9.35).

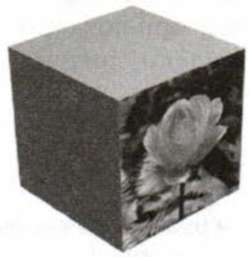
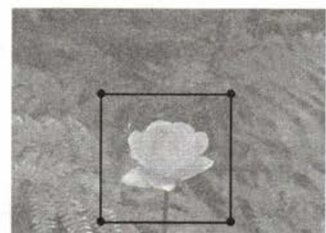


Рисунок на грани куба



UV-проекция

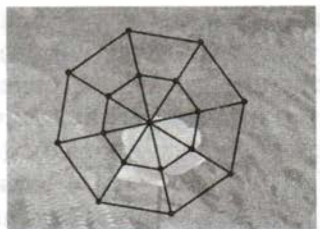
Рис. 9.35

Положение вершин и рёбер такой сетки можно изменять так же, как и положение вершин и рёбер трёхмерной сеточной модели в режиме редактирования. С помощью этого приёма можно, например, вырезать из одной и той же текстуры разные рисунки для каждой грани.

Для того чтобы при построении изображения использовались координаты, заданные пользователем, а не построенные автоматически, в списке **Координаты** (Coordinates) на панели **Отображение** (Mapping) нужно выбрать вариант **UV** вместо **Сгенерировать** (Generated).

Если граней много, возникают некоторые сложности. В этом случае удобно выделить в режиме редактирования сразу несколько граней объекта, тогда в окне редактора UV-проекций будет показана вся соответствующая им сетка (рис. 9.36). Сетку и её части можно перемещать по текстуре (клавиша *G*), вращать (клавиша *R*) и масштабировать (клавиша *S*).

Рисунок на зонтике



UV-проекция

Рис. 9.36

Программа Blender позволяет построить развёртку поверхности сложного объекта, указав, где нужно сделать разрезы. После этого на такой развёртке можно рисовать текстуру для готовой модели. Такой приём широко используется при создании персонажей в играх: по готовой развёртке модели рисуют текстуру (кожу, волосы, шкуру, одежду и т. д.).

Вопросы и задания

1. Расскажите о различиях диффузного и зеркального отражения света. Почему чаще всего нужно учитывать оба варианта?
2. Приведите примеры материалов, у которых практически нет диффузного или зеркального отражения.
3. Что такое шейдер? Зачем нужны разные типы шейдеров?
4. Как регулируется размытость бликов?
5. Что такое многокомпонентные материалы? Зачем они используются?
6. Что такое текстуры? Зачем они используются?
7. Что такое процедурные текстуры? В чём их достоинства и недостатки?
8. Что означает выражение «UV-проекция»?
9. Как наносят текстуру на сложные объекты?

Задачи

1. Попробуйте применять различные шейдеры для одного и того же материала и посмотрите, как меняется внешний вид объекта.
2. Примените различные текстуры к объектам разной формы (кубу, сфере, цилиндру).
3. Создайте плоскость, разбейте её на 4 клетки и присвойте каждой клетке свой цвет.
4. Нарисуйте 6 разных изображений на одном рисунке и назначьте их разным граням куба с помощью UV-развёртки.
5. Постройте модель зонтика и нанесите на него рисунок с помощью UV-развертки.

§ 72
Рендеринг

Рендеринг — это построение готового изображения: проекции трёхмерной сцены на плоскость с учётом материалов, текстур, освещённости, свойств внешней среды и т. п.

Для этого нужно после подготовки трёхмерной модели:

- расставить и настроить источники света;
- установить камеру, которая будет «снимать» сцену, и настроить её свойства;
- определить свойства внешней среды (цвет неба, туман и т. п.);
- выполнить рендеринг (нажать клавишу F12);
- сохранить готовое изображение с помощью клавиши F3.

Результат рендеринга появляется в окне **Редактор UV-изображений** (UV/Image Editor). В этом окне есть так называемые **слоты** (англ. *slot* — позиция, ячейка), в каждом из которых можно хранить одно изображение. Это позволяет запомнить несколько результатов рендеринга, полученных в разных условиях, а затем сравнить их.

Источники света

Источники света в Blender называются **лампами** (англ. *lamp*). Существует несколько типов ламп, различающихся по своим свойствам.

По умолчанию вновь созданная сцена содержит источник типа **Точка** (Point) — точечный источник света, лучи от которого расходятся во все стороны (радиально) — рис. 9.37.

Рис. 9.37

Освещённость зависит от расстояния между источником и поверхностью (согласно законам физики, она обратно пропорциональна квадрату этого расстояния). На рисунке 9.38 показаны элементы, позволяющие настраивать основные свойства лампы.

Рис. 9.38

В верхней части панели **Лампа** (Lamp) находятся кнопки для выбора типа источника света, который можно в любой момент изменить.

Параметр **Энергия** (Energy) определяет силу света. Чуть выше расположено поле выбора цвета. По умолчанию цвет лампы белый, однако в жизни крайне редко встречаются источники, излучающие белый свет (т. е. волны сразу всех длин светового диапазона). Для изменения цвета нужно щёлкнуть мышью в этом поле и выбрать нужный цвет из палитры.

Группа элементов **Затухание** (Falloff) определяет затухание света в зависимости от расстояния. Установленный по умолчанию метод **Обратно-квадратичный** (Inverse Square) лучше всего соответствует законам физики.

Параметр **Расстояние** (Distance) определяет расстояние (в условных единицах), на котором интенсивность света уменьшается в два раза. Если отметить флажок **Сфера** (Sphere), за пределами этого расстояния объекты не будут освещаться вообще.

Отключив флажок **Блик** (Specular), получаем источник, не дающий зеркального отражения. Флажок **Диффузный** (Diffuse) отключает рассеянное отражение света. Если отметить флажок **Только этот слой** (This layer only), источник не будет освещать объекты, которые находятся на других слоях. При включённом флажке **Инвертировать** (Negative) лампа не освещает, а затемняет поверхности.

Теперь рассмотрим другие типы ламп. Их основные параметры аналогичны настройкам лампы типа **Точка** (Point).

Солнце (Sun) — источник, моделирующий направленный солнечный свет (рис. 9.39). Поскольку **Солнце** находится от нас на очень большом расстоянии, солнечные лучи можно считать параллельными.

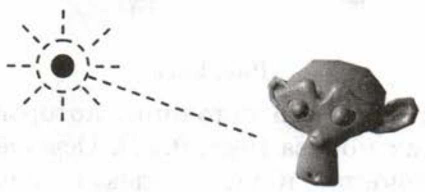


Рис. 9.39

Освещённость в этом случае не зависит от расположения лампы на сцене (в том числе от расстояния от лампы до объекта), а зависит только от направления лучей. Поэтому лампу типа **Солнце** можно ставить в любом месте сцены.

Полусфера (Hemi, от англ. *hemisphere* — полусфера) — источник, моделирующий рассеянный свет от большой полусферы (рис. 9.40). Такой световой поток содержит лучи разных направлений, поэтому освещение получается значительно мягче, чем при солнечном свете. Источники типа **Полусфера** можно использовать для подсветки теневых частей объектов. При освещении объекта таким источником падающей тени от объекта не будет.

Так же как и для лампы типа **Солнце**, освещённость не зависит от расположения лампы на сцене (в том числе от расстояния

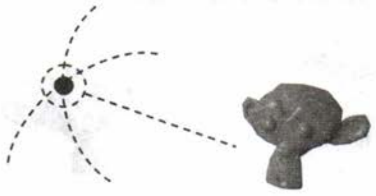


Рис. 9.40

от лампы до объекта), а зависит только от направления лучей. Такую лампу тоже можно ставить в любом месте сцены.

Область (Area) — это источник направленного света от прямоугольной площадки (рис. 9.41). Освещённость зависит от расстояния до источника и угла падения луча на поверхность. Его можно использовать, например, для создания подсветки от экрана телевизора.

Рис. 9.41

Прожектор (Spot) — это источник, который даёт направленный свет в пределах конуса (рис. 9.42). Освещённость зависит от расстояния до источника и угла падения луча на поверхность. Форма пятна может быть круглой или прямоугольной. Световой конус можно сделать видимым, используя эффект **«гало»** (англ. *halo* — ореол, сияние).

Рис. 9.42

Камеры

Камеры (рис. 9.43) — это специальные объекты, которые позволяют посмотреть на сцену с разных точек, как через видоискатель фотоаппарата или видеокамеры.

Рис. 9.43

По умолчанию на сцене находится одна камера. Если камера выделена, её можно перемещать (клавиша *G*) и вращать (клавиша *R*), как и любой объект. Нажав клавишу 0 на цифровой клавиатуре (*Num*0), можно переключиться на вид с камеры, при этом часть сцены, не попавшая в «поле зрения» камеры, затемняется.

Если включён вид с камеры, удобно настраивать его в **«режиме полёта»**, который включается при нажатии клавиш Shift+*F*. Колёсико мыши приближает и удаляет камеру от объекта, а движение мыши поворачивает её в соответствующем направлении. Настроив вид, нужно завершить процедуру щелчком левой кнопкой мыши.

Ещё один вариант — получить нужный вид в окне проекции и затем поставить камеру в найденную таким образом точку наблюдения, нажав клавиши Ctrl+Alt+*Num*0.

Параметры камеры задаются на странице свойств **Камера** (Camera). В режиме **Перспективный** (Perspective) камера снимает изображение с учётом перспективы, а в режиме **Ортогональный** (Orthographic) строит ортогональную проекцию (рис. 9.44).

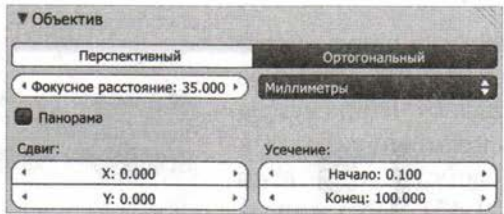


Рис. 9.44

Параметр **Фокусное расстояние** (Angle) соответствует фокусному расстоянию объектива фотоаппарата. По умолчанию используется фокус 35 мм, который примерно соответствует углу зрения человека.

Флажок **Панорама** (Panorama) предназначен для съёмки панорамных сцен.

С помощью полей группы **Сдвиг** (Shift) можно сдвинуть «поле зрения» камеры, не меняя её положение на сцене.

Параметры группы **Усечение** (Clipping) определяют область видимости камеры. Все объекты, находящиеся ближе расстояния **Начало** (Start) и дальше расстояния **Конец** (End), камера «не видит», и на итоговой картинке их не будет.

На сцене можно использовать несколько камер, переключаясь между ними. Чтобы поместить на сцену новую камеру, нужно вы-

брать пункт меню **Добавить – Камера** (Add – Camera). Клавиши Ctrl+*Num*0 делают выделенную камеру активной, т. е. при рендеринге будет построено изображение именно с этой камеры.

Камеру можно «привязать» к какому-то объекту сцены, т. е. сделать так, чтобы она была всё время направлена на этот объект. Для этого на странице свойств **Ограничения объекта** (Odject Constraints) можно добавить ограничение **Слежение** (англ. *Track To*, от *track* — следовать, сопровождать[1]), указав в качестве объекта-цели тот объект, на который нужно направить камеру. После этого при любых перемещениях камера будет всегда направлена на этот объект. Такая связь сохранится и при анимации — камера будет следить за объектом, если он движется.

Если в точке, куда должна смотреть камера, нет никакого объекта, можно добавить на сцену пустой объект (меню **Добавить – Пустышка** (Add – Empty). Пустой объект можно передвигать, так же как и другие объекты, но он не отображается на сцене при рендеринге.

Внешняя среда

Кроме источников света, установленных на сцене, на итоговое изображение влияют параметры внешней среды, которые задаются на странице свойств **Мир** (World) — рис. 9.45). По умолчанию для сцены используется серый фон, который можно изменить с помощью поля **Цвет горизонта** (Horizon Color). **Цвет окружения** (Ambient Color) задаёт цвет теней (по умолчанию чёрный).

Рис. 9.45

Можно сделать градиентный фон — переход между двумя цветами, один из которых — цвет горизонта, а второй — **Цвет зенита** (Zenith Color). Для этого необходимо отметить флажок **Смесь неба** (Blend Sky). При включённом флажке **Реальное небо** (Real Sky) фон зависит от точки установки камеры: на уровне горизонта цвет совпадает с цветом горизонта, а выше и ниже горизонта переходит в цвет зенита. Если включить флажок **Псевдонебо** (Paper Sky), цвет горизонта всегда будет располагаться по центру камеры, независимо от того, как она расположена.

---

[1] В других программах это ограничение называется **Look At** («смотреть на»).

В качестве фона можно установить растровый рисунок. Для этого нужно отменить выделение всех объектов на сцене и добавить новую текстуру на странице свойств **Текстуры** (Texture).

Кроме того, на странице свойств **Мир** есть панели **Туман** (Mist) и **Звёзды** (Stars), с помощью которых можно создать эффекты тумана и звёзд на небе.

Параметры рендеринга

Перед тем как выполнять рендеринг (нажав на клавишу F12), нужно определить, какое именно изображение вам требуется. Для этого на странице свойств **Рендер** (Render) задаются следующие параметры:

- **разрешение** (англ. *resolution*) — размеры получаемой картинки в пикселях (по умолчанию 1920 × 1080 пикселей);
- **масштаб** в процентах[1]; при увеличении масштаба время расчёта сцены также увеличивается; для предварительного просмотра обычно достаточно установить масштаб 25% (рис. 9.46);

Масштаб 10%

Масштаб 25%

Масштаб 100%

Рис. 9.46

- **сглаживание** острых граней (англ. *anti-aliasing*) (рис. 9.47);
- **тип изображения:**
  - BW — чёрно-белое (от англ. *black and white*);
  - RGB — цветное (цветовые каналы: Red — красный, Green — зелёный и Blue — синий);
  - RGBA — цветное с альфа-каналом, определяющим степень прозрачности; нужно помнить, что альфа-канал можно сохранять только в некоторых форматах (например, в PNG);

[1] При уменьшении масштаба программа уменьшает размер получаемого изображения, т. е. при масштабе 10% размер картинки при стандартных настройках будет 192 × 108 пикселей.

Рис. 9.47

- **формат файла** для сохранения (по умолчанию — PNG, часто выбирают другой популярный формат — JPEG);
- **дополнительную информацию** (англ. *stamp* — штамп), которая «впечатывается» в картинку: дату и время, название файла, время рендеринга и др.

Тени

В реальном мире все предметы отбрасывают тени. В программах трёхмерного моделирования для построения теней используются **алгоритмы трассировки лучей.** Это значит, что программа «запускает» большое количество лучей света от источника и просчитывает эффект, который даёт каждый луч при проходе через среду и отражении от граней, встречающихся на его пути. Такие расчёты требуют значительных вычислительных ресурсов, поэтому по умолчанию тени не строятся.

Чтобы включить построение теней, нужно настроить источники освещения и параметры рендеринга. Прежде всего, в свойствах источника света на панели **Тень** (англ. *Shadow*), показанной на рис. 9.48, надо включить режим **Трассировка теней** (англ. *Ray Shadow*).

▼Тень
Без теней | Трассировка теней
Только этот слой
Только для теней
Сэмплинг:
Сэмплов: 1 | Размер мягкого освещения: 1.000

Рис. 9.48

Цвет тени можно настроить с помощью поля в левой части панели (по умолчанию цвет теней чёрный). При включённом флажке **Только этот слой** (This layer only) лампа будет создавать тени только у тех объектов, которые находятся на том же слое, что и сама лампа. Флажок **Только для теней** (Only Shadow) позволяет сделать лампу, которая не освещает объекты, а только создаёт тени.

В реальности редко бывают чёткие тени с резкими краями. Поэтому используют смягчение теней, которое задаётся параметром **Размер мягкого освещения** (Soft Size). Увеличение параметра **Сэмплов** (Samples) позволяет сделать более равномерную и качественную тень, но это требует дополнительного времени при расчёте.

Кроме того, на странице параметров рендеринга **Рендер** на панели **Затенение** (Shadow) нужно отметить флажок **Трассировка лучей** (Ray Tracing).

Вопросы и задания

1. Что такое рендеринг? Что влияет на результат рендеринга?
2. Какие типы ламп есть в Blender? Зачем они используются?
3. Что такое камера?
4. Подумайте, зачем может понадобиться использовать несколько камер.
5. Почему по умолчанию для камеры установлено фокусное расстояние 35 мм?
6. Когда, на ваш взгляд, имеет смысл использовать ортогональную проекцию?
7. Как сделать, чтобы камера всегда была направлена в какую-то точку сцены? А если там нет никакого объекта?
8. Какие свойства внешней среды можно настраивать в *Blender*?
9. Что такое качество рендеринга? Как оно связано со временем рендеринга?
10. Объясните, почему при рендеринге качественных изображений необходимо сглаживание граней (anti-aliasing).
11. Что такое трассировка лучей? Почему эта операция очень трудоёмкая?

Задачи

1. Загрузите в программу какую-нибудь трёхмерную сцену и попробуйте менять освещение, устанавливая дополнительные лампы и настраивая их свойства.

2. Установите для лампы ограничение **Слежение** (Track To) и проверьте, что будет происходить при движении лампы.
3. Включите трассировку лучей и сравните результаты рендеринга с тенями и без них.

§ 73 Анимация

Анимация объектов

Анимация — это быстрая смена изображений, которые называются **кадрами** (англ. *frames*). Если кадры сменяют друг друга чаще, чем 24 раза в секунду, человеческий глаз воспринимает это как непрерывное движение.

Для работы с кадрами в Blender используется окно **Линия времени** (Timeline). По умолчанию на шкале показаны первые 250 кадров (рис. 9.49). **Курсор** (зелёная линия, перемещается мышью), показывает текущий кадр, который изображается в окне трёхмерной проекции.

Рис. 9.49

Под шкалой размещаются поля, в которых записаны номера начального (англ. *start*), конечного (англ. *end*) и текущего кадров анимации (рис. 9.50).

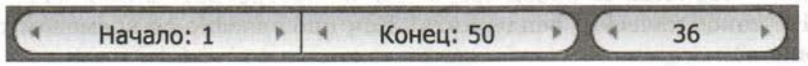


Рис. 9.50

Эти три значения можно ввести с клавиатуры или изменить с помощью стрелок по сторонам полей. Здесь же расположена кнопочная панель управления: кнопки для просмотра (▷ — вперёд, ◁ — назад) и перехода между ключевыми кадрами (рис. 9.51).

Рис. 9.51

Создание анимации в программах трёхмерной графики очень похоже на ручное рисование мультфильмов: сначала строятся так называемые **ключевые кадры**, в которых положение и свойства объектов точно задаются. Затем программа автоматически достраивает оставшиеся (промежуточные) кадры.

Для анимации движения объекта нужно:

1) установить курсор на временно́й шкале на выбранный кадр;
2) задать положение объекта для этого кадра;
3) вставить ключевой кадр, нажав клавишу *I*;
4) повторить эти действия для всех ключевых кадров.

При добавлении ключевого кадра появляется меню, в котором требуется выбрать тип нового кадра. Он зависит от того, какие изменения происходят с объектом: изменение положения (Location), вращение (Rotation), масштабирование (Scaling) или их комбинации. Ключевые кадры обозначаются на временно́й шкале жёлтыми линиями. Чтобы удалить ключевой кадр, нужно сделать его текущим (установить на него курсор) и нажать клавиши Alt+*I*.

В программе Bledner можно анимировать не только перемещение объекта, но и изменение любого свойства, например цвета. Для вставки ключевого кадра изменения цвета нужно нажать правую кнопку мыши на поле установки цвета и выбрать из контекстного меню пункт **Вставить ключевые кадры** (Insert keyframes).

Для каждого свойства, которое участвует в анимации, устанавливаются свои ключевые кадры. Например, для поворота объекта могут быть выбраны ключевые кадры 10, 20 и 100, а для изменения цвета — кадры 1, 20, 50 и 70.

Кнопка справа от кнопочной панели предназначена для включения режима автоматической записи: при каждом изменении свойств объекта на место текущего кадра вставляется ключевой кадр.

Анимация в окне проекции запускается с помощью клавиш Alt+*A* (вперед) или Shift+Alt+*A* (назад). Повторное нажатие останавливает анимацию на текущем кадре. Остановить анимацию и вернуться к начальному кадру можно с помощью клавиши Esc.

Редактор кривых

Когда установлены ключевые кадры, все изменяемые свойства объекта в промежуточных кадрах рассчитываются автоматически. По умолчанию программа выполняет плавное изменение па-

раметров от одного значения к другому. Иногда нужно изменить такое поведение, например сделать переход, который резко начинается и плавно заканчивается, или скачок значения. Для этого в Blender есть возможность ручной настройки переходов между ключевыми кадрами.

Изменение любого параметра (например, координаты или угла поворота) в зависимости от номера кадра можно изобразить на графике. Кривая на рис. 9.52 показывает изменение угла поворота вокруг оси X, рассчитанное программой.

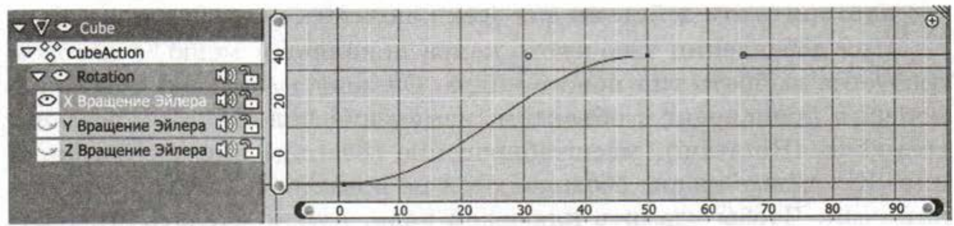


Рис. 9.52

Это окно называется **Редактор кривых** (Graph Editor). В левой части перечислены все параметры, которые изменяются в ходе анимации. В данном случае изменяются только углы поворота объекта **Cube** (ключевые кадры типа **Rotation**). Тип перехода (постоянное значение, линейный, плавный) можно выбрать для каждого узла из всплывающего меню, которое появляется при нажатии клавиш Shift+T.

С помощью флажков можно отключать любую из кривых, например на рис. 9.52 показано только изменение угла поворота вокруг оси X, остальные линии скрыты.

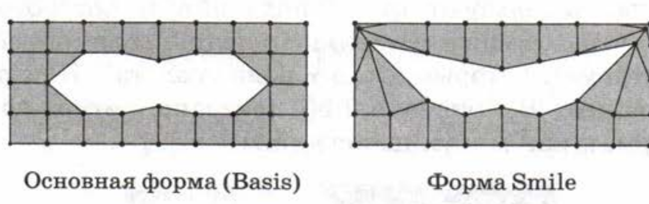
Кривую можно редактировать так же, как и обычный контур. Узлы кривой расположены в ключевых кадрах (на рисунке 9.52 это кадры 1 и 50), у выделенного узла показаны рычаги, с помощью которых можно менять кривизну линии. Узлы перемещаются с помощью правой кнопки мыши (щелчок левой кнопкой заканчивает перемещение) или клавиши G. Ключевые кадры (узловые точки) каждой кривой можно устанавливать независимо от других кривых.

Щелчок на значке с изображением глаза отключает изменение соответствующего параметра при анимации, а с помощью значка «замок» можно заблокировать кривую (защитить её от изменений).

Простая анимация сеточных моделей

Выше мы говорили только об изменении свойств объектов в целом. При создании анимационных фильмов необходимо уметь перемещать части сеточной модели, например, для того, чтобы персонаж моргнул глазами. Для этого используются **ключи формы**, или **ключевые формы** (англ. *shape keys*), — так называются некоторые заранее заданные положения сеточной модели, между которыми выполняется переход. Есть одно важное ограничение: при создании таких ключевых форм нельзя менять геометрию модели, т. е. удалять и добавлять вершины, рёбра и грани.

Сначала нужно создать сами ключевые формы и определить положение узлов сетки для каждой из них (в Blender для этого используется панель **Ключи формы** (Shape keys) на странице свойств сеточной модели **Данные объекта** (Object Data). Например, для анимации улыбающегося рта нужно построить, по крайней мере, две ключевые формы: рот без улыбки (основная форма, которая называется **Basis**) и рот с улыбкой (назовем её *Smile*) (рис. 9.53).



Основная форма (Basis) Форма Smile

Рис. 9.53

Единственное различие этих форм в том, что вершины формы *Smile* передвинуты в другое положение.

В программе Blender удобнее всего использовать окно **Редактора ключей формы** (ShapeKey Editor), в левой части которого перечислены все ключевые формы и показаны коэффициенты (от 0 до 1), определяющие степень влияния каждой формы на текущий кадр (рис. 9.54).

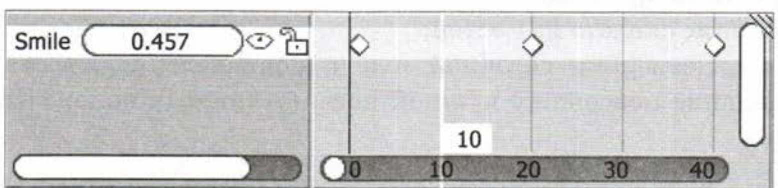


Рис. 9.54

Ключевые кадры в правой части обозначены маркерами-ромбами. Чтобы добавить новый ключевой кадр, достаточно установить на него курсор (светлую линию) и изменить значение параметра в левой части. Ключевые кадры можно перемещать (клавиша *G*) и масштабировать (регулировать интервал между кадрами, клавиша *S*). Для удаления ключевого кадра нужно выделить соответствующий ему маркер-ромб и нажать клавишу Delete.

Можно использовать не один, а несколько каналов анимации, при этом ключевые кадры каждого канала задаются независимо от других каналов. Форма кривой изменения параметра настраивается в окне **Редактора кривых.**

Арматура

Анимация с помощью ключевых форм хороша тогда, когда нужно изменить положение небольшого числа вершин сеточной модели. Во многих случаях, например при повороте шеи или сустава руки персонажа, необходимо передвинуть сотни вершин. В этом случае используют другой подход, суть которого состоит в том, что внутрь объекта вставляют специальные объекты («кости», «арматуру»), которые играют роль скелета[1]. При рендеринге кости не видны. На рисунке 9.55 показана фигура шахматного короля с арматурой в двух положениях.

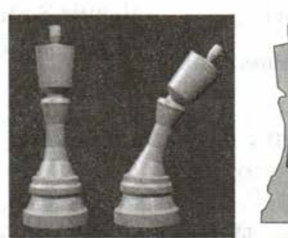
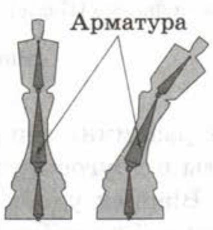



Рис. 9.55

Обычно выделяют три этапа моделирования персонажа с использованием арматуры:

1) создание скелета из костей;
2) привязка вершин сеточной модели к определённым костям;
3) придание персонажу нужной позы (установка положения костей).

[1] В английском языке эта процедура называется rigging (от англ. *rig* — оснастка).

На втором этапе арматуру, которая, как правило, состоит из нескольких связанных костей, нужно сделать родительским объектом для объекта-оболочки, установив между ними связь (клавиши Ctrl+*P*). В отличие от простой связи «объект — объект», когда преобразования объекта-родителя применяются ко всем потомкам, здесь устанавливается особая связь «арматура — оболочка».

Программа автоматически определяет, какие именно вершины сеточной модели оболочки попадают в «зону влияния» каждой из костей и будут перемещаться вслед за ней. Существует специальный режим **Оболочка** (Envelope), в котором можно увидеть и редактировать эти зоны влияния. Режим **Оболочка** включается на панели свойств арматуры (**Данные объекта**). При необходимости можно вручную назначить ведущую кость для каждой вершины. Для этого вершины объединяются в группы, названия которых должны совпадать с названиями объектов-костей.

Таким образом, для того чтобы изменить форму объекта, достаточно изменить положение костей. Вслед за костями переместятся и все связанные с ними вершины сеточной модели, а их могут быть тысячи!

Прямая и обратная кинематика

На рисунке 9.56 показана арматура, которую можно использовать для моделирования руки человека. Кости *А*, *Б* и *В* управляют соответственно плечом, предплечьем и кистью персонажа.

Рис. 9.56

Обычно в модели эти кости связаны отношениями «родитель — потомок»: кость *А* — это родитель для *Б*, а *Б* — родитель для *В*. Перемещение родителя приводит к перемещению всех потомков, т. е. при перемещении кости *А* кости *Б* и *В* перемещаются вслед за ней. Такая связь называется **прямой кинематикой** (англ. *forward kinematics*).

В некоторых случаях прямая кинематика затрудняет построение анимации. Например, пусть персонажу нужно взять что-то в руку. При этом мы знаем точно положение кисти *В*, тогда как положения остальных костей (*А* и *Б*) должны измениться соответственно, чтобы сохранилась связь в цепочке костей. Это значит, что перемещение кости *В* приводит к согласованному перемещению костей *А* и *Б*. Такая связь называется **обратной кинематикой**

(англ. *inverse kinematics*), потому что движение передается в обратную сторону.

Для построения связи «обратная кинематика» на кость *Б* нужно наложить ограничение **Обратная кинематика** (Inverse kinematics). В Blender для этого используется специальная страница свойств **Ограничения кости** (Bone constraints), которая открывается в режиме просмотра **Режим позы** (Pose mode). В параметрах этого ограничения нужно указать длину цепочки костей, на которую действует ограничение (параметр Chain length, длина цепи). В рассмотренном случае длина цепочки равна 2 (кости *А* и *Б*). Отметим, что в Blender кость *В* не должна быть связана ни с какой родительской костью.

Физические явления

Современные программы трёхмерной графики содержат средства для моделирования физических процессов. Это значит, что при построении анимации тела перемещаются с учётом законов физики: дым поднимается вверх или стелется по ветру, вода капает или стекает вниз, тела сталкиваются между собой и отскакивают, ткань полощется на ветру и т. п. В этих моделях для создания реалистичной анимации используются достаточно сложные математические уравнения, их решение требует большого объёма вычислений и мощного компьютера.

В программе Blender для этой цели используют две панели свойств: **Частицы** (Particles) и **Физика** (Physics). С их помощью можно моделировать:

- системы частиц (дым, огонь, волосы, траву);
- жидкости;
- столкновения тел;
- силовые поля, например ветер и магнитное поле;
- ткань.

Математические модели, описывающие эти явления, уже заложены в программу. Изменяя их параметры, можно получать самые разнообразные эффекты.

Система частиц — это модель объекта, который не имеет чётких границ, например пара, огня, дыма, дождя, снега и т. п. Для таких объектов невозможно построить сеточную модель, поэтому они моделируются как поток маленьких частиц, каждая из которых имеет скорость, цвет, направление движения. При движении частиц учитываются свойства внешней среды — сила тяго-

тения (гравитация), ветер и т. п. Для настройки системы частиц используется множество параметров, меняя которые, можно строить модели различных явлений (дым, огонь и т. д.).

В Blender есть особый тип системы частиц — **«волосы»** (*англ.* hair). Они могут использоваться при моделировании волос человека, шерсти животных, а также травы (рис. 9.57).

Рис. 9.57

Жидкость (англ. *fluid*) моделируется как поверхность с большим числом граней, причём она может состоять из многих отдельных частей (капель). Для того чтобы построить анимацию жидкости, для каждого кадра положение всех вершин рассчитывается на основе предыдущего кадра и физических свойств жидкости (вязкости). Такой пересчёт иногда называют **«выпечкой»** (англ. *bake*). Он занимает длительное время, которое зависит от установленного разрешения (англ. *resolution*) и количества кадров анимации. При этом сеточные модели для каждого кадра создаются на диске в каталоге для временных файлов. При повторном просмотре анимации они уже не пересчитываются заново, а загружаются с диска (этот подход называют *кэшированием*). Однако если вы измените какой-либо параметр, всю анимацию придётся просчитывать заново с самого начала.

Ткань (англ. *cloth*) — это тоже сложная поверхность. В отличие от жидкости ткань создаётся и разбивается на грани вручную (например, используется плоскость, разбитая на части с помощью инструмента **Подразделить** (Subdivide)).

При построении анимации программа рассчитывает для каждого кадра положение всех вершин сеточной модели (падающей ткани) с учётом свойств выбранного типа ткани (хлопок, шёлк, кожа и др.). Чем мельче грани модели, тем точнее моделирование, но и больше время расчёта положения ткани в каждом кадре. Для объектов, которые должны задерживать ткань, нужно

установить свойство **«столкновение»** (англ. *collision*). Примеры использования ткани в трёхмерных моделях показаны на рис. 9.58. Для создания модели полощущегося флага использовалось силовое поле типа **«ветер»** (англ. *wind*).

Рис. 9.58

Мягкие тела (англ. *soft bodies*) — это специальный аппарат для реалистичного моделирования движения тел, обладающих упругостью. Без него (вручную) было бы практически невозможно грамотно построить анимацию, в которой предметы сталкиваются, сминаются, отскакивают друг от друга и т. п.

Когда для сеточной модели устанавливается свойство **«мягкое тело»**, грани приобретают свойства пружинок. Их жёсткость можно регулировать с помощью настроек на панели **Мягкое тело** (Soft Body).

Рендеринг

Конечный результат анимации — это видеоролик, записанный в одном из видеоформатов, например AVI[1], MPEG, QuickTime, Ogg Theora. Рендеринг происходит покадрово, т. е. сначала программа строит полное изображение кадра 1, затем — кадра 2 и т. д. Важная характеристика видео — частота кадров (англ. **FPS** — *frames per second*, кадры в секунду). Обычно в кино используется частота 24 кадра в секунду[2], при которой человеческий глаз воспринимает смену кадров как непрерывное движение. В этом случае для создания ролика, длящегося 10 секунд, нужно построить анимацию из 240 кадров.

---

[1] Строго говоря, формат AVI — это *контейнер*, внутри которого видео и звук могут быть упакованы с помощью различных кодеков (алгоритмов сжатия).

[2] В телевизионном стандарте PAL, который принят в Европе, используется частота 25 кадров с секунду, а в стандарте NTSC (США, Канада) — около 30 кадров в секунду.

Рендеринг сложных сцен, включающих миллионы граней, может занимать значительное время (дни и недели!) и требовать больших вычислительных мощностей компьютера. Прежде всего, важно быстродействие процессора и объём оперативной памяти. Поэтому рендеринг видеороликов выполняется, как правило, на нескольких мощных компьютерах, объединённых в локальную сеть. Существуют организации, предоставляющие такие услуги **(рендер-фермы)**[1].

В программе Blender режимы рендеринга настраиваются на странице свойств **Рендер.** На панели **Вывод** (Output) нужно выбрать каталог для сохранения видеоролика, формат и имя файла. На панели **Размеры** (Dimensions) задаются размеры изображения, номера начального и конечного кадров анимации, а также частота кадров. По умолчанию установлена частота 24 кадра в секунду.

Рендеринг запускается с помощью кнопки **Анимация** (Animation) на странице свойств **Рендер** или комбинацией клавиш Ctrl+F12.

Вопросы и задания

1. Расскажите об основных принципах анимации по ключевым кадрам.
2. Как можно изменять поведение объектов в промежуточных кадрах?
3. Объясните, как выполняется анимация с ключевыми формами.
4. Что такое арматура и зачем она нужна?
5. Сравните анимацию по ключевым формам и анимацию с помощью арматуры. Когда удобно использовать каждый из этих способов?
6. Объясните различие между связями «прямая кинематика» и «обратная кинематика».
7. Расскажите о возможностях моделирования физических процессов в программах трёхмерной графики.
8. Что такое система частиц и зачем она используется?
9. Как строится анимация ткани?
10. Вспомните, где ещё в компьютерной технике используется кэширование.
11. Что такое «мягкие тела»?
12. Почему рендеринг видеороликов представляет собой серьёзную проблему? Как она может быть решена?

---

[1] Например, http://farmerjoe.info и http://renderfarm.fi

Задачи

1. Постройте простую анимацию одного из объектов-примитивов, изменяя его положение, углы поворота и размеры.
2. Постройте анимацию улыбающегося рта.
3. Постройте фигуру шахматного короля, управляемого арматурой. Создайте небольшую анимацию, в которой король наклоняется в разные стороны.

*4. Добавьте волосы на голову обезьянки с помощью системы частиц.
*5. Постройте анимацию прыгающего теннисного мяча.
*6. Постройте анимацию флага, полощущегося на ветру.

§ 74 Язык VRML

Как вы уже знаете, трёхмерные сцены хранятся в компьютере как векторные изображения. Каждый объект в векторном формате задаётся координатами своих вершин и свойствами (например, характеристиками материала). Эти данные могут храниться во внутреннем («машинном») представлении, однако для ручной обработки (и для передачи через Интернет) удобнее, если 3D-модель представляет собой обычный текст без оформления (англ. *plain text)*.

В этом параграфе мы кратко познакомимся с **языком VRML** (Virtual Reality Modeling Language — язык моделирования виртуальной реальности), который позволяет сохранить трёхмерную сцену в текстовом файле и потом просматривать её в специальной программе или в веб-браузере (с помощью дополнительного модуля — *плагина*). Язык VRML «понимают» многие программы трёхмерного моделирования, в том числе системы автоматизированного проектирования (САПР). С помощью VRML сделаны виртуальная экскурсия по мемориалу на Мамаевом кургане[1], виртуальный выставочный центр на сайте www.3dexpo.ru и 3D-модели исторических событий (например, полета в космос Юрия Гагарина) на сайте фирмы Parallel Graphics[2].

[1] http://www.volgograd.ru/mamayev-kurgan/

[2] http://www.parallelgraphics.com/products/showroom/event/

С помощью VRML можно описать не только форму трёхмерных объектов, но и их физические свойства: цвет, текстуру, блеск, прозрачность. Объекты могут быть гиперссылками на другие документы. Язык VRML позволяет создавать анимацию, менять освещение, включать и выключать звуки, а также добавлять к сцене программный код на языках Java или JavaScript.

Трёхмерную сцену в VRML называют **миром** (англ. *world*), поэтому VRML-файлы имеют расширение wrl. Это обычные текстовые файлы, которые можно редактировать в любом текстовом редакторе. Кроме того, существуют специальные VRML-редакторы, например WhiteDune[1].

www

Для просмотра VRML-моделей нужно установить соответствующий плагин к браузеру (например, Cortona3D Viewer[2]) или самостоятельное приложение (например, кроссплатформенную программу view3dscene[3]).

Построим с помощью VRML трёхмерную модель комнаты. Её стены (а также пол и потолок) можно представить в виде плит (блоков, параллелепипедов). Пусть длина каждой стены — 4 м, высота — 3 м, а толщина — 0,1 м. Тогда стена в плоскости XOY (рис. 9.59) описывается так:

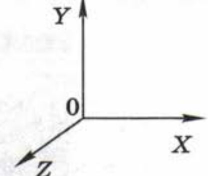


Рис. 9.59

```
#VRML V2.0 utf8
Shape
  {
  geometry Box { size 4 3 0.1 }
  }
```

В первой строке указана версия языка VRML (2.0) и кодировка текста (utf-8 — это один из вариантов UNICODE). У единственного безымянного объекта `Shape` (англ. *shape* — форма, фигура) задано только одно свойство — `geometry` (геометрия). Слово `Box` означает, что эта фигура — параллелепипед («коробка»), его размеры (по осям X, Y и Z соответственно) указаны после слова `size` (размер).

Объекты **сцены** в VRML называются **узлами**. Названия классов объектов начинаются с заглавной буквы, а названия свойств **(полей)** — со строчных. В нашем примере объект класса `Shape` имеет поле `geometry`, определяющее форму тела. Значение этого

1 http://vrml.cip.ica.uni-stuttgart.de/dune/

2 http://www.cortona3d.com/cortona

3 http://vrmlengine.sourceforge.net/view3dscene.php

поля — объект класса `Box` (параллелепипед)[1]. Узел `Box`, в свою очередь, имеет поле `size`, определяющее размеры плиты по каждой координатной оси.

Кроме параллелепипедов в языке VRML есть объекты других классов, например `Sphere` (шар), `Cylinder` (цилиндр), `Cone` (конус). Более сложные фигуры строятся из отдельных граней.

Объекты VRML имеют довольно много полей. Если значение поля не указано, используется некоторое стандартное значение (значение по умолчанию). Например, по умолчанию тело располагается так, чтобы его центр совпадал с началом координат, и равномерно «покрашено» в белый цвет.

Приведённый выше VRML-код можно записать в файл (назвав его, например, `first.wrl`) и загрузить в программу-просмотрщик. При этом мы не просто увидим объёмную стену, но и сможем «походить» вокруг неё. В программе view3dscene это будет выглядеть так, как показано на рис. 9.60.

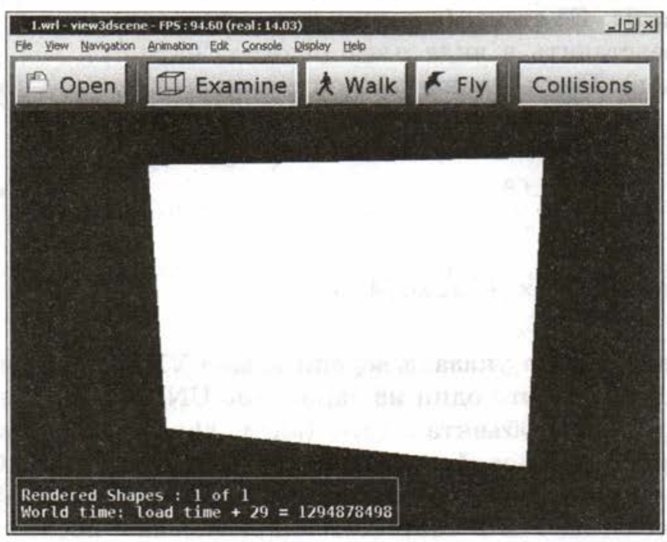


Рис. 9.60

Программа должна выполнять рендеринг для загруженной трёхмерной модели очень быстро, поэтому качество картинки будет существенно хуже, чем в профессиональных программах 3D-моделирования типа Blender или 3ds Max.

---

[1] Каждому объекту можно присвоить собственное имя, но делать это не обязательно.

В нашем файле точка наблюдения не задана, и программа самостоятельно выбирает её (по умолчанию). Положение этой точки можно задать явно, добавив в VRML-файл код:

```
Viewpoint
  {
  position 0 0 5
  }
```

Здесь свойство position (положение) объекта Viewpoint (точка наблюдения) определяет начальные координаты наблюдателя. В данном случае он находится в точке с координатами $X = 0$, $Y = 0$ и $Z = 5$.

Существует несколько режимов просмотра трёхмерной сцены:

- **Examine** (рассматривать, исследовать) — наблюдатель неподвижен, объект можно поворачивать и рассматривать под разными углами;
- **Walk** (ходить, делать обход) — сцена неподвижна, а наблюдатель перемещается внутри неё «по поверхности земли»;
- **Fly** (летать) — отличается от предыдущего режима «выключенной» силой тяжести.

Кнопка **Collisions** (столкновения) определяет, может ли наблюдатель проходить сквозь стены и предметы (если она отключена, то может).

Чтобы придать стене более реальный вид, надо заполнить у объекта Shape еще одно поле — appearance («внешность», «наружность»), отвечающее за то, как объект выглядит. Все свойства, влияющие на внешний вид объекта, собраны в единый объект с именем Appearance. Зададим для стены материал (свойство material объекта Appearance) и установим для этого материала цвет (свойство diffuseColor):

```
#VRML V2.0 utf8
Shape
  {
  geometry Box { size 4 3 0.1 }
  appearance Appearance
    {
    material Material { diffuseColor 0.7 0.7 0.7 }
    }
  }
```

Числа после названия свойства diffuseColor задают цвет стены в виде трёх составляющих модели RGB — красной, зелёной и синей, каждая из которых принимает значение от 0 до 1 (это соответствует диапазону от 0 до 255 при целочисленном RGB-кодировании цвета). В данном случае все они равны 0,7 — это светло-серый цвет.

Вместо того чтобы «красить» стену, можно было «оклеить» её обоями, т. е. наложить рисунок (текстуру). Для этого нужно задать свойство texture (текстура):

```
#VRML V2.0 utf8
Shape
  {
  geometry Box { size 4 3 0.1 }
  appearance Appearance
    {
    texture ImageTexture { url["texture.png"] }
    }
  }
```

В данном случае текстура относится к классу ImageTexture (текстура-рисунок) и находится в файле texture.png в текущем каталоге. Рисунок растягивается на всю поверхность, поэтому его пропорции должны соответствовать соотношению размеров стены.

Остальные стены строятся аналогично, но с одним существенным отличием: их придётся смещать в пространстве (иначе по умолчанию их центр будет расположен в начале координат). Предположим, что созданная стена будет «дальней от нас». Тогда «левая» стена имеет размеры 0,1 × 3 × 4 м (по осям X, Y и Z соответственно), и её нужно передвинуть на 2,05 м влево вдоль оси X и на 1,95 м «в сторону наблюдателя» вдоль оси Z (величина 0,05 м — это половина толщины стены!).

Для перемещения объектов в виртуальном пространстве используется узел Transform (превращать, изменять). Он способен выполнять для присоединённых к нему узлов (перечисленных в списке children — «потомки», «подчинённые объекты») следующие действия (и любые их комбинации!):

- translation (смещение) — смещение центра объекта вдоль каждой из трёх осей координат;
- rotation (вращение, поворот) — поворот объекта вокруг трёх осей координат;
- scale (изменение масштаба) — умножение размеров объекта на коэффициенты, отдельно по каждой оси.

В нашем случае требуется только перемещение. В список children (он записывается в квадратных скобках) мы включим одну фигуру (объект Shape) — параллелепипед, изображающий левую стену. Этот код нужно добавить в конец VRML-файла:

```
Transform
  {
  translation -2.05 0 1.95
  children
    [
    Shape
      {
      geometry Box { size 0.1 3 4 }
      }
    ]
  }
```

Теперь вы можете самостоятельно построить «виртуальную комнату» целиком.

Вопросы и задания

1. Как вы понимаете термин «виртуальная реальность»?
2. Какие свойства трёхмерных объектов можно моделировать с помощью VRML?
3. Что представляет собой VRML-файл? Какое расширение он имеет и почему?
4. Какое программное обеспечение требуется для работы с VRML?
5. Сравните системы координат, которые используются в математике и в VRML 2.0.
6. Объясните, что такое сцена, узлы и поля. Приведите примеры.
7. Что такое значение по умолчанию? Какие преимущества даёт такой способ задания значений при передаче VRML-модели по сети?
8. Каково по умолчанию положение тел относительно начала координат? Что нужно сделать, чтобы изменить положение объекта?
9. Перечислите режимы просмотра трёхмерных сцен. Чем они различаются?
10. Как называется узел VRML, отвечающий за внешний вид объектов?
11. Каким образом кодируется цвет в языке VRML?
12. Как нанести текстуру на поверхность объекта?
13. Какие действия можно выполнять с помощью узла Transform?
14. Обсудите, где можно использовать язык VRML.

Задачи

1. Сделайте комнату замкнутой: опишите все четыре стены, пол и потолок. Поскольку по умолчанию точка наблюдения будет вне комнаты, подумайте, как попасть внутрь (используйте кнопку **Collisions**).
2. Подготовьте и наложите на каждую стену отдельную текстуру.
3. Напишите программу, которая создаёт VRML-файл по введённым размерам стен комнаты.

*4. Напишите программу, которая создаёт VRML-файл с описанием шахматной доски, состоящей из 64 чередующихся чёрных и белых блоков (объектов `Box`).

5. Постройте простейший лабиринт из нескольких коридоров. Пройдите его от точки входа до точки выхода. Используя режим полёта (**Fly**), посмотрите на лабиринт сверху.
6. Используя комбинацию простейших геометрических тел, попробуйте создать какие-нибудь простые объёмные предметы. Например, конус и пара цилиндров позволяют «построить» ракету, а из сфер разного радиуса можно создать модель планетной системы.
7. Используя блоки (параллелепипеды), постройте объёмные буквы «Г», «Е» и «Ш».

*8. Найдите информацию о полях узла `Material` и посмотрите, как их значения влияют на изображение объекта.

*9. Найдите информацию об узле `Transform`. Примените режимы `rotation` и `scale`.

*10. Напишите VRML-код, который строит снеговика.

Практические работы к главе 9

Работа № 77 «Управление сценой»
Работа № 78 «Работа с объектами»
Работа № 79 «Сеточные модели»
Работа № 80 «Модификаторы»
Работа № 81 «Пластина»
Работа № 82 «Тела вращения»
Работа № 83 «Материалы»
Работа № 84 «Текстуры»
Работа № 85 «UV-развёртка»
Работа № 86 «Рендеринг»
Работа № 87 «Анимация»
Работа № 88 «Анимация. Ключевые формы»
Работа № 89 «Анимация. Арматура»
Работа № 90 «Язык VRML»

ЭОР к главе 9 на сайте ФЦИОР (http://fcior.edu.ru)

- Изображения. Виды
- Общие понятия об аксонометрических проекциях
- Проекции простейших геометрических фигур на плоскость

Самое важное в главе 9

- Трёхмерные сцены строятся с помощью векторной графики.
- Трёхмерные модели объектов состоят из отдельных граней (полигонов), чаще всего треугольных или четырёхугольных. Каждый полигон ограничен рёбрами.
- Для каждой грани можно независимо задавать свойства материала — цвет, блики, шероховатость и т. п. Можно наносить на грани рисунок — текстуру.
- Рендеринг — это построение плоского изображения трёхмерной сцены с учётом материалов, текстур, освещённости, свойств внешней среды. При этом выполняются сложные математические расчёты хода большого количества лучей от источников света, поэтому для рендеринга сложных сцен нужен быстродействующий процессор и большой объём оперативной памяти.
- Анимация трёхмерных сцен строится по кадрам: человек вручную определяет положение объектов в ключевых кадрах, а все промежуточные кадры строятся автоматически.
- Для хранения трёхмерных сцен можно использовать как двоичные, так и текстовые файлы.

Для заметок

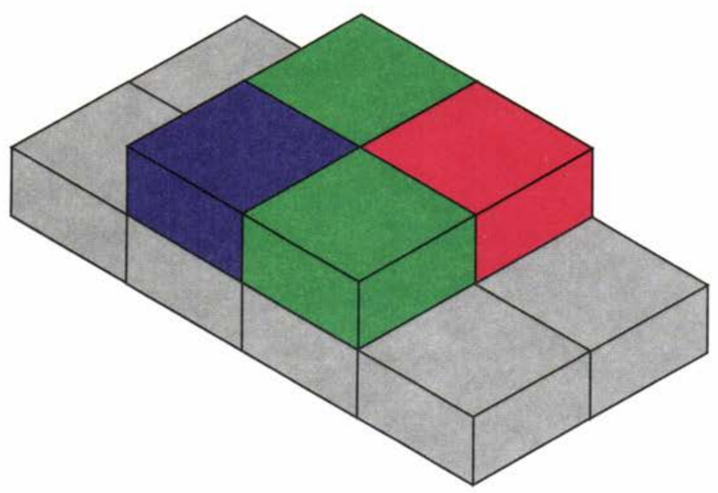
Для заметок

Для заметок

Для заметок

Для заметок

Фильтр Байера



Цветовые каналы модели RGB