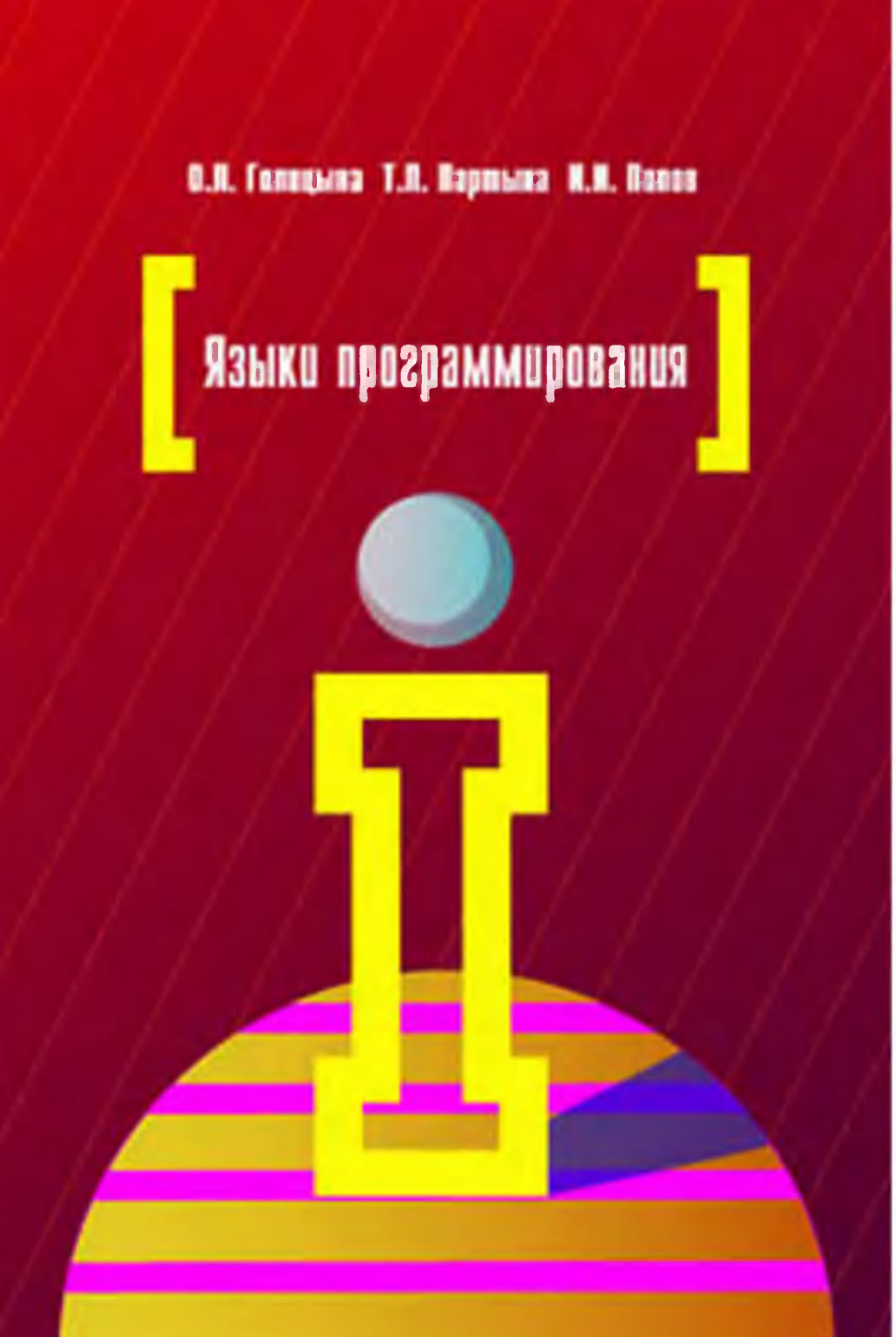


О. Л. Голицына, Т. Л. Партыка,
И. И. Попов.

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Допущено Министерством образования и науки Российской Федерации в качестве учебного пособия для студент образовательных учреждений среднего профессионального образования

Москва
ФОРУМ — ИНФРА-М

УДК 004.2(075.32)
ББК 32.973-02я723
П57

Рецензенты:
доктор технических наук, профессор, зав. кафедрой
информационных систем в экономике и менеджменте
РЭА им. Г. В. Плеханова *В. П. Романов*;
председатель цикловой (предметной) комиссии
математического колледжа, преподаватель *В. П. Агальцов*

Голицына О. Л., Партыка Т. Л., Попов И. И.
П57 Языки программирования: учеб. пособие. — М.: ФОРУМ: ИНФРА-М, 2008. — 400 с.: ил. — (Профессиональное образование).

ISBN 978-5-91134-171-8 (ФОРУМ)
ISBN 978-5-16-003200-9 (ИНФРА-М)

Рассмотрены кодирование и представление информации в ЭВМ, структуры алгоритмов, эволюция и классификация языков, основные понятия, связанные с разработкой программ. Дается описание конкретных языков и систем программирования — Basic, Pascal, Delphi, FoxPro.

Для учащихся и студентов, специализирующихся в области вычислительных устройств, машин и общей информатики.

УДК 004.2(075.32)
ББК 32.973-02я723

ISBN 978-5-91134-171-8 (ФОРУМ)
ISBN 978-5-16-003200-9 (ИНФРА-М)

Введение

Вплоть до XVII в. деятельность общества в целом и каждого человека в отдельности была направлена на овладение веществом, т. е. познание свойств вещества и изготовление сначала примитивных, а потом все более сложных орудий труда, вплоть до механизмов и машин, позволяющих изготовлять потребительские ценности.

Затем в процессе становления индустриального общества на первый план вышла проблема овладения энергией — сначала тепловой, затем электрической, наконец, атомной. Овладение энергией позволило освоить массовое производство потребительских ценностей и, как следствие, повысить уровень жизни людей и изменить характер их труда.

В то же время человечеству свойственна потребность выразить и запомнить информацию об окружающем мире — так появились письменность, книгопечатание, живопись, фотография, радио, телевидение. В истории развития цивилизации можно выделить несколько информационных революций — преобразование общественных отношений из-за кардинальных изменений в сфере обработки информации, информационных технологий. Следствием подобных преобразований являлось приобретение человеческим обществом нового качества.

В конце XX в. человечество вступило в новую стадию развития — стадию построения информационного общества. Информация стала важнейшим фактором экономического роста, а уровень развития информационной деятельности и степень вовлеченности и влияния ее на глобальную информационную инфраструктуру превратились в важнейшее условие конкурентоспособности страны в мировой экономике. Понимание неизбежности прихода этого общества наступило значительно раньше. Австралийский экономист К. Кларк еще в 40-е гг. говорил о

наступлении общества информации и услуг, общества с новыми технологическими и экономическими возможностями. Американский экономист Ф. Махлуп выдвинул предположение о наступлении информационной экономики и превращении информации в важнейший товар в конце 50-х гг. В конце 60-х гг. Д. Белл фиксировал превращение индустриального общества в информационное. Что касается стран, ранее входивших в СССР, то процессы информатизации в них развивались замедленными темпами.

Информатизация изменяет всю систему общественного производства и взаимодействия культур. С приходом информационного общества начинается новый этап развития. Меняется вся система информационных коммуникаций. Разрушение старых информационных связей между отраслями экономики, направлениями научной деятельности, регионами, странами усилило экономический кризис конца века в странах, которые уделяли развитию информатизации недостаточное внимание. Важнейшая задача общества — восстановить каналы коммуникации в новых экономических и технологических условиях для обеспечения четкого взаимодействия всех направлений экономического, научного и социального развития как отдельных стран, так и в глобальном масштабе.

В качестве средства для хранения, переработки передачи информации научно-технический прогресс предложил обществу компьютер (электронно-вычислительную машину — ЭВМ). Но вычислительная техника не сразу достигла необходимого уровня. В ее развитии отмечают предысторию и четыре поколения ЭВМ. Предыстория начинается в глубокой древности с различных приспособлений для счета (абак, счеты), а первая счетная машина появилась лишь в 1642 г. Ее изобрел французский математик Б. Pascal. Построенная на основе зубчатых колес, она могла суммировать десятичные числа. Все четыре арифметических действия выполняла машина, созданная в 1673 г. немецким математиком Г. Лейбницем. Она стала прототипом арифмометров, использовавшихся вплоть до 1960-х гг.

Впервые идея программно управляемой счетной машины, имеющей арифметическое устройство, устройства управления, ввода и печати (хотя и использующей десятичную систему счисления), была выдвинута в 1822 г. английским математиком Ч. Бэббиджем.

Программные средства ЭВМ являются одним из важнейших факторов информатизации, наряду с такими, как аппаратное обеспечение (технические средства обработки, передачи, ввода-вывода и хранения данных), информационное обеспечение (файлы, базы данных и информационные ресурсы) и человеческий фактор (пользователи средств информатизации, включая администраторов, операторов и рядовых пользователей).

Программные средства в свою очередь подразделяются на:

- операционную систему (ОС), которая обеспечивает функционирование и взаимосвязь всех компонентов компьютера и предоставляет пользователю доступ к его аппаратным возможностям;
- прикладное программное обеспечение (ППО), которое также можно далее разделить на две группы программ — средства разработки и приложения.

Средства разработки — это инструменты программиста. Традиционными средствами разработки являются системы (среды) программирования (СП), использующие алгоритмические (процедурные) языки программирования (ЯП). Основой систем программирования являются трансляторы, т. е. программы, обеспечивающие перевод исходного текста программы на машинный язык (объектный код), которые подразделяются на интерпретаторы и компиляторы.

Приложения (программные оболочки, средства пользователя) представляют собой программные продукты или пакеты прикладных программ (ППП), ориентированные в основном на непрограммирующего пользователя и реализующие определенную группу функций или информационных технологий — работу с документами, мультимедийными материалами, осуществление коммуникации и пр.

В настоящем учебном пособии речь идет о средствах разработки программных продуктов — различного рода системах программирования более или менее широкой ориентации, базирующихся на различных ЯП.

В 1-й главе рассмотрены информационные основы и представление информации в ЭВМ, кодирование символьной, цифровой информации; логические и алгоритмические основы программного обеспечения — алгоритмы, структуры алгоритмов. Описывается программирование в машинных адресах и ассемб-

леры, рассмотрена эволюция и классификация языков и систем программирования, а также основные понятия, связанные с разработкой и развитием программного обеспечения.

Во 2-й главе дается описание языка программирования Basic, в том числе примеры программ на ЯП Basic. Рассматриваются переменные и типы данных, операции и операторы языка.

В главе 3 рассматривается ЯП Pascal, в том числе примеры простых программ на ЯП Pascal, форматы языка программирования Pascal, переменные и константы, типы данных. Далее рассматриваются выражения и операции, операторы языка, структурированные типы данных, динамические данные, процедуры и функции. Рассматриваются компоненты структуры программы, методы организации ввода-вывода данных и работы с файлами.

В главе 4 речь идет об основных принципах объектно-ориентированного программирования, в основном на примере интегрированной среды разработки приложений Delphi (объектно-ориентированное расширение ЯП Pascal). Рассматриваются интерфейс среды Delphi, характеристика проекта Delphi, компиляция и выполнение проекта, разработки приложения. Описываются средства управления параметрами интегрированной среды разработки, связь между ЯП Pascal и визуальной средой разработки приложений Delphi. В качестве стандартного примера рассматривается разработка приложения Калькулятор в средах Delphi и Visual Basic.

В 5-й главе рассматривается система FoxPro, являющаяся «пограничным продуктом» между СУБД и системами программирования. Описываются типы данных, команды и операторы языка, создание и модификация базы данных, создание и модификация форматов представления данных.

Настоящее учебное пособие базируется на материалах, которые авторы накопили в процессе практической, исследовательской, а также преподавательской (МИФИ, МИСИ, РГГУ, РЭА им. Г. В. Плеханова, МФПА) деятельности. Авторы выражают благодарность коллегам, принявшим участие в обсуждении материала, рецензентам, а также студентам РГГУ и РЭА им. Г. В. Плеханова за предоставленные иллюстративные материалы.

Глава 1
ОСНОВНЫЕ ПРИНЦИПЫ ПРОГРАММИРОВАНИЯ

Безусловно, наряду с любой другой целесообразной человеческой деятельностью, программирование может рассматриваться как средство самовыражения личности или способ приятного времяпрепровождения в компании единомышленников. Соглашаясь с этим «где-то в глубине души», авторы однако, предполагают относиться к программированию, как к компоненту колоссальной информационной индустрии, в которой вращаются миллиарды долларов и заняты миллионы человек. В настоящей главе планируется рассмотреть основные составляющие программирования, связанные с ними понятия, принципы функционирования, а также место программного обеспечения (ПО) в общей совокупности инструментария информатики. Они включают: информационные основы, логические основы, алгоритмические основы и методы разработки программных продуктов.

1.1. Информационные основы программирования. Представление информации в ЭВМ

Понятие «информация» является таким же фундаментальным как понятия «материя», «энергия» и другие философские категории. Это атрибут, свойство сложных систем, связанное с их развитием и самоорганизацией [16]. Известно большое количество различных определений информации, отличия информации от данных, знаний и пр. Мы здесь ограничимся только рассмотрением некоторых практически важных понятий и определений.

Классификация информации

С точки зрения обработки информации на компьютерах информация может классифицироваться, например, по признакам, отражающим структуру данных и форму представления информации (табл. 1.1).

Таблица 1.1. **Некоторые классы информации (по структуре и форме)**

Основание для классификации	Информация			
По уровням сложности	Сигнал	Сообщение, документ	Информационный массив	Информационный ресурс
По типу сигнала	Аналоговая (непрерывная)	Цифровая (дискретная)		
По уровням доступа и организации	Данные в регистровой памяти ЭВМ	Данные в оперативной памяти ЭВМ	Файлы данных на внешних устройствах	Базы данных
По способам кодирования и представления (данные, файлы и базы данных — БД)	Цифровая (вычислительные данные, двоичные)	Символьная (алфавитно-цифровая, строчная)	Графическая	
По организации данных (файлы и БД)	Табличная	Текстовая	Графическая	

Измерение количества информации

Термин «информация» имеет корень «form» (форма), что разумно трактовать как «информирование — придание формы, вывод из состояния неопределенности, бесформенности», поэтому следует подходить к определению понятия «количество информации», исходя из того, что информацию, содержащуюся в сообщении, можно трактовать в смысле ее воздействия на объект, или, иначе, уменьшения неопределенности знаний «приемника информации» об объекте.

В свое время К. Шенноном в качестве единицы информации было предложено принять один б и т (от *англ.* BIT — **BI**nary digi**T** — двоичная цифра). Сегодня в вычислительной технике битом считается минимальная порция памяти компьютера, необходимая для хранения одного из двух знаков — «0» и «1», используемых для представления данных и команд.

Поскольку бит — слишком малая единица, на практике обычно применяется б а й т, равный восьми битам. В частности, восемь бит требуется для того, чтобы закодировать любой из 256 символов основного компьютерного кода ASCII ($256 = 2^8$).

Используются также более крупные производные единицы информации:

- килобайт (Кбайт, KB) = 1024 байт = 2^{10} байт;
- мегабайт (Мбайт, MB) = 1024 Кбайт = 2^{20} байт $\approx 10^6$ байт;
- гигабайт (Гбайт, GB) = 1024 Мбайт = 2^{30} байт $\approx 10^9$ байт.

С увеличением объемов обрабатываемой информации входят в употребление такие производные единицы, как:

- терабайт (Тбайт, TB) = 1024 Гбайт = 2^{40} байт $\approx 10^{12}$ байт;
- петабайт (Пбайт, PB) = 1024 Тбайт = 2^{50} байт $\approx 10^{15}$ байт;
- экзобайт = 10^{18} байт и пр.

Это так называемые «десятичные» единицы. В качестве альтернативной IEC (Международная электротехническая комиссия) предложила в 1998 г. «двоичные» единицы:

- KiB (KibiByte) — 2^{10} = 1024 байт;
- MiB (MibiByte) = 1024 KiB;
- GiB (GibiByte) = 1024 MiB (MibiByte) и т. д.

Кодирование символьной информации

Код (*code*) — совокупность знаков, символов и правил представления информации. Рассмотрим методы дискретного представления информации, или кодирования (которые, надо сказать, появились задолго до вычислительных машин). Первым широко известным примером является азбука Морзе (АМ), в которой буквы латиницы (или кириллицы) и цифры кодируются сочетаниями из «точек» и «тире» (табл. 1.2). Воспользуемся данным кодом для иллюстрации основных понятий, связанных с кодированием (не вдаваясь в теорию кодирования).

К о д и р у е м ы е (о б о з н а ч а е м ы е) элементы входного алфавита обычно называют с и м в о л а м и.

Символом (служит условным знаком какого-нибудь понятия, явления), как правило, является цифра, буква или иероглиф естественного языка, знак препинания, знак пробела, специальный

Таблица 1.2. **Фрагменты кода Морзе**

Символ входного алфавита	Мнемоническое обозначение по МСС*	Кодовая (знаковая) комбинация
A	alfa	.-
B	bravo	-...
C	charlie	-.-.
D	delta	-..
E	echo	.
...	...	...
Y	...	-.—
Z	yankee	—..
1	zulu	.——
...	...	...
9	nine	——.

* Международный Свод Сигналов.

знак, символ операции. Кроме этого, учитываются *управляющие («непечатаемые») символы.*

Кодирующие (обозначающие) элементы выходного алфавита называются знаками; количество различных знаков в выходном алфавите назовем *значностью* (*-арностью*, *-ичностью*, например «бинарный» или «двоичный» код); количество знаков в кодирующей последовательности для одного символа — *разрядностью кода.*

Пространственно-временно́е расположение знаков кода приводит к понятиям *параллельных* или *последовательных* кодов. При последовательном коде каждый временной такт предназначен для отображения одного разряда слова. Здесь все разряды слова фиксируются по очереди одним и тем же элементом и проходят через одну и ту же линию передачи (например, радио- или оптические сигналы либо передача данных по двум проводам, двухжильному кабелю).

При параллельном коде все знаки символа представляются в одном временном такте, каждый знак проходит через отдельную линию (например, по четырем проводам, четырехжильному кабелю), образуя символ (т. е. символ передается в один прием, в один момент времени).

Для последовательного кода характерно временно́е разделение каналов при передаче информации, для параллельного — пространственное. В зависимости от применяемого

кода различаются устройства параллельного и последовательного действия.

Применительно к азбуке Морзе:

- *символами* являются элементы языкового алфавита (буквы A—Z или А—Я) и цифровой алфавит (здесь это цифры 0—9);
- знаками являются «точка» и «тире» (или «+» и «–» либо «1» и «0», короче — два любых разных знака);
- поскольку знаков два, АМ является *двузначным* (бинарным, двоичным) кодом, если бы их было 3, то мы имели бы дело с троичным, тернарным, трехзначным кодом;
- поскольку число знаков в АМ колеблется от 1 (буквы Е, Т) до 5 (цифры), здесь имеет место код с *переменной разрядностью* (в АМ часто встречающиеся в тексте символы обозначены более короткими кодовыми комбинациями, нежели редкие символы).
- поскольку знаки передаются последовательно (электрические импульсы, звуковые или оптические сигналы разной длины, соответствующие «точкам» и «тире»), АМ есть *последовательный код.*

В табл. 1.3, 1.4 приводится перечень наиболее известных кодов, некоторые из них использовались первоначально для связи, кодирования данных, а затем для представления информации в ЭВМ.

Таблица 1.3. **Характеристики некоторых наиболее известных кодов**

Наименование кода	Расшифровка/перевод	Другие названия	Разрядность	Комментарий
Baudot	Код Бодо	IA-1 — international alphabet № 1	5	В прошлом — европейский стандарт для телеграфной связи
M2	MKKTT-2 CCITT-2	IA-2	5	Телеграфный код, предложенный Международным Комитетом по телефонии и телеграфии (MKKTT) и заменивший код Бодо
ASCII-7	American Standard Code for Information Interchange	ISO-7 IA-5, USASCII, ANSI X3.4	7	Код для передачи данных, поддерживает 128 символов, включающих прописные и строчные символы латиницы, цифры, специальные значки и управляющие символы. После добавления некоторых национальных символов (10 бинарных комбинаций) был принят Международной организацией по стандартизации (ISO) как стандарт ISO-7

Окончание табл. 1.3

Наименование кода	Расшифровка/перевод	Другие названия	Разрядность	Комментарий
ASCII-8	То же		8	Для внутреннего и внешнего представления данных в вычислительных системах. Включает стандартную часть (128 символов) и национальную (128 символов). В зависимости от национальной части, кодовые таблицы различаются
EBCDIC	Expanded Binary Coded Decimal Information Code		8	Предложен фирмой IBM для машин серий IBM/360-375 (внутреннее представление данных в памяти), а затем распространившийся и на системы других производителей
Hollerith	Код Холлерита	Код перфокарт (ПК), рис. 1.1	12	Предложен для ПК (1913 г.), затем использовавшийся для кодирования информации перед вводом в ЭВМ с ПК
UNICODE	UNIversal Code		16	Поскольку в 16-разрядном UNICODE можно закодировать 65 536 символов вместо 128 для ASCII, то отпадает необходимость в создании модификаций таблиц кодов. UNICODE охватывает 28 000 букв, знаков, слогов, иероглифов национальных языков мира

Таблица 1.4. **Фрагменты некоторых кодовых таблиц (указаны 16-ричные коды символов)**

Символ	IA-2	Бодо	ISO-7	EBCOIC	ASCII-8	Холлерит
A	03	10	41	C1	A1	900
B	19	06	42	C2	A2	880
C	0E	16	43	C3	A3	840
D	09	1E	44	C4	A4	820
a			61	81	E1	
b			62	82	E2	
c			63	83	E3	
d			64	84	E4	
. (точка)	1C	05	2E	4B	4E	842
, (запятая)	0C	09	2C	6B	4C	242
: (двоеточие)	1E		3B	5E	5B	40A
? (вопрос)	10	0D	3F	6F	5F	206

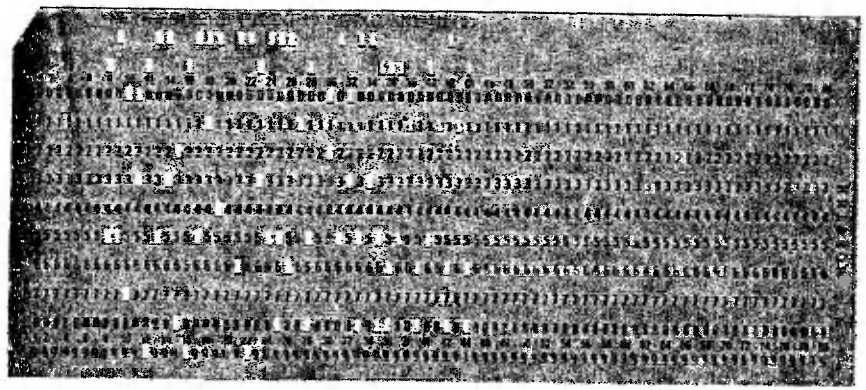


Рис. 1.1. Перфокарта Холлерита

Поскольку знаки передаются последовательно (электрические импульсы, звуковые или оптические сигналы разной длины, соответствующие «точкам» и «тире»), АМ есть последовательный код.

Кодирование и обработка чисел

Кроме кодирования символов, в ЭВМ очевидную важность имеют кодирование и представление чисел.

Системы счисления. Человек привык считать предметы десятками, сотнями и пр. Это — десятичная система счисления, которая не является единственно возможной (известна, например, двенадцатеричная система счисления).

Наиболее естественный способ представления числа в компьютерной системе заключается в использовании строки битов, называемой двоичным числом — числом в двоичной системе счисления.

Основание позиционной системы счисления — количество различных цифр (P), используемых для изображения числа в позиционной системе счисления. Значения цифр лежат в пределах от 0 до $P-1$.

В общем случае запись любого числа N в системе счисления с основанием P будет представлять собой ряд (многочлен) вида

$$N = a_{m-1} \times P^{m-1} + a_{m-2} \times P^{m-2} + \ldots + a_k \times P^k + \ldots$$

$$\ldots + a_1 \times P^1 + a_0 \times P^0 + \ldots + a_{-1} \times P^{-1} + a_{-2} \times P^{-2} + \ldots + a_{-s} \times P^s. \quad (1.1)$$

Нижние индексы определяют местоположение цифры в числе (разряд):

- положительные значения индексов — для целой части числа (m разрядов);
- отрицательные значения — для дробной (s разрядов).

Максимальное целое число, которое может быть представлено в m разрядах:

$$N_{max} = P^m - 1.$$

Минимальное значащее, не равное 0 число, которое можно записать в s разрядах дробной части:

$$N_{min} = P^{-s}.$$

Имея в целой части числа m разрядов, а в дробной — s, можно записать P^{m+s} разных чисел.

Двоичная система счисления (основание $P = 2$) использует для представления информации две цифры — 0 и 1.

Существуют простые правила перевода чисел из одной системы счисления в другую, основанные, в том числе, и на выражении (1.1).

Например, двоичное число 101110,101 равно десятичному числу 46,625:

$$101110{,}101_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 +$$
$$+ 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 46{,}625_{10}.$$

Практически перевод из двоичной системы в десятичную можно легко выполнить, если надписать над каждым разрядом соответствующий ему «вес», а затем и сложить произведения значений соответствующих цифр на их веса:

Вес	128	64	32	16	8	4	2	1
Цифра	0	1	0	0	0	0	0	1

Например, двоичное число 01000001_2 равно 65_{10}. Действительно, $64 \times 1 + 1 \times 1 = 65$.

Таким образом, для перевода числа из позиционной системы счисления с любым основанием в десятичную систему счисления можно воспользоваться выражением (1.1).

Обратный перевод из десятичной системы счисления в систему счисления с другим основанием непосредственно по (1.1)

затруднителен, поскольку все арифметические действия, предусмотренные этой формулой, следует выполнять в той системе счисления, в которую число переводится. Обратный перевод выполняется значительно проще, если предварительно преобразовать отдельно целую и дробную части выражения (1.1) к виду

$$N_{цел} = (((...(a_{m-1} \times P + a_{m-2}) \times P + ... + a_2) \times P + a_1) \times P + a_0;$$

$$N_{др} = P^{-1} \times (a_{-1} + P^{-1} \times (a_{-2} + P^{-1} \times (a_{-3} + ... + P^{-1} \times (a_{-s+1} + P^{-1} \times a_{-s})...))).$$

Алгоритм перевода числа из десятичной системы счисления в систему счисления с основанием P, основанный на этих выражениях, позволяет оперировать числами в той системе счисления, из которой число переводится, и может быть сформулирован следующим образом.

При переводе смешанного числа следует переводить его целую и дробную части отдельно.

1. Для перевода целой части числа ее, а затем целые части получающихся частных от деления следует последовательно делить на основание P до тех пор, пока очередная целая часть частного не окажется равной 0. Остатки от деления, записанные последовательно справа налево, образуют целую часть числа в системе счисления с основанием P.

2. Для перевода дробной части числа ее а затем дробные части получающихся произведений следует последовательно умножать на основание P до тех пор, пока очередная дробная часть произведения не окажется равной 0 или не будет достигнута нужная точность дроби. Целые части произведений, записанные после запятой последовательно слева направо, образуют дробную часть числа в системе счисления с основанием P.

Пусть требуется перевести смешанное число из десятичной в двоичную систему счисления на примере числа 46,625.

1. Переводим целую часть числа:

46 : 2 = 23 (остаток 0);
23 : 2 = 11 (остаток 1);
11 : 2 = 5 (остаток 1);
5 : 2 = 2 (остаток 1);
2 : 2 = 1 (остаток 0);
1 : 2 = 0 (остаток 1).

Записываем остатки последовательно справа налево — 101110, т. е. $46_{10} = 101110_2$.

2. Переводим дробную часть числа:

0,625 × 2 = 1,250;

0,250 × 2 = 0,500;

0,500 × 2 = 1,000 (дробная часть равна 0 ⇒ стоп).

Записываем целые части получающихся произведений после запятой последовательно слева направо — 0,101, т. е.:

$0{,}625_{10} = 0{,}101_2$.

Окончательно: $46{,}625_{10} = 101110{,}101_2$.

Кроме двоичной и десятичной в компьютерах используются также двоично-десятичная и шестнадцатеричная системы счисления (табл. 1.5).

Таблица 1.5. **Перевод цифр из двоичной системы счисления в восьмеричную, шестнадцатеричную, десятичную и обратно**

Триада	Восьмеричная цифра	Тетрада	Шестнадцатеричная цифра	Десятичное число	Двоично-десятичная запись
000	0	0000	0	0	0000–0000
001	1	0001	1	1	0000–0001
010	2	0010	2	2	0000–0010
011	3	0011	3	3	0000–0011
100	4	0100	4	3	0000–0100
101	5	0101	5	5	0000–0101
110	6	0110	6	6	0000–0110
111	7	0111	7	7	0000–0111
		1000	8	8	0000–1000
		1001	9	9	0000–1001
		1010*	A	10	0001–0000
		1011	B	11	0001–0001
		1100	C	12	0001–0010
		1101	D	13	0001–0011
		1110	E	14	0001–0100
		1111	F	15	0001–0101

* Запрещены в двоично-десятичном представлении.

Шестнадцатеричная система счисления часто используется при программировании. Перевод чисел из шестнадцатеричной системы счисления в двоичную весьма прост — он выполняется поразрядно.

Для изображения цифр, больших 9, в шестнадцатеричной системе счисления применяются буквы латиницы: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.

Так, например, шестнадцатеричное число F17B в двоичной системе выглядит как 1111000101111011, а в десятичной — как 61819.

Двоично-десятичная система счисления получила большое распространение в современных компьютерах ввиду легкости перевода в десятичную систему и обратно. Она используется там, где основное внимание уделяется не простоте технического построения машины, а удобству работы пользователя. В двоично-десятичной системе счисления основанием является число 10, но при этом каждая десятичная цифра (0, 1, ..., 9) кодируется четырьмя двоичными цифрами.

Двоично-десятичная система не экономична с точки зрения реализации технического построения машины (примерно на 20 % увеличивается требуемое оборудование), но более удобна при подготовке задач и при программировании.

Представление чисел в ЭВМ

В ЭВМ применяются две формы представления чисел:

- естественная, или форма с фиксированной запятой (точкой) — ФЗ (ФТ);
- нормальная, или форма с плавающей запятой (точкой) — ПЗ (ПТ).

Фиксированная запятая (точка). В форме представления с *фиксированной запятой (точкой)* числа изображаются в виде последовательности цифр с постоянным для всех чисел положением запятой, отделяющей целую часть от дробной.

Например, пусть числа представлены в десятичной системе счисления и имеют пять разрядов в целой части числа (до запятой) и пять — в дробной части (после запятой). Числа, записанные в такую разрядную сетку, имеют вид:

+00721,35500; +00000,00328; -10301,20260.

Эта форма наиболее проста, естественна, но имеет небольшой диапазон представления чисел и поэтому чаще всего неприемлема при серьезных вычислениях.

Диапазон значащих чисел N в системе счисления с основанием P при наличии m разрядов в целой части и s разрядов в дробной части числа (без учета знака числа) будет таким:

$$P^{-s} \le N \le P^m - P^{-s}.$$

Например, при $P = 2$, $m = 10$ и $s = 6$ числа изменяются в диапазоне $0{,}015 < N < 1024$. Если в результате операции получится число, выходящее за допустимые пределы, произойдет переполнение разрядной сетки, и дальнейшие вычисления теряют смысл.

В памяти ЭВМ числа с фиксированной запятой хранятся в трех форматах:

- полуслово — это обычно 16 бит или 2 байта;
- слово — 32 бита или 4 байта;
- двойное слово — 64 бита или 8 байтов.

Отрицательные числа с ФЗ записываются в разрядную сетку в дополнительных кодах, которые образуются прибавлением единицы к младшему разряду обратного кода. Обратный код получается заменой единиц на нули, а нулей на единицы в прямом двоичном коде.

Плавающая запятая (точка). В форме представления с плавающей запятой (точкой) число изображается в виде двух групп цифр:

- мантисса;
- порядок.

При этом абсолютная величина мантиссы должна быть меньше 1, а порядок должен быть целым числом. В общем виде число в форме с плавающей запятой может быть представлено так:

$$N = \pm M \times P^{\pm r},$$

где M — мантисса числа ($|M| < 1$); r — порядок числа (целое число); P — основание системы счисления.

Например, приведенные ранее числа в форме с ПЗ запишутся следующим образом:

$+0{,}721355 \times 10^{3}$;

$+0{,}328 \times 10^{-3}$;

$-0{,}103012026 \times 10^{5}$.

Форма представления с ПЗ обеспечивает значительный диапазон отображения чисел и является основной в современных компьютерах. Так, диапазон значащих чисел в системе счисления с основанием P при наличии m разрядов у мантиссы и s разрядов у порядка (без учета знаковых разрядов порядка и мантиссы) будет

$$P^{-m} \times P^{-(P^s-1)} \le N \le (1 - P^{-m}) \times P^{(P^s-1)}.$$

Например, при $P = 2$, $m = 22$ и $s = 10$ диапазон чисел простирается примерно от 10^{-300} до 10^{300}. Для сравнения скажем, что количество секунд, которые прошли с момента образования планет Солнечной системы составляет около 10^{18}.

Следует заметить, что все числа с плавающей запятой хранятся в так называемом *нормализованном* виде. Нормализованным называют такое число, в старший разряд мантиссы которого больше нуля. У нормализованных двоичных чисел, следовательно, $0{,}5 < |M| < 1$.

Нормализованные, т. е. приведенные к правильной дроби, числа:

$10{,}35_{10} = 0{,}1035_{10} \times 10^{+2}$;

$0{,}00007245_{8} = 0{,}7245_{8} \times 8^{-4}$;

$F5C{,}9B_{16} = 0{,}F5C9B_{16} \times 16^{+3}$.

Обычно в памяти ЭВМ числа с ПЗ хранятся в двух форматах:

1) слово — 31 бита или 4 байта (одинарная точность);

2) двойное слово — 64 бита или 8 байт (двойная точность).

Разрядная сетка для чисел с ПЗ (формат IEEE 754) имеет следующую структуру:

- 31-й (63-й) разряд — знак числа (0 — «минус», 1 — «плюс»);
- с 30-го по 23-й (8 бит для одинарной точности) или с 62-го по 52-й (11 бит для двойной) разряды записывается порядок в прямом двоичном коде. В первом разряде указывается знак порядка (1 — «плюс» или 0 — «минус»);
- с 22-го по нулевой (24 бита для одинарной точности) или с 51-го по нулевой (52 бита для двойной) разряды размещается мантисса числа.

Алгебраическое представление двоичных чисел. Для алгебраического представления чисел (с учетом знака) используются специальные коды:

- прямой код числа;
- обратный;
- дополнительный.

При этом два последних кода позволяют заменить операцию вычитания на сложение с отрицательным числом.

Знак числа обычно кодируется двоичной цифрой, при этом:

- код 0 означает знак «+» (плюс);
- код 1 означает знак «–» (минус).

Прямой код числа N (обозначим $[N]_{пр}$).

Пусть $N = a_1, a_2, a_3, ..., a_m$, тогда:

- при $N > 0$ $[N]_{пр} = 0, a_1, a_2, a_3, ..., a_m$;
- при $N < 0$ $[N]_{пр} = 1, a_1, a_2, a_3, ..., a_m$;
- при $N = 0$ имеет место неоднозначность $[0]_{пр} = 0, 0... = 1, 0...$

Если при сложении в ЭВМ оба слагаемых имеют одинаковый знак, то операция сложения выполняется обычным путем. Если при сложении слагаемые имеют разные знаки, то сначала необходимо выявить большее по абсолютной величине число, из него произвести вычитание меньшего по абсолютной величине числа и разности присвоить знак большего числа.

Выполнение операций умножения и деления в прямом коде выполняется обычным образом, но знак результата определяется по совпадению или несовпадению знаков участвующих в операции чисел.

Операцию вычитания в этом коде нельзя заменить операцией сложения с отрицательным числом, поэтому возникают сложности, связанные с займом значений из старших разрядов уменьшаемого числа.

Обратный код числа N (обозначим $[N]_{обр}$).

Пусть $N = a_1, a_2, a_3, ..., a_m$ и $\tilde{a}$ обозначает *инверсию* a, т. е. если $a = 1$, то $\tilde{a} = 0$, и наоборот. Тогда:

- при $N > 0$ $[N]_{обр} = 0, a_1, a_2, a_3, ..., a_m$;
- при $N < 0$ $[N]_{обр} = 1, \tilde{a}_1, \tilde{a}_2, \tilde{a}_3, ..., \tilde{a}_m$;
- при $N = 0$ имеет место неоднозначность $[0]_{обр} = 0{,}00...0 = 1{,}11...1$.

Для того чтобы получить обратный код отрицательного числа, необходимо все цифры этого числа *инвертировать*, т. е. в знаковом разряде поставить 1, во всех значащих разрядах нули заменить единицами, а единицы — нулями (см. также табл. 1.8).

Например,

для $N = 1011$ $[N]_{обр} = 0{,}1011$;

для $N = -1011$ $[N]_{обр} = 1{,}0100$.

Дополнительный код числа N (обозначим $[N]_{доп}$).

Пусть, как и выше, $N = a_1, a_2, a_3, ..., a_m$ и $\tilde{a}$ обозначает величину, обратную a (инверсию a), т. е. если $a = 1$, то $\tilde{a} = 0$, и наоборот. Тогда:

- при $N \geq 0$ $[N]_{доп} = 0, a_1, a_2, a_3, ..., a_m$;
- при $N \leq 0$ $[N]_{доп} = 1, \tilde{a}_1, \tilde{a}_2, \tilde{a}_3, ..., \tilde{a}_m + 0{,}00...1$.

Для того чтобы получить дополнительный код отрицательного числа, необходимо все его цифры инвертировать (в знаковом разряде поставить единицу, во всех значащих разрядах нули заменить единицами, а единицы нулями) и затем к младшему разряду прибавить единицу. В случае возникновения переноса из первого после запятой разряда в знаковый разряд к числу следует прибавить единицу в младший разряд.

Например,

для $N = 1011$, $[N]_{доп} = 0{,}1011$;

для $N = -1100$, $[N]_{доп} = 1{,}0100$;

для $N = -0000$, $[N]_{доп} = 10{,}0000 = 0{,}0000$ (1 исчезает). Неоднозначности в изображении 0 нет.

Эмпирическое правило: для получения дополнительного кода отрицательного числа необходимо все символы этого числа инвертировать, кроме последней (младшей) единицы и тех нулей, которые за ней следуют.

Типы и структуры данных

Типы данных (табл. 1.6). Классификация информационных единиц, обрабатываемых на ЭВМ, включает следующие их характеристики:

- типы данных, или совокупность соглашений о программно-аппаратурной форме представления и обработки, а также ввода, контроля и вывода элементарных данных;
- структуры данных — способы композиции простых данных в агрегаты и операции над ними.

Ранние языки программирования (ЯП), точнее, системы программирования (СП) — Fortran, Algol, будучи ориентированы исключительно на вычисления, не содержали развитых систем типов и структур данных.

Таблица 1.6. **Типы и структуры данных в некоторых языках программирования**

Тип данных	Язык программирования			
	Algol	Pascal	Basic	FoxPro
Целое короткое (2 байта)	–	`Integer` `Smallint`	`Integer`	—
Целое нормальной длины (4 байта)	`Integer`	`Longint` `Integer`	`Long`	`Integer`
Действительное нормальной длины (4 байта)	`Real`	`Single`	`Single`	—
Действительное двойной длины (8 байт)	—	`Real`	`Double` `Currency`	`Numeric` (до 20 байт), `Double`, `Currency`, `Date`, `DateTime`
Логическое	`Boolean`	`Boolean`	`Boolean`	`Logical`
Символьное (1 байт)	—	`Char`	—	`Character` (1—254 байт)
Массивы	`Array`	`Array`	`Dim`	`Dim`
Записи (структуры)	—	`Record`	—	Записи файлов данных

В стандартном ЯП Algol символьные величины и переменные вообще не предусматривались, а в некоторых реализациях строки (символы в апострофах) могли встречаться только в операторах печати данных.

Типы числовых данных ЯП Algol — `Integer` (целое число), `Real` (действительное) — различаются диапазонами изменения, внутренними представлениями и применяемыми командами процессора ЭВМ (соответственно арифметика с фиксированной и плавающей точкой). Нечисловые данные представлены типом `Boolean` — логические, имеющие диапазон значений {`true`, `false`}.

Позже появившиеся ЯП (СП) Cobol, PL/1, Pascal вводят новые типы данных:

- символьные (цифры, буквы, знаки препинания и пр.);
- числовые символьные для вывода;
- числовые двоичные для вычислений;
- числовые десятичные (цифры 0—9) для вывода и вычислений.

Разновидности числовых данных здесь соответствуют внутреннему представлению и машинным (или эмулируемым) командам обработки. Кроме того, вводятся числа двойного формата (два машинных слова), для обработки которых также необходимо наличие в процессоре (или эмуляция) команд обработки чисел двойной длины (точности).

Приведем пример представления числовой информации в различных перечисленных формах. Пусть задано число $135_{10} = 207_8 = 87_{16} = 10000111_2$, тогда:

- внутренняя стандартная форма представления (тип `Binary` для обработки в двоичной арифметике) сохраняется (10000111_2). Объем — 1 байт, или 8 двоичных разрядов;
- внутренняя форма двоично-десятичного представления (тип `Decimal`, каждый разряд десятичного числа представляется двоично-десятичной, в 4 бита, комбинацией). Представление 135 есть $001\ 011\ 101_2$. Объем — 2,5 байта, 12 двоичных разрядов;
- символьное представление (тип `Alphabetic`, для вывода) — каждый разряд представляется байтом в соответствии с кодом ASCII. Представление 135 есть — $00110001\ 00110011\ 00110101_2$. Объем — 3 байта.

Структуры данных. В ЯП Algol были определены два типа структур: элементарные данные и массивы (векторы, матрицы, тензоры, состоящие из арифметических или логических переменных). Основным нововведением, появившимся первоначально в ЯП Кобол (затем PL/1, Pascal и пр.), являются агрегаты данных (структуры, записи), представляющие собой именованные комплексы переменных разного типа, описывающих некоторый объект или образуюших некоторый достаточно сложный документ.

Термин *запись* подразумевает наличие множества аналогичных по структуре агрегатов, образующих *файл* (картотеку), содержащих данные по совокупности однородных объектов, элементы данных образуют поля, среди которых выделяются элементарные и групповые (агрегатные) — см. также рис. 5.3.

Структурированные типы данных. По мере развития систем программирования в некоторых из них произошло «слияние» понятий типов и структур и появились структурированные типы. Например, в Visual Basic к структурированным типам относятся — дата, массив, строка, объект.

1.2. Логические основы программирования

Начало исследований в области формальной логики было положено работами Аристотеля в IV в. до нашей эры. Однако математические подходы к этим вопросам впервые были указаны Дж. Булем. В честь него алгебру высказывания называют булевой (булевской) алгеброй, а логические значения — булевыми (булевскими). Основу математической логики составляет алгебра высказываний. Алгебра логики используется при построении основных узлов ЭВМ (дешифратор, сумматор, шифратор).

Алгебра логики оперирует с высказываниями. Под высказыванием понимают повествовательное предложение, относительно которого можно утверждать, истинно оно или ложно. Например, выражение «Расстояние от Москвы до Киева больше, чем от Москвы до Тулы» истинно, а выражение «5 < 3» — ложно (см. рис. 5.5).

Высказывания (логические переменные) принято обозначать буквами латинского алфавита (иногда с индексами): A, B, C, ..., X, Y, a, b, c, ..., x, y, z, (x_1, x_2, ..., x_i, ...) и т. д. Если высказывание C истинно, это обозначается как $C = 1$ ($C =$ t, true), а если оно ложно, то $C = 0$ ($C =$ f, false).

Логические операции

В алгебре логики над высказываниями можно производить определенные логические операции, в результате которых получаются новые (выходные) высказывания, значения (истинность или ложность) которых зависит как от значений входных, так и от использованных логических операций.

Конъюнкция. Соединение двух (или нескольких) высказываний в одно с помощью союза И (OR) называется операцией логического умножения, или конъюнкцией. Эту операцию принято обозначать знаками «∧, &» или знаком умножения «×». Сложное высказывание A ∧ B истинно только в том случае, когда истинны оба входящих в него высказывания. Истинность такого высказывания задается табл. 1.7.

Дизъюнкция. Объединение двух (или нескольких) высказываний с помощью союза ИЛИ (OR) называется операцией логического сложения, или дизъюнкцией. Эту операцию обозначают знаками «|,∨» или знаком сложения «+». Сложное высказывание $A \vee B$ истинно, если истинно хотя бы одно из входящих в него высказываний (см. табл. 1.7).

Таблица 1.7. **Таблицы истинности конъюнкции и логической суммы высказываний**

Конъюнкция			Дизъюнкция			
A	B	$A \wedge B$	A	B	$A \vee B$	A xor B
0	0	0	0	0	0	0
0	1	0	0	1	1	1
1	0	0	1	0	1	1
1	1	1	1	1	1	0

В последнем столбце табл. 1.7 размещены результаты модифицированной операции ИЛИ — Исключающее ИЛИ (XOR). Отличается от обычного ИЛИ последней строкой (см. также рис. 1.2, *в*).

Инверсия. Присоединение частицы НЕ (NOT) к некоторому высказыванию называется операцией отрицания (инверсии) и обозначается $\neg A$ (или $\overline{A}$). Если высказывание A истинно, то $\overline{A}$ ложно, и наоборот (табл. 1.8).

Таблица 1.8. **Таблица истинности отрицания**

A	$\overline{A}$
0	1
1	0

Следует отметить, что помимо операций И, ИЛИ, НЕ в алгебре высказываний существует ряд других операций. Например, операция эквивалентности (эквиваленции) $A \sim B$ ($A \equiv B$, или A eqv B) (табл. 1.9).

Таблица 1.9. **Таблицы истинности операций эквивалентности и импликации**

Эквивалентность			Импликация		
A	B	$A \sim B$	A	B	$A \rightarrow B$
0	0	1	0	0	1
0	1	0	0	1	1
1	0	0	1	0	0
1	1	1	1	1	1

Другим примером может служить логическая операция импликации или логического следования ($A \to B$, A IMP B), иначе говоря, «ЕСЛИ A, то B» (табл. 1.9).

Высказывания, образованные с помощью логических операций, называются сложными. Истинность сложных высказываний можно установить, используя таблицы истинности. Например, истинность сложного высказывания $\overline{A} \wedge \overline{B}$ определяется табл. 1.10.

Таблица 1.10. **Таблица истинности высказывания $\overline{A} \wedge \overline{B}$**

A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \wedge \overline{B}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

Высказывания, у которых таблицы истинности совпадают, называются *равносильными* (тождественными). Для обозначения равносильных высказываний используют знак «=» ($A = B$). Рассмотрим сложное высказывание $(A \wedge B) \vee (\overline{A} \wedge \overline{B})$ — табл. 1.11.

Таблица 1.11. **Таблица истинности выражения $(A \wedge B) \vee (\overline{A} \wedge \overline{B})$**

A	$\overline{A}$	B	$\overline{B}$	$A \wedge B$	$\overline{A} \wedge \overline{B}$	$(A \wedge B) \vee (\overline{A} \wedge \overline{B})$
0	1	0	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	1

Если сравнить эту таблицу с таблицей истинности операции эквивалентности высказываний A и B (см. табл. 1.9), то можно увидеть, что высказывания $(A \wedge B) \vee (\overline{A} \wedge \overline{B})$ и $A \sim B$ тождественны, т. е. $(A \sim B) = (A \wedge B) \vee (\overline{A} \wedge \overline{B})$.

В алгебре высказываний можно проводить тождественные преобразования, заменяя одни высказывания равносильными им другими высказываниями.

Свойства логических операций. Исходя из определений дизъюнкции, конъюнкции и отрицания, устанавливаются свойства этих операций и взаимные распределительные свойства. Приведем примеры некоторых из этих свойств:

- коммутативность (перестановочность):
  $A \wedge B = B \wedge A$,
  $A \vee B = B \vee A$;
- закон идемпотентности:
  $A \wedge A = A$, $A \vee A = A$;
- двойное отрицание:
  $\overline{\overline{A}} = A$;
- сочетательные (ассоциативные) законы:
  $A \vee (B \vee C) = (A \vee B) \vee C = A \vee B \vee C$,
  $A \wedge (B \wedge C) = (A \wedge B) \wedge C = A \wedge B \wedge C$;
- распределительные (дистрибутивные) законы:
  $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$,
  $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$;
- поглощение:
  $A \vee (A \wedge B) = A$,
  $A \wedge (A \vee B) = A$;
- склеивание:
  $(A \wedge B) \vee (\overline{A} \wedge B) = B$,
  $(A \vee B) \wedge (\overline{A} \vee B) = B$;
- операция переменной с ее инверсией:
  $A \wedge \overline{A} = 0$,
  $A \vee \overline{A} = 1$;
- операция с константами (0 — `false`, 1 — `true`):
  $A \wedge 1$, $A \vee 1 = 1$,
  $A \wedge 0 = 0$, $A \vee 0 = A$;
- законы де Моргана:
  $\overline{A \wedge B} = \overline{A} \vee \overline{B}$ (условно его можно назвать 1-й);
  $\overline{A \vee B} = \overline{A} \wedge \overline{B}$ (2-й).

Высказывания, образованные с помощью нескольких операций логического сложения, умножения и отрицания, называют-

ся сложными. Истинность всякого сложного высказывания устанавливается с помощью таблиц истинности. Сложные высказывания, истинные (true) для любых значений истинности, входящих в них простых высказываний, называются тождественно-истинными. Наоборот, тождественно-ложными являются формулы, принимающие значение false для любых значений входящих в него простых высказываний.

В табл. 1.12 приведено доказательство истинности дистрибутивного закона. Аналогичным образом могут быть доказаны и другие тождества.

Таблица 1.12. **Доказательство истиииости дистрибутивного закона**

A	B	C	$B \vee C$	$A \wedge (B \vee C)$	$A \wedge B$	$A \wedge C$	$(A \wedge B) \vee (A \wedge C)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

На рис. 1.2 приведены иллюстрации к основным логическим операциям и их композициям (так называемые диаграммы Эйлера — Венна) — области истинности каждого из высказываний и результатов их объединения (дизъюнкции), пересечения (конъюнкции) и других операций. «Первый» из законов де Моргана иллюстрируется рис. 1.2, *д*, *е*.

Логическое значение null. В некоторых ЯП (Visual Basic и пр.) для расширения применимости логических выражений на те случаи, когда значение одного или нескольких логических аргументов неизвестны или не определены, вводится значение null (в дополнение к false и true), как правило, такое значение присваивается компилятором логической переменной по умолчанию. С учетом значения null таблицы истинности основных логических операций приобретают вид табл. 1.13, 1.14.

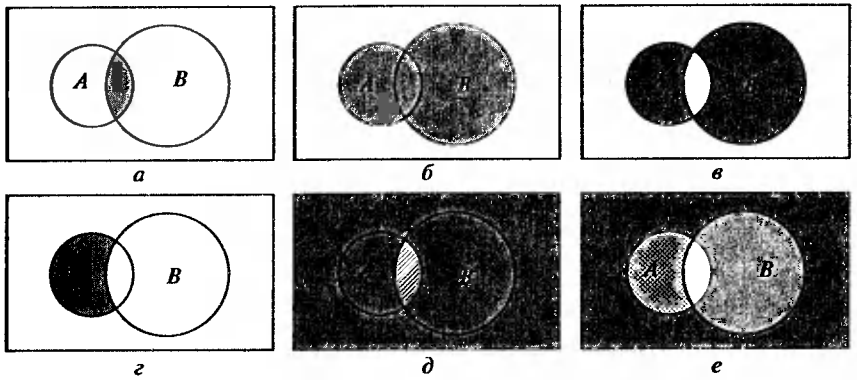


Рис. 1.2. Некоторые примеры диаграмм Эйлера — Венна: *а* — конъюнкция высказываний *A* и *B* (AND); *б* — дизъюнкция высказываний *A* и *B* (OR); *в* — исключающая дизъюнкция (XOR); *г* — разность высказываний (*A*—*B*); *д* — иллюстрация к законам де Моргана (дополнение пересечения высказываний); *е* — иллюстрация к законам де Моргана (объединение дополнений)

Таблица 1.13. **Одноместиая (унариая) операция отрицания с учетом значеиия null**

A	$\bar{A}$
false	true
true	false
null	null

Таблица 1.14. **Некоторые двухместные (бинарные) операции с учетом значения null**

A	B	$A \& B$	$A \vee B$	$A \to B$	$A \sim B$	A xor B
false	false	false	false	true	true	false
false	true	false	true	true	false	true
true	false	false	true	false	false	true
true	true	true	true	true	true	false
false	null	false	false	true	null	null
true	null	null	true	null	null	null
null	false	false	null	null	null	null
null	true	null	true	true	null	null
null	null	null	null	null	null	null

Побитовые операции. В некоторых современных ЯП включены операции побитового сравнения содержимого машинных слов (которые могут содержать числовые, строчные и др. данные) при этом каждый бит результата образуется в соответствии с табл. 1.15 (для бинарных операций). Унарная операция отрицания (NOT) в данном случае реализует очевидную замену «1» на «0», и наоборот.

Таблица 1.15. **Операнды и результаты некоторых операций побитового сравнения**

x	y	$x \& y$	$x \vee y$	x IMP y	x EQV y	x XOR y
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	0	0	1
1	1	1	1	1	1	0

1.3. Алгоритмы и программирование

Понятие *алгоритма* является одним из основных в современной науке и практике. Однако еще на самых ранних этапах развития математики (Древний Египет, Вавилон, Греция) рассматривались вычислительные процессы, с помощью которых искомые величины задачи вычислялись последовательно из исходных данных по определенным правилам и инструкциям. Со временем все такие процессы в математике получили название алгоритмов (алгорифмов).

Алгоритм есть совокупность четко определенных правил, процедур или команд, обеспечивающих решение поставленной задачи за конечное число шагов.

Считается, что термин *алгоритм* происходит от имени средневекового узбекского математика Аль-Хорезми, который еще в IX в. (825 г.) предложил правила выполнения четырех арифметических действий в десятичной системе счисления. Процесс выполнения арифметических действий был назван *алгоризмом*.

С 1747 г. вместо слова *алгоризм* стали употреблять *алгоресмус*, смысл которого состоял в комбинировании четырех операций арифметического исчисления — сложения, вычитания, умножения, деления. К 1950 г. *алгоресмус* стал *алгорифмом*. Смысл алгорифма чаще всего связывался с алгорифмами Евклида — процессами нахождения наибольшего общего делителя двух многочленов, наибольшей общей меры двух отрезков, квадратного корня и т. п.

Способы записи алгоритмов

Алгоритм должен быть понятен (доступен) пользователю и/или машине. Доступность пользователю означает, что он обязан отображаться посредством конкретных формализованных изобразительных средств, понятных пользователю. В качестве таких изобразительных средств используются следующие способы их записи:

- словесный, содержание последовательных этапов алгоритма описывается в произвольной форме на естественном языке;
- формульный, основан на строго формализованном аналитическом задании необходимых для исполнения действий;
- табличный, подразумевает отображение алгоритма в виде таблиц, использующих аппарат реляционного исчисления и алгебру логики для задания подлежащих исполнению взаимных связей между данными, содержащимися в таблице;
- операторный, базируется на использовании для отображения алгоритма условного набора стандартных операторов: арифметических, логических, печати, ввода данных и т. д.; операторы снабжаются индексами и между ними указываются необходимые переходы, а сами индексированные операторы описываются чаще всего в табличной форме;
- графический, отображение алгоритмов в виде блок-схем, весьма наглядный и распространенный способ. Графические символы, отображающие выполняемые процедуры, стандартизованы. Наряду с основными символами используются и вспомогательные, поясняющие процедуры и связи между ними;
- на языке программирования, в виде последовательности команд какого-либо языка программирования.

Приведем пример словесного представления алгоритма на примере нахождения произведения N натуральных чисел ($C = N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (N-1) \cdot N$).

Этот процесс может быть записан в виде следующей последовательности указаний (пунктов):

1. Полагаем C равным единице и переходим к следующему пункту.

2. Полагаем K равным единице и переходим к следующему пункту.

3. Полагаем C равным $C = K \times C$ и переходим к следующему указанию.

4. Проверяем, равно ли K числу N. Если $K = N$, то вычисления прекращаем. Если $K < N$, то увеличиваем K на единицу и переходим к п. 3.

Классификация и свойства алгоритмов

Алгоритмы, в соответствии с которыми решение поставленных задач сводится к арифметическим действиям, называются *численными алгоритмами.*

Алгоритмы, в соответствии с которыми решение поставленных задач сводится к логическим действиям, называются *логическими алгоритмами.* Примерами логических алгоритмов могут служить алгоритмы поиска минимального числа, поиска пути на графе, поиска пути в лабиринте и др.

Алгоритмом является последовательность четких однозначных указаний, которые, будучи применены к определенным имеющимся данным, обеспечивают получение требуемого результата. *Данными* называют все величины, участвующие в решении задачи. Данные, известные перед выполнением алгоритма, являются начальными, *исходными данными.* Результат решения задачи — это конечные, *выходные данные.*

Каждое указание алгоритма предписывает исполнителю выполнить одно конкретное законченное действие. Исполнитель не может перейти к выполнению следующей операции, не закончив полностью выполнения предыдущей. Предписания алгоритма надо выполнять последовательно одно за другим, в соответствии с указанным порядком их записи. Выполнение всех предписаний гарантирует правильное решение задачи.

Поочередное выполнение команд алгоритма за конечное число шагов приводит к решению задачи, к достижению цели. Разделение выполнения решения задачи на отдельные операции

(выполняемые исполнителем по определенным командам) — важное свойство алгоритмов, называемое *дискретностью.*

Для того чтобы алгоритм мог быть выполнен, нельзя включать в него команды, которые исполнитель не в состоянии исполнить. У каждого исполнителя имеется свой перечень команд, которые он способен выполнить. Совокупность команд, которые могут быть выполнены исполнителем, называется *системой команд исполнителя.*

Каждая команда алгоритма должна определять однозначное действие исполнителя. Такое свойство алгоритмов называется *определенностью (или точностью) алгоритма.*

Алгоритм, составленный для конкретного исполнителя, должен включать только те команды, которые входят в его систему команд. Это свойство алгоритма называется *понятностью.* Алгоритм не должен быть рассчитан на принятие каких-либо самостоятельных решений исполнителем, не предусмотренных составлением алгоритма.

Еще одно важное требование, предъявляемое к алгоритмам, — *результативность (или конечность)* алгоритма. Оно означает, что исполнение алгоритма должно закончиться за конечное число шагов.

Поскольку разработка алгоритмов — процесс творческий, требующий умственных усилий и затрат времени, предпочтительно разрабатывать алгоритмы, обеспечивающие решения всего класса задач данного типа. Например, если составляется алгоритм решения кубического уравнения $ax^3 + bx^2 + cx + d = 0$, то он должен быть *вариативен*, т. е. обеспечивать возможность решения для любых допустимых исходных значений коэффициентов a, b, c, d. Про такой алгоритм говорят, что он удовлетворяет требованию *массовости.* Свойство массовости не является необходимым свойством алгоритма. Оно, скорее, определяет качество алгоритма; в то же время свойства точности, понятности и конечности являются необходимыми (в противном случае это не вполне алгоритм).

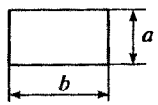
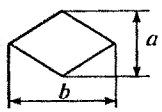
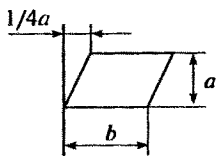
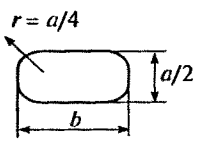
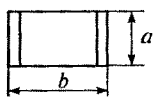
Запись алгоритмов в виде блок-схем

Алгоритмы можно записывать различными методами. Форма записи, состав и количество операций алгоритма зависят от исполнителя этого алгоритма. Если задача решается с помощью

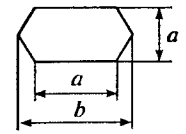
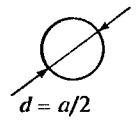
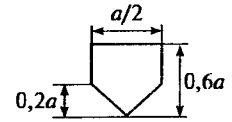
ЭВМ, алгоритм решения задачи должен быть записан в понятной для машины форме, т. е. в виде программы.

Схема алгоритма — графическое представление алгоритма, дополняемое элементами словесной записи. Каждый пункт алгоритма отображается на схеме некоторой геометрической фигурой или блоком. При этом правило выполнения схем алгоритмов регламентирует ГОСТ 19.002—80 «Единая система программной документации» (табл. 1.16).

Таблица 1.16. **Основные элементы блок-схем**

№	Символ	Наименование	Содержание	Размеры по ГОСТ 19.003—80 (ЕСПД): $a = 10, 15, 20$ мм; $b = 1{,}5a$
1	$a := b$ $b := c$	Блок вычислений	Вычислительные действия или последовательность действий	
2		Логический блок	Выбор направления выполнения алгоритма в зависимости от некоторого условия	
3		Блоки ввода-вывода данных	1. Общие обозначения ввода (вывода) данных (вне зависимости от физического носителя). 2. Вывод данных, носителем которых является документ	
4		Начало (конец)	Начало или конец алгоритма, вход в программу или выход из нее	
5		Процесс пользователя (подпрограмма)	Вычисление по стандартной программе или подпрограмме	

Окончание табл. 1.16

№	Символ	Наименование	Содержание	Размеры по ГОСТ 19.003—80 (ЕСПД): $a = 10, 15, 20$ мм; $b = 1{,}5a$
6		Блок модификации	Функция выполняет действия, изменяющие пункты (например, заголовок цикла) алгоритма	
7		Соединитель	Указание связи прерванными линиями между потоками информации в пределах одного листа	
8		Межстраничные соединения	Указание связи между информацией на разных листах	

Блоки на схемах соединяются линиями потоков информации. Основное направление потока информации идет сверху вниз и слева направо (стрелки могут не указываться), снизу вверх и справа налево — стрелки обязательны. Количество входящих линий для блока не ограничено. Выходящая линия — одна, за исключением логического блока.

Базовые структуры алгоритмов

Это определенный набор блоков и стандартных способов их соединения для выполнения типичных последовательных действий. К основным структурам относятся следующие — линейные, разветвляющиеся, циклические (рис. 1.3).

Линейными называются алгоритмы, в которых действия осуществляются последовательно друг за другом.

Разветвляющимся называется алгоритм в который, в отличие от линейных алгоритмов, входит условие, в зависимости от вы-

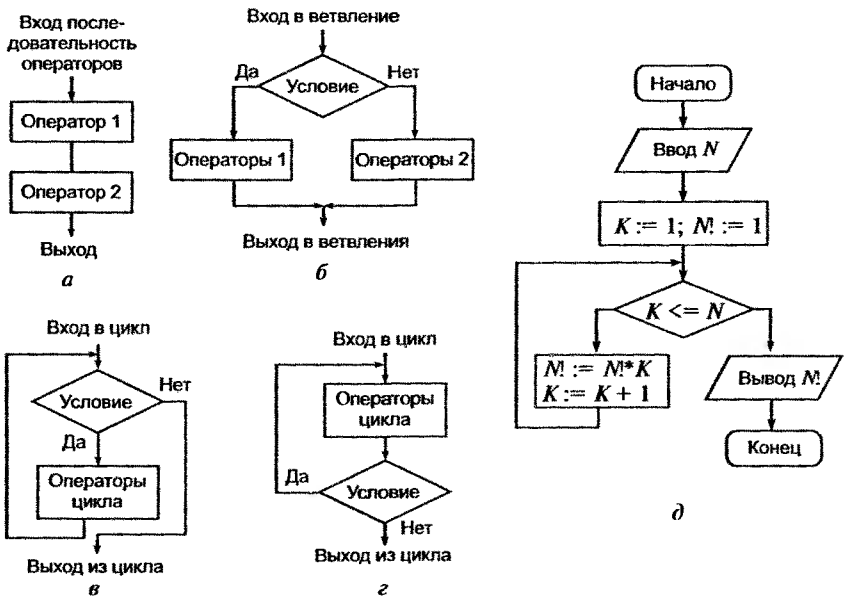


Рис. 1.3. Базовые структуры алгоритмов и программ:
а — линейный алгоритм; *б* — алгоритм с ветвлением; *в* — алгоритм с циклом While (Пока); *г* — алгоритм с циклом For (До); *д* — пример алгоритма вычисления *N*-факториал

полнения или нет которого реализуется та или иная последовательность команд (серий).

В качестве условия в разветвляющемся алгоритме может быть использовано любое понятное исполнителю утверждение, которое может соблюдаться (быть истинно) или не соблюдаться (быть ложно). Такое утверждение может быть выражено как словами, так и формулой. Таким образом, команда ветвления состоит из условия и двух последовательностей команд.

Циклическим называется алгоритм, в котором некоторая последовательность операций (тело цикла) выполняется многократно. Однако «многократно» не означает «неограниченно». Организация циклов, никогда не приводящая к остановке в выполнении алгоритма, является нарушением требования его результативности — получения результата за конечное число шагов.

Перед операцией цикла осуществляются операции начального присвоения значений тем переменным, которые используются

в теле цикла. В цикл в качестве базовых входят следующие структуры: блок проверки условия и тело цикла. Если тело цикла расположено после проверки условий (цикл с предусловием), то может случиться, что при определенных условиях блок «тело цикла» не выполнится ни разу. Такой вариант организации цикла, управляемый предусловием, называется цикл Пока (While) — здесь условие — это условие продолжения цикла.

Возможен другой случай, когда тело цикла выполняется, по крайней мере, один раз и будет повторяться до тех пор, пока не станет истинным условие. Такая организация цикла, когда его тело расположено перед проверкой условия, носит название цикла с постусловием, или цикла До (For). Современные языки программирования имеют достаточный набор операторов, реализующих как цикл Пока, так и цикл До.

Рассмотрим пример алгоритма вычисления факториала (описанный выше в словесной форме), изображенный на рис. 1.3, *д* (с циклом Пока). Переменная N получает значение числа, факториал которого вычисляется. Переменной $N!$, которая в результате выполнения алгоритма должна получить значение факториала, присваивается первоначальное значение 1. Переменной K также присваивается значение 1. Цикл будет выполняться, пока справедливо условие $N \geq K$. Тело цикла состоит из двух операций $N! = N! \times K$ и $K = K + 1$.

Циклические алгоритмы, в которых тело цикла выполняется заданное число раз, реализуются с помощью цикла со счетчиком. Цикл со счетчиком реализуется с помощью команды повторения.

Процесс решения сложной задачи довольно часто сводится к решению нескольких более простых подзадач. Соответственно при разработке сложного алгоритма он может разбиваться на отдельные алгоритмы, которые называются вспомогательными. Каждый такой вспомогательный алгоритм описывает решение какой-либо подзадачи.

Процесс построения алгоритма методом последовательной детализации состоит в следующем. Сначала алгоритм формулируется в «крупных» блоках (командах), которые могут быть непонятны исполнителю (не входят в его систему команд), и записываются как вызовы вспомогательных алгоритмов. Затем происходит детализация, и все вспомогательные алгоритмы подробно расписываются с использованием команд, понятных исполнителю.

1.4. Языки и системы программирования

Способы описания языков программирования

Для описания языков программирования используются две системы описания:

- нотация Бэкуса (впервые предложена при описании языка Algol);
- нотация IBM (разработана фирмой для описания языков Cobol и JCL).

Обе нотации используют следующие обозначения:

- < > — угловые скобки (или двойные кавычки " ") обозначают элемент программы, определяемый пользователем (<Идентификатор>, <Список параметров>, "Условие" и пр.). В соответствующих местах реальной программы будет находиться идентификатор переменной, список параметров и пр.).

Нотация Бэкуса содержит конструкции следующего вида:

```
<Оператор присваивания> ::= <Переменная> := <Выражение>
<Идентификатор> ::= <Буква>|<Идентификатор><Буква>|
<Идентификатор><Цифра>
```

Левая часть определения конструкции языка содержит наименование определяемого элемента, взятого в угловые скобки.

Правая часть включает совокупность элементов, соединенных знаком |, который трактуется как «или» и объединяет альтернативы — различные варианты значения определяемого элемента.

Части соединяются оператором ::=, который может трактоваться как является по определению.

Существенно, что многие определения носят рекурсивный характер, т. е. левая часть содержит правую (как для элемента <Идентификатор> в вышеприведенных примерах), и вместо элементов в правой части определения можно подставлять любые их значения. Например, в определении идентификатора, начиная рассматривать его как букву A, можно затем получать любые комбинации: AA, A0, AA1, AA1B и т. п.

Нотация IBM включает следующие элементы:

- [ ] — квадратные скобки. Обозначают возможное отсутствие элемента конструкции. Например, return [<Выражение>] — в этой конструкции <Выражение> не обязательно;
- { } — фигурные скобки

  {<Выражение-1> | <Выражение-2> | <Выражение-3>}

  или $\left\{\begin{array}{l}\text{<Выражение-1>}\\ \text{<Выражение-2>}\\ \text{<Выражение-3>}\end{array}\right\}$, означают обязательное присутствие одного из <Выражений>;
- | — вертикальная черта. Разделяет список значений обязательных элементов, одно из которых должно быть выбрано;
- ... — горизонтальное многоточие, следующее после некоторой синтаксической конструкции, обозначает последовательность конструкций той же самой формы, что и предшествующая многоточию конструкция. Например, {<Выражение> [,<Выражение>]...} обозначает, что одно или более <Выражений>, разделенных запятыми, может появиться между фигурными скобками.

Рекурсивные определения в IBM-нотации не используются.

Архитектуры ЭВМ

Рассмотрим вначале проблему программирования в абсолютных (машинных) адресах. Вычислительная машина (система) независимо от типа и поколения состоит из двух основных групп устройств:

- центрального устройства (ЦУ);
- периферийных (внешних) устройств (ВУ).

Исторически первые ЭВМ классической структуры и состава были выпущены фирмой IBM (США) — Computer Installation System/360 (фирменное наименование — «Вычислительная установка системы 360», в дальнейшем известная как просто IBM/360) в 1964 г., и с последующими модификациями (IBM/370, IBM/375) поставлялись вплоть до конца 1980 гг., когда под влиянием микроЭВМ (ПК) не начали постепенно сходить со сцены. ЭВМ данной серии послужили основой для

разработки в СССР и странах-членах СЭВ так называемой Единой системы ЭВМ (ЕС ЭВМ), которые в течение трех десятилетий являлись основой отечественной компьютеризации.

Здесь же необходимо упомянуть также о еще двух популярных в СССР сериях ЭВМ, прототипами которых являлись зарубежные разработки — СМ ЭВМ (семейство малых ЭВМ, прототип — машины PDP/11 фирмы Digital Equipment Corporation (также известной как Digital или DEC) и серия «М» (М-5000, М-6000) управляющих машин (прототип — разработки фирмы Hewlett-Packard — HP 2000/HP 3000 и пр.).

Архитектура ЭВМ характеризует состав и топологию соединения и взаимодействия данных устройств [15, 16]. Основные типы архитектур — звездная, иерархическая, магистральная.

Внешние устройства (ВУ) разделяются на три типа:

- ввода информации;
- вывода информации;
- хранения;
- интерактивные устройства (ввода-вывода).

Однотипные ЦУ и устройства хранения данных могут использоваться в различных типах машин. Известны примеры того, как фирмы, начавшие свою деятельность с производства управляющих машин, совершенствуя свою продукцию, перешли к выпуску систем, которые в зависимости от конфигурации ВУ могут исполнять как роль универсальных, так и управляющих машин (Hewlett-Packard и Digital Equipment Corporation).

Абстрактное центральное устройство

Перечислим основные понятия и рассмотрим структуру и функции абстрактного центрального устройства ЭВМ (1—2-е поколения ЭВМ) (рис. 1.4), арифметико-логическое устройство (АЛУ) (arithmetic and logic unit — ALU) которого предназначено для обработки целых чисел и битовых строк.

Команда (instruction) — описание операции, которую нужно выполнить. Каждая команда начинается с *кода операции (КОП)*, содержит необходимые *адреса*, характеризуется форматом, который определяет структуру команды, ее организацию, код, длину,

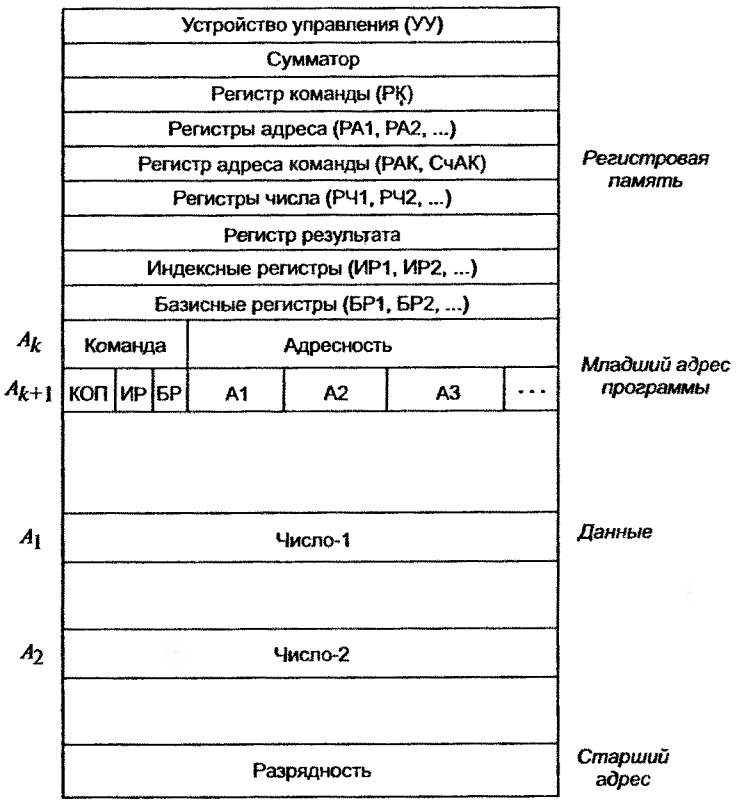


Рис. 1.4. Структура абстрактного центрального устройства ЭВМ

метод расположения адресов. Длина различных команд может быть как одинаковой, так и разной.

Цикл процессора — период времени, за который осуществляется выполнение команды исходной программы в машинном виде; состоит из нескольких *тактов*.

Такт работы процессора — промежуток времени между соседними импульсами *генератора тактовых импульсов*, частота которых есть *тактовая частота процессора*. Выполнение *короткой команды* (арифметика с фиксированной запятой, логические операции), о которых речь здесь и пойдет, обычно занимает пять тактов:

- выборка команды;
- расшифровка кода операции (декодирование);

- генерация адреса и выборка данных из памяти;
- выполнение операции;
- запись результата в память.

Процедура, соответствующая такту, реализуется определенной логической цепью (схемой) процессора, обычно именуемой *микропрограммой.*

Регистры — устройства, предназначенные для временного хранения данных ограниченного размера. Важной характеристикой регистра является высокая скорость приема и выдачи данных. Регистр состоит из разрядов, в которые можно быстро записывать, запоминать и считывать слово, команду, двоичное число и т. д. Обычно регистр имеет ту же разрядность, что и машинное слово. Перечислим основные виды регистров.

Регистры общего назначения (РОН) (General Purpose Registers) — общее название для регистров, которые временно содержат данные, передаваемые или принимаемые из памяти.

Регистр команды (РК) (Instruction Register — IR) служит для размещения выполняемой команды, которая находится в нем в течение текущего цикла процессора.

Регистр (РАК), или счетчик (СчАК) адреса команды (program counter — PC) — регистр, содержащий адрес текущей команды.

Регистр адреса (числа) — РА(Ч) содержит адрес одного из операндов выполняемой команды (регистров может быть несколько).

Регистр числа (РЧ) содержит операнд выполняемой команды (регистров также может быть несколько).

Регистр результата (РР) предназначается для хранения результата выполнения команды.

Сумматор — регистр, осуществляющий операции сложения (логического и арифметического двоичного) чисел или битовых строк, представленных в *прямом или обратном коде* (иногда РЧ и РР включают в состав сумматора).

Цикл выполнения команды может выглядеть следующим образом.

1. В соответствии с содержимым СчАК (адрес очередной команды) УУ извлекает из главной, или оперативной, памяти (ОП) очередную команду и помещает ее в РК.

Некоторые команды УУ обрабатывает самостоятельно, без привлечения АЛУ (например, по команде «перейти по адресу 2478» величина 2478 сразу заносится в СчАК, и процессор переходит к выполнению следующей команды.

Типичная команда содержит:

- код операции (КОП), характеризующий тип выполняемого действия (сложение, вычитание и пр. чисел; сравнение строк; передача управления, обращение к ВУ и пр.);
- номера индексного (ИР) и базисного (БР) регистров (в некоторых машинах — адреса слов, ячеек ОП, в которых размещена соответствующая информация);
- адреса операндов А1, А2 и других, участвующих в выполнении команды (чисел, строк, других команд программы).

2. Осуществляется расшифровка (декодирование) команды.

3. Адреса А1, А2 и др. помещаются в регистры адреса.

4. Если в команде указаны ИР или БР, то их содержимое используется для модификации РА — фактически выбираются числа или команды, смещенные в ту или иную сторону по отношению к адресу, указанному в команде.

При этом ИР используются для текущего изменения адреса, связанного с работой программы (например, при обработке массива чисел). БР используется для глобального смещения программы или данных в ОП.

5. По значениям РА осуществляется чтение чисел (строк) и помещение их в РЧ.

6. Выполнение операции (арифметической, логической и пр.) и помещение результата в РР.

7. Запись результата по одному из адресов (если необходимо).

8. Увеличение содержимого СчАК на единицу (переход к следующей команде).

Очевидно, что за счет увеличения числа регистров возможно распараллеливание, перекрытие операций. Например, при считывании команды СчАК можно автоматически увеличить на 1, подготовив выборку следующей команды. После расшифровки текущей команды РК освобождается и в него может быть помещена следующая команда программы. При выполнении операции возможна расшифровка следующей команды и т. д. Все это является предпосылкой построения так называемых конвейерных структур (pipeline).

Система команд

Существовали и существуют различные подходы к выбору систем команд процессоров. Сегодня сложились некоторые представления об оптимальных системах команд.

Адресность команд ЦП тесно связана с *разрядностью* ОП. Типичная команда состоит из фиксированной части (КОП, ИР, БР) и адресной части, в которой указаны адреса операндов. Известны одно-, двух- и трехадресные машины (системы команд). Очевидна связь таких параметров ЦУ, как *длина адресного пространства*, *адресность*, *разрядность*. Увеличение разрядности позволяет увеличить адресность команды и длину адреса (т. е. объем памяти, доступной данной команде). Увеличение адресности, в свою очередь, приводит к повышению быстродействия обработки (за счет снижения числа требуемых команд).

В трехадресной машине, например, сложение двух чисел требует одной команды (извлечь число по А1, число по А2, сложить и записать результат по А3). В двухадресной необходимы две команды (первая — извлечь число по А1 и поместить в РЧ (или сумматор), вторая — извлечь число по А1, сложить с содержимым РЧ и результат записать по А2). Легко видеть, что одноадресная машина потребует три команды. Поэтому неудивительно, что основная тенденция в развитии ЦУ ЭВМ состоит в увеличении разрядности. Кроме перечисленных преимуществ, это обеспечивает также увеличение диапазона значений обрабатываемых чисел и количества информации, извлекаемой за один такт работы машины из ОП или записываемой в ОП. Ранние ПЭВМ имели разрядность 8, объем ОП 64 Кбайт, более последние модели обладают разрядностью 32—64 и объемом ОП 128—1024 Мбайт.

Охарактеризуем группы операций некоторой абстрактной системы команд.

1. *Операции пересылки* — перемещение содержания машинного слова в следующих разновидностях: *регистр—регистр*, *регистр—память*, *память—регистр*, *ОП—ОП*. Каждой из модификаций обычно соответствует уникальный код команды (КОП).

2. *Операции арифметики с фиксированной запятой* (+, –, *, / , % и пр.). Модификации команды возможны те же.

3. *Операции арифметики с плавающей запятой.*

4. *Операции сравнения* содержания машинных слов (в зависимости от результата — >, <, = вырабатывается некий признак или флаг, помещаемый в один из регистров).

5. *Операции условного* и безусловного перехода (условный переход обычно кооперируется с операцией сравнения).

6. *Побитовые операции* с парой машинных слов.

7. *Операции индексной арифметики* — изменения содержания индексных регистров (в некоторых системах — соответствующих ячеек ОП), используются для обращения к последовательным элементам массива.

8. *Операции прерывания* — переход к зарезервированной выделенной заранее области памяти для обработки сбойных, аварийных и других ситуаций.

9. *Операции обращения к внешнему устройству* — поиск блока на магнитной ленте или диске, считывание блока, запись блока на носитель и пр.

Надо заметить, что возможны модификации перечисленных команд для работы с числами двойной длины, строками, векторами и пр.

Ассемблер для ПЭВМ (процессор i8086). Основные понятия

Системы автоматизации программирования прошли длительный путь развития. На первом этапе программисты работали в абсолютных адресах, набивая программы в машинных кодах на картах Холлерита (см. рис. 1.1). Приведем пример двухадресной команды:

```
+10 00 1234 7653
```

(сложить содержимое адреса 1234_8 с содержимым адреса 7653_8 и записать по адресу 7653_8).

Затем появляются библиотеки стандартных программ (БСП). Обращение к подпрограмме из библиотеки по прежнему осуществляется вручную, например командами:

```
-31 77 7777 0021
+00 00 1322 1323
```

(обратиться к подпрограмме № 0021 — перевод числа из десятичного представления в двоичное. Исходное число находится по адресу 1322_8, результат записать по адресу 1323_8).

Следующим этапом является появление ассемблеров и макроассемблеров. В отечественной практике подобные системы в разные периоды получали названия ЯСК (языки символического кодирования), ССК (системы символического кодирования), автокоды и пр.

Языки ассемблерного типа позволяют вместо двоичных форматов машинных команд использовать их мнемонические символьные обозначения (имена). Являясь существенным шагом вперед, ассемблерные языки все еще оставались машинно-зависимыми, а программист все также должен был быть хорошо знаком с организацией и функционированием аппаратной среды конкретного типа ЭВМ. При этом ассемблерные программы все еще затруднительны для чтения, трудоемки при отладке и требуют больших усилий для переноса на другие типы ЭВМ. Однако и сейчас ассемблерные языки используются при необходимости разработки высокоэффективного программного обеспечения (минимального по объему и максимальной производительности).

Программирование на ассемблере. Текст исходной программы состоит из операторов ассемблера, каждый из которых занимает отдельную строку этого текста. Различают два типа операторов: инструкции и директивы. Первые при трансляции преобразуются в команды процессора, которые исполняются после загрузки в память загрузочного модуля программы, имеющего расширение `.com` или `.exe`. Операторы второго типа управляют процессом ассемблирования — преобразования текста исходной программы в коды объектного модуля (расширение `.obj`).

Типичный формат оператора ассемблера имеет следующий вид:

```
[<Метка>:]<Код операции>[<Операнд-1> [,<Операнд-2>]]
[;<Комментарий>],
```

где элементы, указанные в квадратных скобках, могут отсутствовать.

Пробелы вводятся произвольно, но минимум один пробел должен быть после кода операции.

<Метка> — идентификатор, присваиваемый первому байту того оператора, в котором она появляется.

<Код операции> — мнемоническое обозначение соответствующих команд процессора.

<Операнды> оператора ассемблера описываются выражениями. Выражения конструируются на основе операций над числовыми и текстовыми константами, метками и идентификаторами переменных с использованием знаков операций и некоторых зарезервированных слов.

Некоторые операции ассемблера ПЭВМ. Приведем пример оператора ассемблера:

```
l0c_1: mov ax,(DAT_1+4) SHR 4.
```

Здесь использованы такие операции ассемблера, как ( ), + и SHR.

Некоторые арифметические операторы приводятся в табл. 1.17. Более полные списки команд (включая режимы MMX, SSE1-4) можно найти, например, в [16].

Таблица 1.17. **Некоторые арифметические операторы ассемблера ПЭВМ**

Обозначение	Наименование	Синтаксис	Действие	Примечания
+	Бинарный плюс	<Выражение-1> +<Выражение-2>	Суммирует значения двух выражений	Оператор может использоваться для прибавления целого числа к операнду, перемещаемому в памяти. Операндом, перемещаемым в памяти, может быть только одно из выражений. Оба выражения могут быть целыми числами
+	Унарный плюс	+<Выражение>	Не изменяет знак и значение выражения	Унарная операция обладает более высоким приоритетом, чем оператор сложения
-	Вычитание (бинарный минус)	<Выражение-1> -<Выражение-2>	Вычитает значение одного выражения из другого	Операндами могут быть целые числа или операнды, перемещаемые в памяти. Если оба операнда являются адресами памяти, то они должны располагаться в одном сегменте памяти

Окончание табл. 1.17

Обозначение	Наименование	Синтаксис	Действие	Примечания
–	Унарный минус	–<Выражение>	Изменяет знак выражения на противоположный	Унарная операция обладает более высоким приоритетом, чем оператор вычитания
*	Умножение	<Выражение-1> *<Выражение-2>	Перемножение значений двух выражений	Выражения должны быть целыми числами и не могут быть адресами, перемещаемыми в памяти
/	Деление	<Выражение-1> /<Выражение-2>	Деление значения одного выражения на другое	
MOD	Деление по модулю	<Выражение-1> MOD <Выражение-2>	Выдает остаток от деления	

При программировании на макроассемблере можно формировать обращение к часто повторяющейся последовательности команд с помощью одного оператора. Этот прием несколько напоминает вызов подпрограмм в языках высокого уровня, но между ними лежит значительное различие, заключающееся в том, что подпрограмма, занимающая некоторый участок памяти, может быть исполнена неограниченное число раз путем передачи ей управления из вызывающей программы, в которую подпрограмма сама затем возвращает управление. В ассемблере используются вызовы макроопределений.

В развитии инструментального программного обеспечения (т. е. программного обеспечения, служащего для создания программных средств в любой проблемной области) рассматривают пять поколений языков программирования. Языки программирования, как средство общения человека с ЭВМ, от поколения к поколению улучшали свои характеристики, становясь все более доступными в освоении непрофессионалами.

Классификация ЯП

Определяющие факторы классификации обычно жестко не фиксируются. Чтобы продемонстрировать пример типичной классификации, опишем наиболее часто применяемые факторы,

дадим им условные названия и приведем образцы ЯП для каждой из классификационных групп (табл. 1.18).

Таблица 1.18. **Классификация ЯП**

Фактор	Характеристика	Группы	Примеры ЯП
Уровень ЯП	Степень близости ЯП к архитектуре компьютера	Низкий	Автокод, ассемблер
		Высокий	Fortran, Pascal, ADA, Basic, C и др. ЯВУ
		Сверхвысокий	Сетл
Специализация ЯП	Потенциальная или реальная область применения	Общего назначения (универсальные)	Algol, PL/1, Simula, Basic, Pascal
		Специализированные	Fortran (инженерные расчеты), Cobol (коммерческие задачи), Refal, Lisp (символьная обработка), Modula, Ada (программирование в реальном времени)
Алгоритмичность (процедурность)	Возможность абстрагироваться от деталей алгоритма решения задачи. Алгоритмичность тем выше, чем точнее приходится планировать порядок выполняемых действий	Процедурные	Ассемблер, Fortran, Basic, Pascal, Ada
		Непроцедурные	Prolog, Langin

Элементы языков программирования

Элементы ЯП могут рассматриваться на следующих уровнях:

- *алфавит* — совокупность символов, отображаемых на устройствах печати и экранах и/или вводимых с клавиатуры терминала. Обычно это набор символов Latin-1 с исключением управляющих символов. Иногда в это множество включаются неотображаемые символы с указанием правил их записи (комбинирование в лексемы);
- *лексика* — совокупность правил образования цепочек символов (лексем), образующих идентификаторы (переменные и метки), операторы, операции и другие лексические компоненты языка. Сюда же включаются зарезервированные (ключевые) слова ЯП, предназначенные для обозначения операторов, встроенных функций и пр.

Иногда эквивалентные лексемы, в зависимости от ЯП, могут обозначаться как одним символом алфавита, так и несколькими. Например, операция присваивания значения в ЯП Basic обозначается как «=», а в языке Pascal — «:=». Операторные скобки в Си задаются символами «{» и «}», а в Pascal — **Begin** и **End**. Граница между лексикой и алфавитом, таким образом, является весьма условной, тем более что компилятор обычно на фазе лексического анализа заменяет распознанные ключевые слова внутренним кодом (например, **Begin** — 512, **End** — 513) и в дальнейшем рассматривает их как отдельные символы;

- *морфология* — совокупность правил и возможностей варьирования написания лексических единиц, в пределах которых они будут правильно распознаны компилятором. Например, для оператора перехода в некоторых реализациях Basic записи **go**, **goto**, **go to** эквивалентны, а в ЯП FoxPro наименования команд и ключевые слова могут быть сокращены вплоть до 4-х начальных букв (например, эквивалентны команды **report**, **repor**, **repo**);
- *синтаксис* — совокупность правил образования языковых конструкций или предложений ЯП — блоков, процедур, составных операторов, условных операторов, операторов цикла и пр. Особенностью синтаксиса является принцип вложенности (рекурсивность) правил построения конструкций. Это значит, что элемент синтаксиса языка в своем определении прямо или косвенно в одной из его частей содержит сам себя. Например, в определении оператора цикла телом цикла является оператор, частным случаем которого является все тот же оператор цикла;
- *семантика* — смысловое содержание конструкций, предложений языка. Семантический анализ — это проверка смысловой правильности конструкции. Например, если мы в выражении используем переменную, то она должна быть определена ранее по тексту программы, а из этого определения может быть получен ее тип. Исходя из типа переменной, можно говорить о допустимости операции с данной переменной. Семантические ошибки возникают при недопустимом использовании операций, массивов, функций, операторов и пр.
- *зарезервированные (служебные)* слова, имеющиеся в каждом ЯП, могут быть использованы только по своему специальному назначению (как имена функций, операторов, операций и пр.). В табл. 1.19 в качестве примера приводится список зарезервированных слов Basic.

Таблица 1.19. **Зарезервированные слова ЯП Basic**

And	Boolean	Byte	Call	Case	Choose	Const
Currency	Date	Deftype	Dim	Do	Double	Each
Else	End	Eqv	Exit	Explicit	For	Function
Get	GoSub	GoTo	If	Imp	Integer	Is
Let	Like	Long	Loop	Mod	Module	New
Next	Not	Object	Option	Or	Private	Property
Public	ReDim	Return	Select	Set	Single	Static
Stop	String	Sub	Switch	Then	Variant	WEnd
While	With	Xor				

Структура исходной программы на ЯП. Исходная программа (source module), как правило, состоит из следующих частей (впервые эти требования были сформулированы в языке Cobol):

- раздела идентификации — области, содержащей наименование программы, а также дополнительную информацию для программистов и/или пользователей;
- раздела связи — фрагмента текста, описывающего внешние переменные, передаваемые вызывающей программой (если таковая имеется), т. е. ту часть исходных данных, которая обязательно поступает на вход программы при ее запуске. Эти переменные часто называют параметрами программы;
- раздела оборудования (среды) — описания типа ЭВМ, процессора, требований к оперативной и внешней памяти, существенных с точки зрения выполнимости программы. Дело в том, что даже среди семейства однотипных ЭВМ могут существовать отличия в наборе машинных инструкций (команд), средств программирования ввода-вывода, кодированного представления данных, в связи с этим описание среды, приводимое в данном разделе, оказывается необходимым транслятору с языка с точки зрения оптимизации выполнения или вообще оценки возможности создания рабочей программы;

- раздела данных — идентификации (декларации, объявления, описания) переменных, используемых в программе, и их типов. Понятие типа позволяет осуществлять проверку данных на совместимость в операциях еще на этапе трансляции программы и отвергнуть недопустимые преобразования;
- раздела процедур — собственно программной части, содержащей описание процессов обработки данных. Элементами процедуры являются операторы и стандартные функции, входящие в состав соответствующего языка программирования.

Для сравнения предлагаем читателям ознакомиться со структурой программ на более современном ЯП — Pascal (см. рис. 3.1).

Операторы ЯП. К типовым операторам управления вычислительным процессом относятся следующие:

- организация циклов (выполняемых до исчерпания списка или до достижения управляющей переменной заданного значения, или пока выполняются некоторые условия);
- ветвление программы — выполнение альтернативных групп операторов при заданных условиях;
- блоки операторов — группы, выполняемые как целое;
- операторы перехода — условная или безусловная передача управления на определенный оператор, снабженный меткой, или условный/безусловный выход из цикла или блока.

Заметим, что все перечисленные операторы (кроме перехода) принято называть операторами структурного программирования, а языки, в которых операторов перехода нет, — языками структурного программирования.

Важной компонентой управляющих операторов обычно являются логические условия, сопоставляющие значения констант, функций, переменных отношениями типа $\geq$, $\leq$, $>$, $<$, $\neq$ и вырабатывающие решение о продолжении цикла, ветвлении, выходе из блока и т. д. Несколько простых условий объединяются в составные посредством логических операций И, ИЛИ, НЕ.

Операторы присваивания значений выполняют следующие функции:

- пересылку значений переменных, констант, функций в принимающую переменную;
- вычисление значений арифметической (числовой) переменной в рамках существующих в языке правил построения арифметических выражений (обычно включающих символы «( )» — скобки, «+» — сложение, «–» — вычитание, «/» — деление, «*» — умножение, «**» или «^» — возведение в степень);
- вычисление значений строчной (символьной) переменной путем соединения, пересечения, вычисления строк;
- вычисление логических переменных в рамках правил образования логических выражений (обычно включающих скобки и операции И, ИЛИ, НЕ).

Аналогично могут быть выделены типичные группы функций:

- стандартные алгебраические и арифметические — `Sin`, `Cos`, `Sqrt`, `Min`, `Max` и др.;
- стандартные строчные — выделение, удаление подстроки, проверка типа переменной и т. д.;
- нестандартные функции, в том числе описание операций и форматов ввода-вывода данных; преобразование типов данных; описание операций над данными, специфичными для конкретной системы программирования, ОС или типа ЭВМ.

В табл. 1.20 дается обзор основных операторных возможностей ЯП FoxPro, Pascal, Basic и C (Си).

Таблица 1.20. **Основные структуры некоторых ЯП**

Элемент языка	Язык (система) программирования			
	Pascal	Basic	C	FoxPro
Идентификация, связь	**Program** <Имя> (<Параметры>)		**Main** (<Параметры>)	**Procedure** <Имя> **Parameters** <Параметры>
Типы данных	Integer, Real, Boolean	Integer, Boolean, Single, Double	Int, Float, Char, Long, Short	n(umeric), a(lphabetical), l(ogical), date
Агрегаты данных (массивы)	**Array**[1:20] of **Integer**	**Dim** a(20) As String	char a[20]	**Dimension** a(20)
Циклы	**For** ... **Do** **Repeat**...**Until** **While** ... **Do**	**For** ... **Next** **Do** ... **Loop** **While** ... **Whend**	**For**(i=1,i<10, i++),**Do**... **While**	**Do While** **Enddo**

Окончание табл. 1.20

Элемент языка	Язык (система) программирования			
	Pascal	Basic	C	FoxPro
Выход из цикла вовне	`Break`	`Exit For`	`Break`	`Exit`
Переход к заголовку цикла	`Continue`	—	`Continue`	`Loop`
Блоки, составные операторы	`Begin... End`	—	`{}`	`Do..Enddo; If...Endif`
Ветвление	`If...Then... Case...of`	`If...Then Else Select Case`	`If...Then... Else`	`If...Then... Else, Do Case`
Переход на метку	`Goto <Метка>`	`Goto <Метка>`	`Goto <Метка>`	—
Логические операции (НЕ, И, ИЛИ)	`not, and, or`	`not, and, or`	`! && !!`	`.and., .or., .not.`
Пересылка, присваивание	`:=`	`Let, =`	`=`	`=`
Арифметическив операции	`+, –, *, /`	`+, –, *, /, %` (деление нацело)	`+, *, / %`	`+, –, *, /`
Стандартные числовые функции	`Abs(), Sqrt(), Sin(), Exp(), Ln(), Round()`	`Abs(), Sqrt(), Sin(), Cos(), Exp(), Log()`	В стандартном языке — нет	`Abs(), Cos(), Exp(), Max(), Min(), sign()`
Строчные функции	`Pos(), Str()`	`Left(), Right(), Len()`	`strcat strlin`	`+, Substr(), Left(), Right(), Rtrim(), Ltrim()`
Ввод-вывод	`Read(), Readln(), Write()`	`Print(), Input()`	`printf get, put`	`Say, get, ?, ??`
Комментарии	`// текст`	`rem текст`	`/* текст */`	`* текст && текст`

Необходимо отметить, что конкретные ЯП могут не требовать наличия всех выше перечисленных разделов исходного модуля. В некоторых случаях описания переменных могут размещаться произвольно в тексте или опускаться, при этом тип переменной

определяется компилятором, исходя из системы умолчаний; есть средства программирования, в которых тип переменной задается в момент присвоения ей значения другой переменной или константы и т. д. Существуют фрагменты описания данных, которые могут быть отнесены как к разделу данных, так и к разделу оборудования (указания на устройство, длину и формат записи, организацию файла и т. п.).

Подпрограммы. При разработке программ на алгоритмических языках широко используются *подпрограммы*. Подпрограмма — это средство, позволяющее многократно использовать в разных местах основной программы однажды описанный фрагмент алгоритма.

Как уже упоминалось выше, первоначально наборы таких программных заготовок именовались библиотеками стандартных программ (БСП).

В большинстве ЯП не проводится концептуальное различие между такими объектами, как программа и подпрограмма (процедура, функция). В связи с этим всякая подпрограмма может приобретать иерархическую структуру, включая в себя подчиненные (вызываемые) подпрограммы, процедуры и т. д., имеющие стандартный состав.

Объявления (типы, переменные, константы), использующиеся любой подпрограммой, относятся к одной из двух категорий — категории *локальных* объявлений и категории *глобальных* объявлений. Локальные объявления принадлежат подпрограмме, описаны внутри нее и могут использоваться только ею. Глобальные объявления принадлежат программе в целом и доступны как самой программе, так и всем ее подпрограммам. Обмен данными между основной программой и ее подпрограммами обычно осуществляется посредством глобальных переменных.

Если имя глобального объявления совпадает с именем локального, то внутри подпрограммы обычно объявление интерпретируется как локальное, и все изменения, вносимые, например, в значение такой переменной, актуальны только в рамках подпрограммы.

Описание подпрограммы может содержать список аргументов (параметров), которые называются *формальными*. Каждый параметр из списка формальных параметров является локальным по отношению к подпрограмме, для которой он объявлен. Это означает, что глобальные переменные, имена которых сов-

падают с именами формальных параметров, становятся недоступными для использования в подпрограмме.

Все формальные параметры можно разбить на две категории:

- параметры, вызываемые подпрограммой *по своему значению* (т. е. параметры, которые передают в подпрограмму свое значение и не меняются в результате выполнения подпрограммы);
- параметры, вызываемые подпрограммой *по наименованию* (т. е. параметры, которые становятся доступными для изменения внутри подпрограммы).

Главное различие этих двух категорий — в механизме передачи параметров в подпрограмму. При вызове параметра по значению происходит копирование памяти, занимаемой параметром, в *стек* и использование в дальнейшем в операторах подпрограммы локальной копии параметра. Основное значение параметра (глобальное по отношению к подпрограмме) при этом остается без изменения. Следует отметить, что использование такого механизма при передаче, например, массивов большой длины может отрицательно повлиять на быстродействие программы, заполняя стек лишней информацией.

При вызове параметра по наименованию в подпрограмму передается адрес памяти (глобальной по отношению к подпрограмме), в которой размещено значение параметра. Таким образом, в качестве локальной переменной выступает ссылка на глобальное размещение параметра, обеспечивающая доступ к самому значению.

При обращении к подпрограмме формальные параметры заменяются на соответствующие по типу и категории *фактические параметры* вызывающей программы или подпрограммы.

Системы программирования

Рассмотренные выше средства являются важными функциональными компонентами соответствующей системы программирования, т. е. среды окружения программиста, позволяющей ему разрабатывать прикладные программы (программировать приложения, разрабатывать приложения) для соответствующих ЭВМ и операционных систем.

Система программирования (СП) является инструментальным средством программиста — разработчика системных или прикладных программ. Первоначально, в ранних ОС, системы программирования входили в состав ОС и были доступны только «избранным».

Широкое распространение ПЭВМ и доступность СП привели к приобщению широких кругов пользователей к увлекательному занятию — написанию собственных программ в самых различных средах.

Современные системы программирования обычно предоставляют пользователям такие средства разработки программ, как:

- создание и редактирование текстов программ;
- компилятор или интерпретатор;
- библиотеки стандартных процедур и функций;
- утилиты (вспомогательные рабочие программы);
- отладчик (средство, помогающее находить и устранять ошибки в программе);
- редактор связей;
- встроенная справочная служба.

Эти инструменты взаимодействуют между собой через обычные файлы с помощью стандартных возможностей файловой системы.

В табл. 1.21 приводятся различные классификации СП.

Таблица 1.21. **Классы систем программирования (СП)**

Признак классификации	Типы
Набор исходных языков	Одноязыковые Многоязыковые
Возможности расширения	Замкнутые Открытые
Трансляция	Компиляция Интерпретация

Различают системы общего назначения и языково-ориентированные системы.

Системы общего назначения содержат набор программных инструментов (например, текстовый редактор, редактор связей и т. п.), позволяющих выполнять разработку программ на разных языках программирования. Для программирования на кон-

кретном языке программирования требуются дополнительные инструменты, ориентированные на этот язык.

Языково-ориентированные системы предназначены для поддержки разработки программ на каком-либо одном языке программирования, причем построение такой среды базируется на знаниях об этом языке.

Отличительной особенностью многоязыковых систем программирования является то, что отдельные части (секции, модули или сегменты) программы могут быть подготовлены на различных языках и объединены во время или перед выполнением в единый модуль.

В открытую систему можно ввести новый входной язык с транслятором, не требуя изменений в системе.

Трансляция в системе программирования реализуется в форме компиляции или интерпретации. С точки зрения выполнения работы компилятор и интерпретатор существенно различаются. В интерпретирующей системе осуществляется покомандная расшифровка и выполнение инструкций входного языка (в среде данной системы программирования); в компилирующей — подготовка результирующего модуля, который может выполняться на ЭВМ практически независимо от среды. После того как текст программы откомпилирован, исходный текстовый файл уже не участвует в дальнейшем построении и запуске программы, в то время как текст программы, обрабатываемой интерпретатором, должен заново переводиться при каждом очередном запуске. Таким образом, интерпретируемые программы проще исправлять и изменять, но откомпилированные — быстрее работают.

Каждый конкретный язык программирования ориентирован либо на компиляцию, либо на интерпретацию — в зависимости от того, для каких целей он создавался. Например, Pascal обычно используется для решения довольно сложных задач, в которых важна скорость работы программ. Поэтому данный язык обычно реализуется с помощью компилятора. С другой стороны, Basic создавался как язык для обучения алгоритмизации и программированию, в этом случае построчное выполнение программы делает процесс обучения более наглядным. Иногда для одного языка имеется и компилятор, и интерпретатор. В этом случае для разработки и тестирования программы можно воспользоваться интерпретатором, а затем откомпилировать отлаженную программу, чтобы повысить скорость ее выполнения.

Системы программирования обеспечивают весь спектр задач по обработке информации. С их помощью можно решать вычислительные задачи, обрабатывать тексты и графические изображения, осуществлять хранение и поиск данных и т. д. Кроме того, сами системы программирования представляют собой программы, написанные на языках программирования, т. е. созданные с помощью систем программирования.

Популярные системы программирования — Turbo Basic, Quick Basic, Turbo Pascal, Turbo C.

Рассмотрим структуру абстрактной многоязыковой, открытой, компилирующей системы программирования и процесс разработки приложений в данной среде (рис. 1.5).

Ввод. Программа на исходном языке (исходный модуль) готовится с помощью текстовых редакторов и в виде текстового файла или раздела библиотеки поступает на вход транслятора.

Трансляция. Трансляция исходной программы есть процедура преобразования исходного модуля в промежуточную, так называемую объектную форму. Трансляция в общем случае включает в себя препроцессинг (предобработку) и компиляцию.

Препроцессинг — необязательная фаза, состоящая в анализе исходного текста, извлечения из него директив препроцессора и их выполнения.

Директивы препроцессора представляют собой помеченные спецсимволами (обычно %, #, &) строки, содержащие аббревиатуры, символические обозначения и т. д., конструкций, включаемых в состав исходной программы перед ее обработкой компилятором.

Данные для расширения исходного текста могут быть стандартными, определяться пользователем либо содержаться в системных библиотеках ОС.

В качестве примера рассмотрим директивы, определяющие функционирование компилятора в системах программирования Pascal.

Каждая директива компилятора заключается в фигурные скобки и начинается с символа «$», после которого без пробела должно быть указано имя директивы: `{$I+}`, `{$IFDEF}`, `{$ELSE}`.

Различают три вида директив препроцессора:

- ключевые директивы;
- директивы параметров;
- директивы условной компиляции.

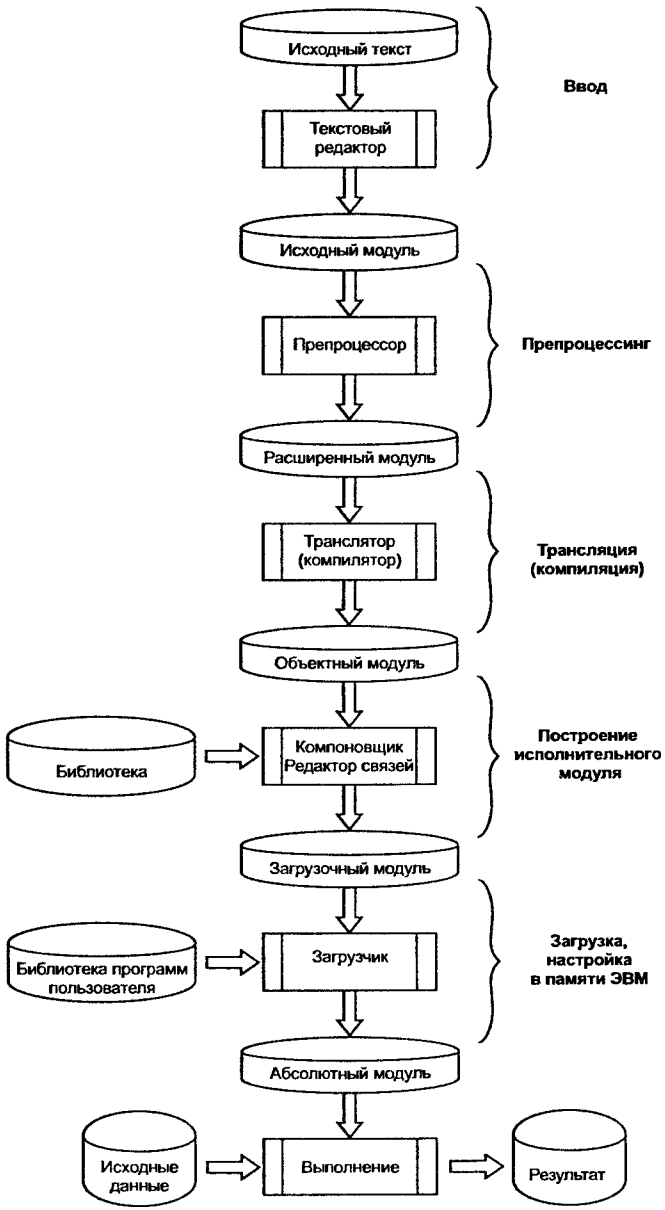


Рис. 1.5. Схема разработки прикладных программ в среде типичной системы программирования

Ключевая директива включает или отключает определенные директивой возможности компилятора. Ключевые директивы могут носить глобальный или локальный характер. Глобальные директивы определяют весь процесс компиляции и должны размещаться в тексте программы до начала всех объявлений. Локальные директивы определяют компиляцию только части кода, могут многократно включать или отключать заданные режимы компиляции и располагаться в любых местах текста программы. Примеры ключевых директив:

- `{$I+}` или `{$I-}` — соответственно включает или отключает контроль ошибок файлового ввода-вывода;
- `{$R+}` или `{$R-}` — директивы компилятора, включающие и отключающие проверку диапазона целочисленных значений и индексов.

Директивы параметров задают значения различных параметров, например имена файлов, размеры отводимой памяти.

Ключевые директивы и директивы параметров обычно имеют установки по умолчанию.

Директивы условной компиляции позволяют в зависимости от задания тех или иных условий компилировать или исключать из программы на стадии компиляции отдельные фрагменты кода. Условная компиляция дает возможность программисту управлять компиляцией программного кода, например, использование во фрагменте кода директив:

```
{$IFOPT Q+}
...
{$ENDIF}
```

позволит компилировать заключенный между директивами текст программы только в том случае, если включена опция `Q` проверки переполнения при целочисленных операциях.

Компиляция — в общем случае многоступенчатый процесс, включающий следующие фазы:

- лексический и морфологический анализ — проверка лексического состава входного текста и перевод составных символов (операторов, скобок, идентификаторов и пр.) в некоторую промежуточную внутреннюю форму (таблицы, графы, стеки, гиперссылки), удобную для дальнейшей обработки;

- синтаксический анализ — проверка правильности конструкций, использованных программистом при подготовке текста;
- семантический анализ — выявление несоответствий типов и структур переменных, функций и процедур;
- генерация объектного кода — завершающая фаза трансляции.

Выполнение трансляции (компиляции) может осуществляться в различных режимах, установка которых производится с помощью ключей, параметров или опций. Может быть, например, потребовано только выполнение фазы синтаксического анализа и т. п.

Объектный модуль (`object module`) представляет собой текст программы на машинном языке, включающий машинные инструкции, словари, служебную информацию.

Объектный модуль не работоспособен, поскольку содержит неразрешенные ссылки на вызываемые подпрограммы библиотеки транслятора (в общем случае — системы программирования), реализующие функции ввода-вывода, обработки числовых и строчных переменных, а также на другие программы пользователей или средства пакетов прикладных программ.

Построение исполнительного модуля (`load module`). Построение загрузочного модуля осуществляется специальными программными средствами. Они именуются по-разному — редактор связей, построитель задач, компоновщик, сборщик, но основной функцией их является объединение объектных и загрузочных модулей в единый загрузочный модуль с последующей записью в библиотеку или файл. Полученный модуль в дальнейшем может использоваться для сборки других программ и т. д., что создает возможность наращивания программного обеспечения.

Загрузка программы. Загрузочный модуль после сборки либо помещается в пользовательскую библиотеку программ, либо непосредственно направляется на исполнение. Выполнение модуля состоит в загрузке его в оперативную память, настройке по месту в памяти и передаче ему управления. Образ загрузочного модуля в памяти называется абсолютным модулем, поскольку все команды ЭВМ здесь приобретают окончательную форму и получают абсолютные адреса в памяти. Формирование абсолютного модуля может осуществляться как программно путем обработки командных кодов модуля программой-загрузчиком (`loader`), так и аппаратно путем применения индексирования и базирования команд загрузочного модуля и приведения указанных в них относительных адресов к абсолютной форме.

Современные системы программирования позволяют удобно переходить от одного этапа к другому. Это осуществляется в рамках так называемой интегрированной среды программирования, которая содержит в себе текстовый редактор, компилятор, компоновщик, встроенный отладчик и в зависимости от системы или ее версии предоставляет программисту дополнительные удобства для написания и отладки программ.

1.5. Разработка и развитие программного обеспечения (ПО)

Основные этапы решения задач на ЭВМ

Работа по решению прикладной задачи на компьютере проходит через следующие этапы:

- постановка задачи;
- математическая формализация;
- построение алгоритма;
- составление программы на языке программирования;
- отладка и тестирование программы;
- анализ полученных результатов.

Технологическая цепочка решения задачи на ЭВМ предусматривает возможность возвратов на предыдущие этапы после анализа полученных результатов (рис. 1.6). Часто в эту цепочку

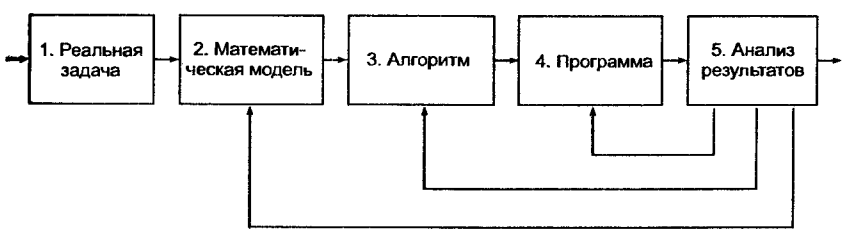


Рис. 1.6. Технологическая цепочка решения задачи на ЭВМ

включают еще один пункт: составление сценария интерфейса (т. е. взаимодействия между пользователем и компьютером во время исполнения программы).

Рассмотрим каждый из этапов.

Постановка задачи. Этап постановки задачи включает:

- формулировку условия задачи;
- определение конечных целей решения задачи;
- описание исходных данных (их типов, диапазонов возможных значений, структуры и т. п.);
- определение формы выдачи результатов.

На этом этапе необходимо четко определить, что именно известно и что требуется получить в результате. Пусть, например, надо найти решение системы линейных алгебраических уравнений с двумя неизвестными (если оно существует и единственно):

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1; \\ a_{21}x_1 + a_{22}x_2 = b_2, \end{cases}$$

или в матричной форме $\mathbf{A} \times \mathbf{x} = \mathbf{b}$.

Постановка такой задачи выглядит следующим образом.

Дано: $\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ — матрица коэффициентов; $\mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$ — вектор свободных членов.

Необходимо найти вектор $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ — решение системы (или пару величин x_1 и x_2).

Исходные данные (элементы матрицы **A** и вектора **b**) — произвольные действительные числа.

Выдача результата должна содержать найденные значения x_1 и x_2 или сообщение о том, что единственного решения не существует.

Математическая формализация. Построение математической модели заключается в формализации способа получения результата из исходных данных, опирается на анализ существующих аналогов и анализ технических и программных средств и включает следующую последовательность шагов:

- разработки математической модели — формального выражения связи между исходными данными и результатом;

- разработки структур данных, поддерживающих преобразование исходных данных в результат.

Компьютер как формальное вычислительное устройство решает задачу, выполняя команды, выраженные на языке программирования. Это становится возможным, если все необходимые для решения задачи действия формализованы, т. е. представлены как последовательность операций (математических, логических, сравнения) над определенными переменными.

Из курса линейной алгебры известно, что система линейных уравнений имеет единственное решение тогда и только тогда, когда определитель матрицы системы отличен от нуля. Это условие записывается следующим образом:

$$\Delta = a_{11} \cdot a_{22} - a_{21} \cdot a_{12} \neq 0.$$

Общеизвестно, что если условие существования и единственности решения выполняется, то для поиска решения системы можно использовать следующие выражения (правила Крамера):

$$x_1 = \frac{\Delta_1}{\Delta} = \frac{b_1 \cdot a_{22} - b_2 \cdot a_{12}}{a_{11} \cdot a_{22} - a_{21} \cdot a_{12}}; \quad x_2 = \frac{\Delta_2}{\Delta} = \frac{b_2 \cdot a_{11} - b_1 \cdot a_{21}}{a_{11} \cdot a_{22} - a_{21} \cdot a_{12}}.$$

Разработка структур данных сводится к выбору структуры для размещения вводимых пользователем коэффициентов системы и свободных членов.

Построение алгоритма. Этап построения алгоритма предполагает формирование строгой и четкой системы правил, определяющей последовательность действий, которая за конечное число шагов должна привести к результату. В этот этап входят:

- выбор формы записи алгоритма (естественный язык, блок-схема, псевдокод);
- проектирование алгоритма.

Выбрав блок-схему как форму записи алгоритма, получаем алгоритм решения поставленной задачи, представленный на рис. 1.7.

Составление программы на языке программирования. Этап составления программы включает следующие шаги:

- выбор языка программирования;
- уточнение способов организации данных;
- запись алгоритма на выбранном языке программирования.

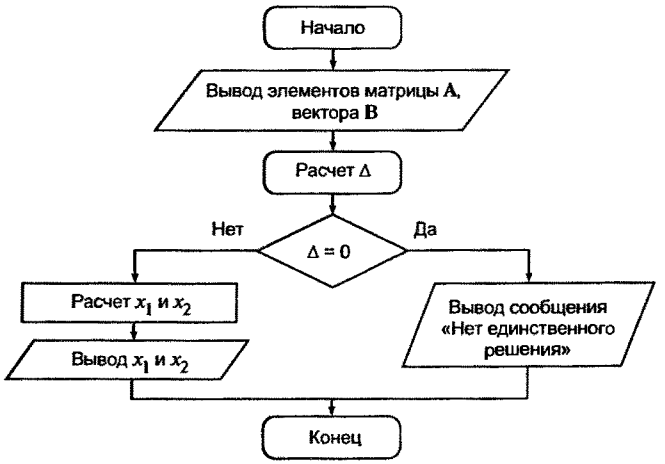


Рис. 1.7. Блок-схема алгоритма решения системы линейных уравнений 2-го порядка

Предположим, что для записи алгоритма выбран популярный язык Basic (см. гл. 2). Тогда программа поиска решения системы может быть, например, такой:

```
Dim A(2,2), B(2)      'Описание исходной матрицы A и вектора b
For I% = 1 to 2
    For J% = 1 to 2              'Ввод элементов матрицы
       Input("Введите коэффициент A["+Str(I%)+","
        + Str(J%)+"]:", A[I%,J%])
    Next J%
Next I%
For I% = 1 to 2                  'Ввод свободных членов
       Input("Введите свободный член B["+Str(I%)+"]:", B[I%]);
Next I%
DELTA = A[1,1]*A[2,2]-A[1,2]*A[2,1]      'Расчет определителя
If DELTA = 0 Then
    Print ("Нет единственного решения")  'Вывод сообщения
Else
    X1 = (B[1]*A[2,2]-b[2]*A[1,2])/DELTA
    X2 = (B[2]*A[1,1]-b[1]*A[2,1])/DELTA
    Print ("X1 = ", X1)    'Вывод значения X1
    Print ("X2 = ", X2)    'Вывод значения X2
End If
End
```

Отладка и тестирование. Отладка программы предполагает следующие действия:

- синтаксическую отладку;
- отладку семантики и логической структуры;
- тестовые прогоны и анализ результатов тестирования.

Под отладкой программы понимается процесс испытания работы программы и исправления обнаруженных при этом ошибок. Обнаружить ошибки, связанные с нарушением правил записи программы на ЯП (синтаксические и семантические ошибки), помогает используемая система программирования. Пользователь получает сообщение об ошибке, исправляет ее и снова повторяет попытку выполнить программу.

Проверка на компьютере правильности алгоритма производится с помощью тестов. Тестом, например, можно назвать конкретный вариант значений исходных данных, для которого известен ожидаемый результат. Например, для программы решения системы линейных уравнений необходимо построить тесты, позволяющие проверить работоспособность как для варианта, когда определитель матрицы A равен нулю, так и для варианта, когда решение системы существует и единственно.

Анализ получаемых результатов. Последний этап — применение разработанной программы для получения искомых результатов. На этом этапе могут быть сделаны выводы о некорректности постановки задачи или разработанной математической модели. В этом случае происходит возврат на этап постановки задачи или на этап математической формализации, что приводит иногда к повторной разработке программы (см. рис. 1.6).

Программы, имеющие большое практическое или научное значение, используются длительное время. Иногда в процессе эксплуатации программы исправляются, дорабатываются, поэтому важным этапом жизни программ является их *сопровождение*, включающее при необходимости доработку программы для решения новых задач, а также составление документации не только по использованию программы, но и по математической модели, алгоритму, набору тестов и т. п.

Цели и задачи разработки программного обеспечения

Рассмотрим далее проблематику разработки программного обеспечения (ПО) на более общем (промышленно-профессиональном) уровне. Говоря юридически, программное обеспече-

ние — это совокупность программ, процедур работы и соответствующей документации для вычислительной системы (ВС).

Стоимость и качество производимого ПО определяются уровнем развития инженерного программирования. Важность инженерного программирования обусловливается следующими двумя тенденциями:

- ПО является сложным изделием и стоимость его постоянно увеличивается;
- ПО оказывает значительное и все возрастающее воздействие на общественное благосостояние.

По разным источникам, стоимость разрабатываемого в мире ПО на рубеже веков составляла от 70 до 120 млрд долл., а в настоящее время — от 150 до 200 млрд долл. Для сравнения: мировой рынок вооружений в 2002 г. оценивался в 30 млрд долл. Мировые тенденции роста стоимости ПО характеризуются тем, что стоимость ПО по отношению к стоимости технических средств вычислительной техники имеет более высокий темп роста. Подобный рост спроса на ПО предъявляет значительные требования к инженерному программированию:

- существенно повысить производительность труда при разработке ПО;
- повысить эффективность сопровождения ПО, так как оно составляет около половины стоимости ПО.

Рост спроса на ПО является следствием того, что автоматизация труда человека с помощью ЭВМ становится все более и более выгодной. Эта тенденция может быть подтверждена следующими данными. В настоящее время в США более половины работающих используют ЭВМ в своей профессиональной деятельности, не обязательно зная как функционируют ТС и ПО.

Общие принципы разработки ПО

Программное обеспечение различается по назначению, выполняемым функциям, формам реализации. В этом смысле всякое ПО — сложная, достаточно уникальная программная система. Однако существуют некоторые общие принципы, которые следует использовать при разработке ПО.

Частотный принцип. Основан на выделении в алгоритмах и в обрабатываемых структурах групп действий и данных по частоте

использования. Для действий, которые чаще встречаются при работе ПО, обеспечиваются условия их наиболее быстрого выполнения. К данным, к которым происходит частое обращение, обеспечивается наиболее быстрый доступ. «Частые» операции стараются делать более короткими.

Принцип модульности. Под модулем в общем случае понимают функциональный элемент рассматриваемой системы, имеющий оформление, законченное и выполненное в пределах требований системы, и средства сопряжения с подобными элементами или элементами более высокого уровня данной или другой системы.

Способы обособления составных частей ПО в отдельные модули могут быть существенно различными. Чаще всего разделение происходит по функциональному признаку. В значительной степени разделение системы на модули определяется используемым методом проектирования ПО.

Принцип функциональной избирательности. Этот принцип является логическим продолжением частотного и модульного принципов и используется при проектировании ПО, объем которого существенно превосходит имеющийся объем оперативной памяти. В ПО выделяется некоторая часть важных модулей, которые постоянно должны быть в состоянии готовности для эффективной организации вычислительного процесса. Эту часть в ПО называют ядром или монитором. В состав монитора, помимо чисто управляющих модулей, должны войти наиболее часто используемые модули. Программы, входящие в состав монитора, постоянно хранятся в оперативной памяти. Остальные части ПО размещаются на внешних запоминающих устройствах и загружаются в оперативную память только по вызову, перекрывая друг друга при необходимости.

Принцип генерируемости. Основное положение этого принципа определяет такой способ исходного представления ПО, который бы позволял осуществлять настройку на конкретную конфигурацию технических средств, круг решаемых проблем, условия работы пользователя.

Принцип функциональной избыточности. Этот принцип учитывает возможность выполнения одной и той же работы (функции) различными средствами. Особенно важен учет этого принципа при разработке пользовательского интерфейса для выдачи

данных из-за психологических различий в восприятии информации.

Принцип «умолчания». Применяется для облегчения организации связей с системой как на стадии генерации, так и при работе с уже готовым ПО. Принцип основан на хранении в системе некоторых базовых описаний структур, модулей, конфигураций оборудования и данных, заранее определяющих условия работы с ПО. Эту информацию ПО использует в качестве заданной, если пользователь забудет или сознательно не конкретизирует ее.

Общесистемные принципы. При создании и развитии ПО рекомендуется применять следующие общесистемные принципы:

- *принцип включения*, который предусматривает, что требования к созданию, функционированию и развитию ПО определяются со стороны более сложной, включающей его в свой состав системы;
- *принцип системного единства*, который состоит в том, что на всех стадиях создания, функционирования и развития ПО его целостность будет обеспечиваться связями между подсистемами, а также функционированием подсистемы управления;
- *принцип развития*, который предусматривает в ПО возможность его наращивания и совершенствования компонентов и связей между ними;
- *принцип комплексности*, который заключается в том, что ПО обеспечивает связность обработки информации как отдельных элементов, так и для всего объема данных в целом на всех стадиях обработки;
- *принцип информационного единства*, т. е. во всех подсистемах, средствах обеспечения и компонентах ПО используются единые термины, символы, условные обозначения и способы представления;
- *принцип совместимости*, который состоит в том, что язык, символы, коды и средства обеспечения ПО согласованы, обеспечивают совместное функционирование всех его подсистем и сохраняют открытой структуру системы в целом;
- *принцип инвариантности*, который предопределяет, что подсистемы и компоненты ПО инвариантны к обрабатываемой информации, т. е. являются универсальными или типовыми.

Взаимодействие человека с системой

Типы пользователей. Можно выделить три квалификационные категории пользователей, которые занимаются разработкой и использованием программного обеспечения:

- разработчики ПО — специалисты в области применения ЭВМ, способные разрабатывать базовые методы, средства и оснащение ПО, общесистемное ПО, инструментальные и технологические средства, осуществлять генерацию и настройку ПО на условия конкретного применения;
- пользователи, которые хорошо знают тонкости построения системы и могут ее модифицировать, т. е. прикладные программисты, которые знают методологию проектирования, алгоритмы прикладной области и могут разрабатывать специализированное ПО, используя общесистемное ПО;
- пользователи, работающие в системе с помощью ориентированного на них языка взаимодействия. Процесс работы в этом случае сводится к заданию исходных данных, постановке задачи, проведению расчетов, анализу результатов и принятию решений.

Психофизиологические особенности взаимодействия человека и ЭВМ. ЭВМ дополняет человека, но не заменяет его, поэтому рассмотрение основных особенностей их сотрудничества является необходимым.

Логический метод рассуждения. У человека он основан на интуиции, на использовании накопленного опыта и воображения. Метод ЭВМ строг и систематичен. Наиболее удачной является композиция, когда ЭВМ реализует отдельные расчетные процедуры, а их логическая последовательность определяется человеком — создателем проектируемого объекта.

Способность к обучению. Человек обучается постепенно, степень «образованности» ЭВМ определяется ее программным обеспечением. Желательно, чтобы содержание информации, получаемой на запрос пользователя, могло изменяться по требованию пользователя.

Обращение с информацией. Емкость мозга человека для сохранения детализированной информации невелика, но обладает интуитивной, неформальной возможностью ее организации. Эффективность вторичного обращения к памяти зависит от време-

ни. В ЭВМ емкость памяти большая, организация — формальная и детализированная, вторичное обращение не зависит от времени. Поэтому целесообразно накапливать и организовывать информацию автоматическим путем и осуществлять ее быстрый вызов по удобным для человека критериям.

Оценка информации. Человек умеет хорошо разделять значимую и несущественную информацию, а ЭВМ таким свойством не обладает. Поэтому должна существовать возможность макропросмотра информации большого объема, что позволяет человеку выбрать интересующую его часть, не изучая всю накопленную информацию.

Отношение к ошибкам. Человек часто допускает существенные ошибки, исправляя их интуитивно, при этом метод обнаружения ошибок чаще всего также интуитивный. ЭВМ, наоборот, не проявляет никакой терпимости к ошибкам и метод обнаружения ошибок строго систематичен. Однако в области формальных ошибок возможности ЭВМ значительно больше, чем при обнаружении неформальных. Поэтому нужно обеспечить возможность пользователю вводить в ЭВМ исходную информацию в свободной форме, написанную по правилам, близким к обычным математическим выражениям и к разговорной речи. ЭВМ выполняет контроль и преобразование информации к стандартному виду, удобному в процедурах обработки и формального устранения ошибок. Затем желательно обратное преобразование этой информации для показа пользователю в наглядной, например, в графической форме, для обнаружения смысловых ошибок.

Обращение со сложными описаниями. Человеку трудно воспринять большое количество информации. Поэтому следует поручать ЭВМ автоматическое разбиение сложных конфигураций на относительно независимые части, охватываемые одним взглядом. Естественно, что изменения, сделанные в одной из этих частей, должны автоматически производиться во всех остальных.

Распределение внимания по многим задачам. Выполнить это условие человеку, в основном, не удается. При решении подзадачи приходится отвлекаться от основной задачи. Поэтому в ЭВМ организована система прерываний, восстанавливающая состояние основной задачи к моменту, нужному для пользователя. Аналогичным образом ЭВМ обслуживает процедуру анализа нескольких вариантов решения.

Память по отношению к проведенной работе. Человек может забыть как то, что уже сделано, так и то, что ему запланировано еще сделать. Этот недостаток компенсирует ЭВМ, которая четко фиксирует и информирует пользователя о выполненных процедурах и предстоящей работе.

Способность сосредоточиться. Эта способность у человека зависит от многих факторов, например, продолжительности и напряжения внимания, влияния среды, общего состояния. Усталостью обусловливается рассеянность, удлинение реакций, нецелесообразные действия. В связи с этим интерактивная система с разделением времени должна адаптироваться к времени реакции отдельного пользователя.

Терпение. При многократном повторении одних и тех же действий человек может испытывать чувство досады. Поэтому предусматривается, например, ввод исходных данных одним массивом при многократном анализе этих данных. К этому же относится включение в системы макрокоманд или гибкие сценарии.

Самочувствие. ЭВМ должна беречь самочувствие пользователя, его чувство собственного достоинства и показывать ему, что именно машина его обслуживает, а не наоборот. Вопросы, ответы и замечания должны соответствовать разговору между подчиненным и его руководителем, определяющим ход и направление процесса проектирования.

Эмоциональность. Это чувство свойственно человеку и чуждо ЭВМ. ПО должно возбуждать у пользователя положительные эмоции и не допускать отрицательных.

Жизненный цикл программного обеспечения ИС (ЖЦ ПО)

ЖЦ ПО — непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации. Основным нормативным документом, регламентирующим ЖЦ ПО, является международный стандарт ISO/IEC 12207 (Международной организации по стандартизации/Международной электротехнической комиссии). Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.

Структура ЖЦ ПО по стандарту ISO/IEC 12207 базируется на трех группах процессов:

- основные процессы ЖЦ ПО (приобретение, поставка, разработка, эксплуатация, сопровождение);
- вспомогательные процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, решение проблем);
- организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого ЖЦ, обучение).

Разработка включает в себя все работы по созданию ПО и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала, и т. д. Разработка ПО включает в себя, как правило, анализ, проектирование и реализацию (программирование).

Эксплуатация включает в себя работы по внедрению компонентов ПО в эксплуатацию, в том числе конфигурирование базы данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала и т. д., и непосредственно эксплуатацию, в том числе локализацию проблем и устранение причин их возникновения, модификацию ПО в рамках установленного регламента, подготовку предложений по совершенствованию, развитию и модернизации системы.

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает выбор методов и инструментальных средств для реализации проекта, определение методов описания промежуточных состояний разработки, разработку методов и средств испытаний ПО, обучение персонала и т. п. Обеспечение качества проекта связано с проблемами *верификации*, *проверки* и *тестирования* ПО. Верификация — это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа. Проверка позволяет оценить соответствие параметров разработки исходным требованиям. Проверка частично совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям. В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом.

Управление конфигурацией является одним из вспомогательных процессов, поддерживающих основные процессы жизненного цикла ПО, прежде всего процессы разработки и сопровождения ПО. При создании проектов сложных ИС, состоящих из многих компонентов, каждый из которых может иметь разновидности или версии, возникает проблема учета их связей и функций, создания унифицированной структуры и обеспечения развития всей системы. Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации конфигурационного учета, планирования и управления конфигурациями ПО отражены в проекте стандарта ISO 12207-2.

Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами. Результатами анализа, в частности, являются функциональные модели, информационные модели и соответствующие им диаграммы. ЖЦ ПО носит итерационный характер — результаты очередного этапа часто вызывают изменения в проектных решениях, выработанных на более ранних этапах.

Модели жизненного цикла

Среди известных моделей жизненного цикла можно выделить следующие:

- *каскадная модель* (до 1970-х гг.) — последовательный переход на следующий этап после завершения предыдущего;
- *итерационная модель* (70—80-е гг.) — итерационные возвраты на предыдущие этапы после выполнения очередного;
- *спиральная модель* (80—90-е гг.) — прототипная модель, предполагающая постепенное расширение прототипа ПО.

Каскадная модель. Для этой модели жизненного цикла характерна автоматизация отдельных несвязанных задач, не требующая выполнения информационной интеграции и совместимости, программного, технического и организационного сопряжения. В рамках решения отдельных задач каскадная модель жизненного цикла по срокам разработки и надежности оправдывала себя. Применение каскадной модели жизненного цикла к большим и сложным проектам вследствие большой длительности процесса проектирования и изменчивости требований за это время может привести к их практической нереализуемости.

Итерационная модель. Создание комплексных систем ПО предполагает проведение увязки проектных решений, получаемых при реализации отдельных задач. Подход к проектированию *снизу вверх* обусловливает необходимость таких итерационных возвратов, когда проектные решения по отдельным задачам комплектуются в общие системные решения и при этом возникает потребность в пересмотре ранее сформулированных требований. Как правило, вследствие большого числа итераций возникают рассогласования в выполненных проектных решениях и документации. Запутанность функциональной и системной архитектуры созданного ПО, трудность в использовании проектной документации сразу вызывают на стадиях внедрения и эксплуатации необходимость перепроектирования всей системы. Длительный жизненный цикл разработки ПО заканчивается этапом внедрения, за которым начинается жизненный цикл создания нового ПО.

Спиральная модель. Используется подход к организации проектирования ПО *сверху вниз*, когда сначала определяется состав функциональных подсистем, а затем постановка отдельных задач. Соответственно сначала разрабатываются такие общесистемные вопросы, как организация интегрированной базы данных, технология сбора, передачи и накопления информации, а затем технология решения конкретных задач. В рамках комплексов задач программирование осуществляется по направлению от головных программных модулей к исполняющим отдельные функции. При этом на первый план выходят вопросы взаимодействия интерфейсов программных модулей между собой и с базой данных, а на второй — реализация алгоритмов.

В основе спиральной модели жизненного цикла лежит применение прототипной технологии или RAD-технологии (Rapid

Application Development — технология быстрой разработки приложений[1]. Согласно этой технологии, ПО разрабатывается путем расширения программных прототипов, повторяя путь от детализации требований к детализации программного кода. Естественно, что при прототипной технологии сокращается число итераций и возникает меньше ошибок и несоответствий, которые необходимо исправлять на последующих итерациях, при этом проектирование ПО осуществляется более быстрыми темпами и упрощается создание проектной документации.

Жизненный цикл при использовании RAD-технологии предполагает активное участие конечных пользователей будущей системы на всех этапах разработки и включает четыре основные стадии информационного инжиниринга:

- *анализ и планирование информационной стратегии.* Пользователи вместе со специалистами-разработчиками участвуют в идентификации проблемной области;
- *проектирование.* Пользователи принимают участие в техническом проектировании под руководством специалистов-разработчиков;
- *конструирование.* Специалисты-разработчики проектируют рабочую версию ПО с использованием языков четвертого поколения;
- *внедрение.* Специалисты-разработчики обучают пользователей работе в среде новой ИС.

Программная документация, вырабатываемая при выполнении проекта

К программной документации относятся следующие виды документов:

1. Техническое задание — это результат сбора и анализа исходных данных исследования конкретной предметной области и работы разработчика для заказчика.

2. Техническое предложение — совокупность рекомендаций по реализации конкретной задачи.

3. Эскизный проект — документ с предварительным определением технических, математических, информационных, про-

[1] *Martin J.* Rapid Application Development. New York: Macmillan, 1991.

граммных, метрологических средств, с организационно-методическим обеспечением.

4. Технический проект — документ, в котором определены перечисленные в эскизном проекте средства.

5. Рабочий проект — завершающий документ, в котором окончательно определены все средства, начиная от технических и кончая организационными методами обеспечения.

6. Паспорт на программное обеспечение и программу.

7. Паспорта на отдельные программные модули.

8. Инструкция системному программисту.

9. Инструкция программисту.

10. Инструкция пользователю.

11. Инструкция по эксплуатации.

12. Листы изменения.

Реинжиниринг бизнес-процессов. CASE-технологии

Разработка и развитие *корпоративных АИС* заключаются во встраивании *информационных технологий (ИТ)* в *деловые процессы организации (бизнес-процессы)* в качестве органической и неотъемлемой части последних [6].

Это встраивание является основанием для переосмысливания и переструктуризации самих бизнес-процессов — реинжиниринг бизнес-процессов (РБП) или в оригинале — business process reengineering (BPR).

Проектирование таких ИС (ранее определяемых в отечественной практике как АСУП или ОАСУ) всегда содержало декларации о включении человека в эти системы. Если для некоторой информационно-справочной системы общего назначения ее пользователь мог (пусть с натяжкой) выступать как элемент, внешний по отношению к системе, то рассматриваемые ИС по своей сути — *человеко-машинные информационно-управляющие системы*. Этот факт часто упускается еще на этапах анализа и построения общей архитектуры ИС. (Выражение «пользователь системы» дополнительно может подталкивать к концептуальной ошибке.) Теперь, когда в центр бизнес-реинжиниринга ставится всемерная поддержка, усиление информационных и аналитических возможностей деятельности каждого работника, какое-либо отделение ИС от функционирования предприятия в целом становится неприемлемым. В силу этого в процессах проектирования целесообразно считать, что корпоративная ИС составляет информационно-управляющую систему, включающую бизнес-архитектуру предприятия, его персонал, используемую ИТ-архитектуру.

Это положение позволяет точнее определить расширяющиеся границы корпоративной ИС. Следует исходить из того, что в виде ИС проектируется часть предприятия, которая непосредственно осуществляет «бизнес», т. е. организационно-производственную деятельность.

Основные недостатки каскадных схем:

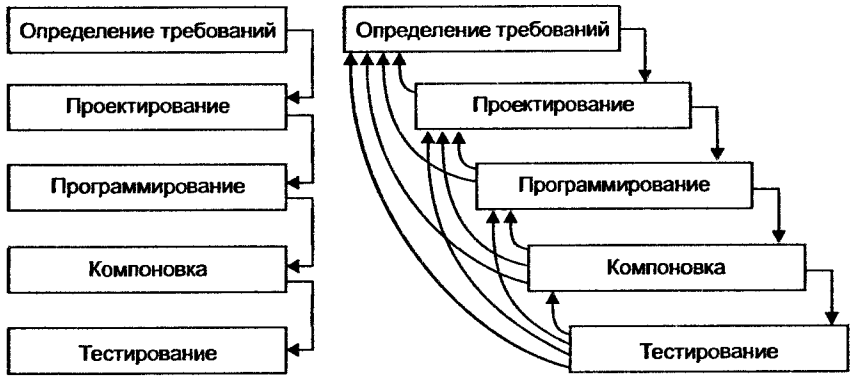
- *отставание* — существенное запаздывание с получением результатов;
- *бесполезность или вред* — и в зарубежной, и в отечественной литературе практики и аналитики оценивали проектирование ИС как действие, очень часто ведущее к примитивной автоматизации (по сути — «механизации») существующих производственных функций работников. Дж. Мартин отмечает, что «... легче идти по проторенной дорожке документирования сложившегося бумажного потока, чем определять насущные потребности бизнеса».

Таким образом, фиксировались неправильные способы работы, возможно, приносящие значительный вред предприятию (в отечественной практике АСУ популярным было выражение «автоматизация беспорядка»);

- *жесткость* — из-за определяющего влияния иерархических структур на процессы и результаты проектирования ИС для представления в ИС функций и данных применявшиеся подходы получили общее условное название «структурное проектирование». Однако жесткость иерархических структур ограничивает их полезность, и чем дальше, тем менее допустимы эти ограничения.

Один из итогов заключается в том, что в значительной степени именно новейшие достижения в ИТ давали потребителям новые возможности предъявлять более высокие требования к производителям и стимулировать конкуренцию.

Приведем две иллюстрации, отражающие принципиальные недостатки каскадной схемы и разницу между идеализированными и фактическими процессами проектирования АИС. На рис. 1.8, *а* показан идеальный вариант каскадной схемы, по ко-



а *б*

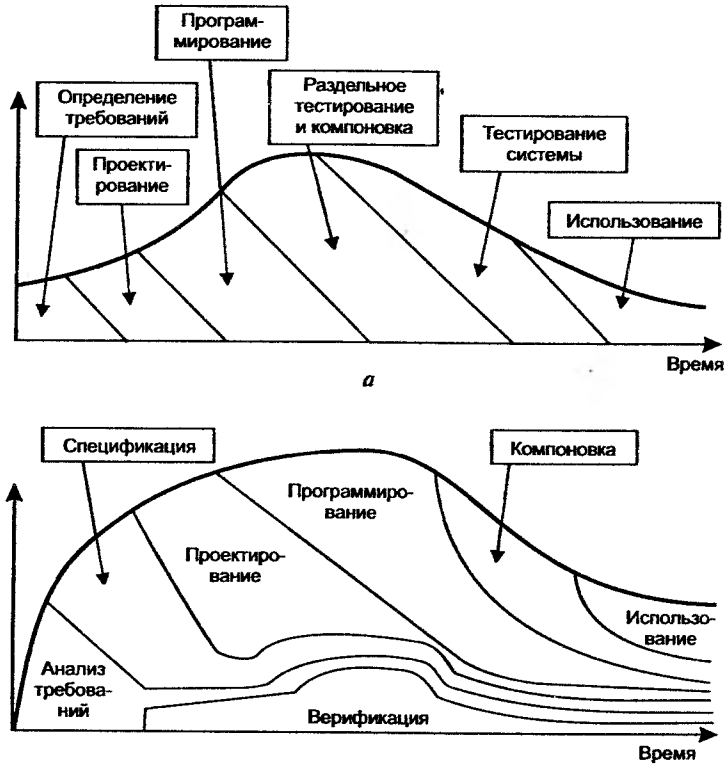
Рис. 1.8. Идеальный (*а*) и реальный (*б*) процессы разработки по каскадной схеме

торой полагается проектировать ИС. Рис. 1.8, *б* иллюстрирует реальные итерации, заставляющие возвращаться к этапам проектирования, и определения требований даже в ходе работ по комплексному тестированию ИС.

На рис. 1.9, *а* показано плановое распределение специалистов, которые должны были бы работать (при последовательном, конвейерном стиле) на разных этапах каскадного проектирования. На рис. 1.9, *б* приведена соответствующая схема Э. Ферентино, из которой видно, что группа, определяющая требования пользователей и разрабатывающая внешние спецификации системы, работает постоянно на всем цикле жизни системы, выполняя корректирующие и контролирующие функции. С тех пор требования к параллельности и спиральности проектирования, комплексности групп разработчиков возросли. Тем не менее многие управляющие разработками ПО считают верной схему рис. 1.9, *а*.

Автоматизация проектирования ПО (CASE-технологии). Перечисленные факторы способствовали появлению программно-технологических средств специального класса — CASE-средств, реализующих CASE-технологию создания и сопровождения ПО. Термин CASE (Computer Aided Software Engineering) используется в настоящее время в весьма широком смысле.

Появлению CASE-технологии и CASE-средств предшествовали исследования в области методологии программирования.



а

б

Рис. 1.9. Распределения людских ресурсов при разработке программной системы: *а* — конвейерное (1970 г.); *б* — реальное (схема Э. Ферентино, 1982 г.)

Программирование обрело черты системного подхода с разработкой и внедрением языков высокого уровня, методов структурного и модульного программирования, языков проектирования и средств их поддержки, формальных и неформальных языков описаний системных требований и спецификаций и т. д.

CASE-технология представляет собой методологию проектирования ПО, а также набор инструментальных средств, позволяющих в наглядной форме моделировать предметную область, анализировать эту модель на всех этапах разработки и сопровождения ИС и разрабатывать приложения в соответствии с информационными потребностями пользователей. Большинство существующих CASE-средств основано на методологиях структурно-

го (в основном) или объектно-ориентированного анализа и проектирования, использующих спецификации в виде диаграмм или текстов для описания внешних требований, связей между моделями системы, динамики ее поведения и архитектуры программных средств [7, 16].

Непроцедурные описания технологий

В то время как алгоритмы и их описания лежат в основе языков программирования процедурного типа, в качестве альтернативы для описания процессов и технологий можно использовать непроцедурные представления или объектно-ориентированные языки, являющиеся атрибутом CASE-технологий.

Диаграммы деятельности (UML-диаграммы). Примером является UML (Universal Modeling Language) — универсальный язык моделирования, который был разработан с целью создания оптимального и универсального языка для описания как предметной области, так и конкретной задачи программирования.

На рис. 1.10 приводится диаграмма деятельности (activity diagram), описывающая процесс заключения договора о страховании. В этом процессе участвуют — клиент, страховщик, кассир и начальник. Входными данными являются документы клиента. Выходными — страховой полис. На диаграмме деятельности изображаются объекты. Данные объекта представляются атрибутами, а его поведение — операциями.

SADT-диаграммы. Это графический язык описания функциональных систем (Structured Analysis and Design Teqnique — Технология Структурного Анализа и проектирования). Следующим этапом развития SADT считается методология IDEF0.

IDEF0 — Integration Definition For Function Modeling — методология функционального моделирования, в основе которой лежит представление любой изучаемой и (или) описываемой системы в виде набора взаимодействующих и взаимосвязанных блоков, отображающих процессы, операции, действия. Каждому процессу (операции, действию) ставится в соответствие блок. На IDEF0-диаграмме, основном документе при анализе и проектировании систем, блок представляет собой прямоугольник (см. рис. 1.11). Каждый функциональный блок в рамках единой рассматриваемой системы должен иметь свой уникальный идент-

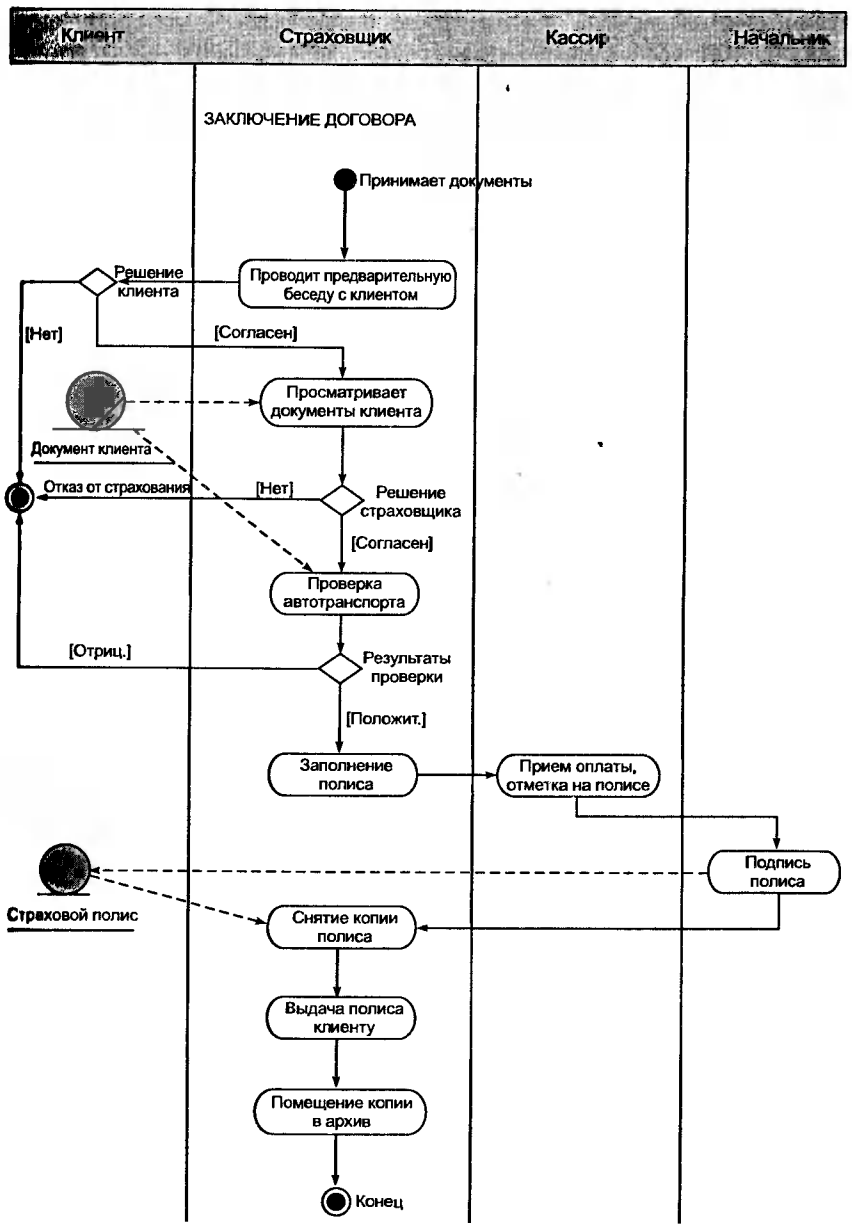


Рис. 1.10. UML-диаграмма

фикационный номер. Интерфейсы, посредством которых блок взаимодействует с другими блоками или с внешней по отношению к моделируемой системе средой, представляются дугами (стрелками), входящими в блок или выходящими из него.

Стороны блока трактуются следующим образом:

- верхняя сторона имеет значение «Управление» (Control);
- левая сторона имеет значение «Вход» (Input);
- правая сторона имеет значение «Выход» (Output);
- нижняя сторона имеет значение «Механизм» (Mechanism).

Схема кодирования дуг ICOM получила название по первым буквам английских эквивалентов слов вход (Input), управление (Control), выход (Output), механизм (Mechanism).

На рис. 1.11 приведен пример модели IDEF0 (своего рода вариант рис. 1.10) для процесса заключения договора о страховании.

На входе системы страхователь будет:

- подавать документы (*а*);
- задавать вопросы (*б*);
- подавать заявление о выплате страхового возмещения (*в*).

На выходе он может получить:

- ответ на вопрос (*г*);
- страховой полис (*д*);
- страховое возмещение (*е*);
- отказ в страховании (*ж*);
- отказ в выплате страхового возмещения (*з*);

Кроме этого, он сам может отказаться от заключения договора.

Информационные потоки и процедуры здесь следующие:

- вопросы и документы клиентов могут поступать как в договорный отдел (*1*), так и в отдел страховых выплат (*2*);
- при заключении договора сотрудник договорного отдела оформляет полис. Клиент вносит деньги и получает полис. На этом этапе после рассмотрения документов возможен также отказ в страховании;
- заявление на выплату рассматривается в отделе страховых выплат. Туда же подаются акты, протоколы и другие документы, подтверждающие факт наступления страхового случая. Заявление утверждает начальник центра (*3*). После рассмотрения документов возможен отказ в страховом возмещении;

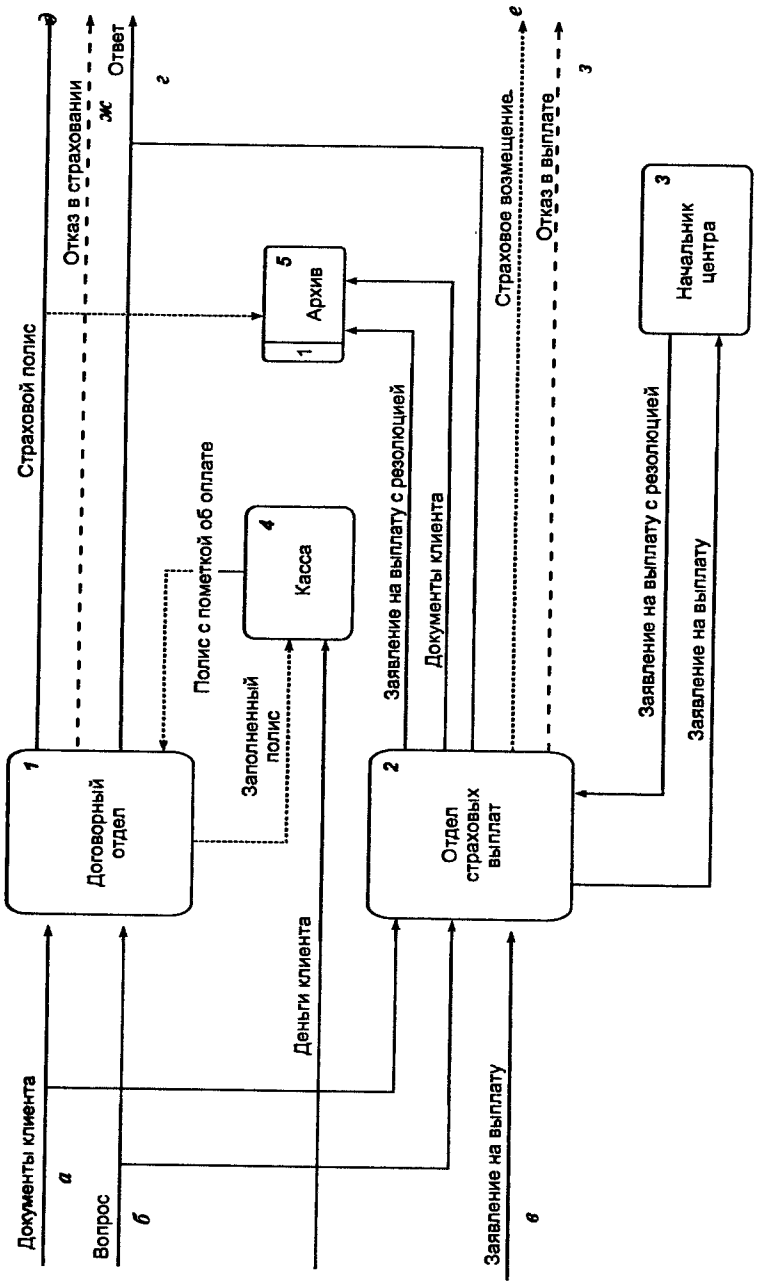


Рис. 1.11. Модель процесса на IDEF0-диаграмме

- после заключения договора деньги клиента проходят через кассу (*4*), оформленные документы помещаются в архивное хранилище (*5*).

Контрольные вопросы

1. Дайте классификацию информации.
2. Перечислите методы кодирования символов.
3. Перечислите методы кодирования численной информации.
4. В чем заключаются особенности двоичной арифметики?
5. Переведите 32451_{10} в шестнадцатеричную и восьмеричную системы счисления.
6. Переведите 32451_{16} в десятичную и восьмеричную системы счисления.
7. В чем заключаются особенности двоичной арифметики?
8. Подсчитайте произведение $1FA_{16}$ и $2BC_{16}$ по модулю 8.
9. Подсчитайте сумму 457_8 и 375_8 по модулю 3.
10. Составьте таблицы истинности для левого $(\overline{A \wedge B})$ и правого $(\overline{A} \vee \overline{B})$ выражений 1-го закона де Моргана. Проверьте их на соответствие.
11. Составьте таблицы истинности для левого $(\overline{A \vee B})$ и правого $(\overline{A} \wedge \overline{B})$ выражений 2-го закона де Моргана. Проверьте их на соответствие.
12. Составьте таблицу истинности для $(A \wedge B) \vee (\overline{A} \wedge \overline{B})$ с учетом значения `null`.
13. Проверьте выполнимость законов де Моргана с учетом значения `null`.
14. Последний столбец таблицы истинности для двухместных операций, очевидно, может содержать $16 = 2^4$ различных сочетаний «1» и «0». Следовательно, всего может быть определено 16 логических операций над двумя переменными, из которых нами рассмотрены только пять. Составьте таблицу истинности для одной из 9 оставшихся вне рассмотрения функций и попытайтесь построить логическое выражение для этой функции.
15. Перечислите базовые структуры алгоритмов и программ.
16. Нарисуйте блок-схему алгоритмов получения произведения n чисел, размещенных в некотором массиве ($n = 15$).
17. Изобразите блок-схему алгоритма определения максимального элемента массива **A** размерности 20.
18. Расширяя возможности алгоритма определения максимума, нарисуйте блок-схему алгоритма сортировки массива **A** размерности 15 путем использования вложенных циклов.
19. Приведите примеры задач, для реализации которых применимы: а) линейные алгоритмы; б) разветвляющиеся алгоритмы; в) циклические алгоритмы.
20. Охарактеризуйте разницу между циклом типа `До` и циклом типа `Пока`.

21. Приведите примеры задач, для реализации которых целесообразно применять циклические структуры: а) с постусловием; б) с предусловием.
22. Изобразите блок-схему алгоритма определения минимального числа в последовательности из *n* произвольных чисел.
23. Изобразите блок-схему алгоритма определения суммы квадратов первых *N* чисел натурального ряда.
24. Постройте блок-схему простого диалогового алгоритма, который обращается к пользователю с просьбой ввести сначала строку **Имя**, а затем строку **Настроение**. В результате диалога на экране может появиться следующий общий совместный текст:

```
Программа> Здравствуйте! Как Ваше имя?
Пользователь> Гаврик
Программа> Доброе утро, Гаврик! Как настроение?
Пользователь> так себе
Программа> У меня тоже так себе, Гаврик!
```

25. Модифицируйте алгоритм вычисления суммы квадратов первых *n* чисел натурального ряда для вычисления суммы квадратов: а) только четных чисел (до *n*); б) только нечетных чисел (до *n*).
26. Охарактеризуйте общие принципы разработки ПО.
27. Охарактеризуйте стадии жизненного цикла ПО.
28. Что такое модель жизненного цикла и каковы ее разновидности?
29. Перечислите недостатки моделей жизненного цикла.
30. В чем состоит сущность CASE-технологий?

Глава 2
ЯЗЫК ПРОГРАММИРОВАНИЯ BASIC

В прошлом веке один английский миссионер, желая облегчить контакт с туземным населением, выделил из английского языка самую простую и распространенную его часть, содержащую около 300 слов и почти не имеющую грамматики. Это подмножество языка, названное «Basic English», действительно оказалось весьма простым для усвоения и поэтому вскоре завоевало популярность не только среди туземцев, но и иммигрантов.

Подобную цель создания средства для расширения и облегчения контактов, только не между различными группами людей, а между непрофессионалами и ЭВМ, поставили перед собой сотрудники Дартмутского колледжа Джон Кемени и Томас Куртс. Разработанный ими в 1964 г. язык BASIC (Basic), как и всякий другой язык программирования, является формальной знаковой системой, используемой для связи человека с ЭВМ и предназначенной для описания данных и алгоритмов их обработки на вычислительной машине. Название BASIC является аббревиатурой английской фразы «Beginner's All-purpose Symbolic Instruction Code», что в переводе означает «многоцелевой язык символических команд для начинающих». (Злые языки утверждали, что изобретатели сначала придумали название для нового, простого в изучении языка, а затем уже такую расшифровку).

И сегодня, спустя 40 лет после изобретения Basic, он остается самым простым для освоения из десятков языков общецелевого программирования, имеющихся в распоряжении любителей программирования. Более того, он прекрасно справляется с работой.

Несмотря на наличие мощных профессиональных языков Си и Pascal, даже на их фоне Basic считается развитым языком, снабженным мощными средствами решения специфических задач, которые обычно большинство пользователей решают с помощью небольших компьютеров, а именно работая с файлами и выводя тексты и графическое изображение на экран дисплея.

Для многих мини- и микроЭВМ Basic предназначался в качестве единственного языка программирования высокого уровня. Это обстоятельство привело к появлению различных его версий, включающих в себя многочисленные эффективные средства программирования из других алгоритмических языков. Таким образом, на сегодняшний день мы имеем не конкретный Basic, а целую группу однотипных диалоговых языков, называемых этим именем.

Приведем краткое описание ЯП Basic на примере его современной реализации — Visual Basic.

2.1. Примеры программ на ЯП Basic

Рассмотрим вначале простейшую диалоговую программу:

```
Print("Здравствуйте! Как Вас зовут?")
Input(Nam$)
If Nam$ = "*" Then Exit Do
If Nam$ <> "" Then
      Do
      Print("Добрый день, "+ Nam$);
      Print("Как настроение?");
      Input(nastr$);
      If nastr$ = "*" Then Exit Do
      If nastr$ <> "" Then
      Print("У меня тоже "+nastr+", "+Nam);
      End If
      Loop
Loop
```

Программа состоит из двух вложенных циклов **Do...Loop**. Выход из внешнего и внутреннего циклов происходит при условии ввода в строке `Nam` или `Nastr` символа `"*"` (звездочка) с помощью оператора **`Exit Do`**.

При попытке ввести пустую строку в ответ на запрос программа возвращается в заголовок, соответственно, внутреннего или внешнего цикла и повторяет запрос на ввод строки. Управление условными переходами осуществляется условным оператором **`If`**.

Процедуры `Print()` и `Input()` осуществляют соответственно вывод на экран и ввод с клавиатуры строк текста. Строки ограничиваются символами " (двойные кавычки). Пустая строка выглядит как "". Конкатенация (слияние) строк осуществляется операцией «+».

Условный оператор, представленный здесь, имеет структуру **`If ... End If`**. Логические условия, представленные в примере, образуются с помощью операций сравнения «равно» (=) и «не равно» (<>).

В программе создаются и используются две переменные пользователя строкового типа — `Nastr` и `Nam`. Как это видно из текста, компилятор Basic не различает строчные и прописные символы, поэтому написания `Nam` и `nam` эквивалентны. Кроме того, в данной программе отсутствуют описания (объявления, декларации) переменных. Это возможно, поскольку Basic, подобно ЯП Fortran, позволяет определить тип в тексте программы непосредственно. В данном примере суффикс $ говорит, что данная переменная — строковая (суффикс %, наоборот, задает числовой тип).

Рассмотрим далее программу сортировки элементов одномерного массива по возрастанию.

```
Dim W_Array%(100), I as integer, J as integer, I_min as integer
Input('Введите количество элементов массива (N<100), N%)
For I = 1 to N%     'Ввод элементов массива
  Input("Введите элемент массива №"+Str(I)+ ":", W_Array[I]);
Next I
For I = 1 to N%     ' Внешний цикл прохода по элементам массива
     I_min= I;      ' Начальное задание индекса
                    ' минимального элемента
     For J = I + 1 to N%  ' Внутренний цикл поиска
                          ' максимального элемента
                          ' в пределах от I + 1 до N
     If W_Array%[J] < W_Array%[I_min] Then I_min = J
     Next J
     Swap(W_Array%[I_min], W_Array[I])    ' Перестановка
                                          ' элементов: I-го
                                          ' и максимального
Next I
For I = 1 to N% do      ' Вывод отсортированного массива
     Print("Элемент массива №"+Str(I)+ "= ", W_Array[I])
Next I
```


Структурно программа состоит из следующих компонентов:

- цикла ввода исходного массива;
- двух вложенных циклов сортировки массива;
- цикла вывода результатов сортировки.

В результате работы программы на экране может быть отображен следующий диалог:

```
Введите количество элементов массива> 10
Введите элемент массива № 1>2
Введите элемент массива № 2>11
Введите элемент массива № 3>3
Введите элемент массива № 4>1
Введите элемент массива № 5>4
Введите элемент массива № 6>16
Введите элемент массива № 7>5
Введите элемент массива № 8>12
Введите элемент массива № 9>7
Введите элемент массива № 10>8
16
12
11
8
7
5
4
3
2
1
```

Циклы в данном примере организуются вложенными операторами **`For I=1 to N% ... Next I`** и **`For J=1 to N% ... Next J`**. В качестве условного оператора здесь используется упрощенная конструкция **`If`** (без **`End If`**), так как здесь при условии выполняется простой (а не составной, как в первом примере) оператор.

В программе появились объявления переменных — массива целых чисел и двух целых переменных. Из текста программы ясно, что обращение к элементу массива осуществляется по индексу, заключенному в квадратные скобки. Комментарий начинается с символа ' (одинарная кавычка).

Перестановка элементов массива осуществляется функцией `Swap()`, что избавляет от необходимости совершать последовательность пересылок.

2.2. Переменные и типы данных

Лексика языка программирования Basic

При записи текстов программ на языке Basic разрешается использовать прописные (заглавные) и строчные буквы латинского алфавита (A B C D E F G H I J K L N O P Q R S T U V W X Y Z a b c d e f g h i j k l n o p q r s t u v w x y z), знак подчеркивания «_», цифры (0 1 2 3 4 5 6 7 8 9) и ограничители.

Ограничители представляют собой знаки пунктуации, знаки операций, разделители и служебные (зарезервированные) слова. Назначение некоторых ограничителей приводится в табл. 2.1.

Таблица 2.1. **Некоторые ограничители ЯП Basic**

Знаки	Назначение
'	Апостроф — признак комментария. Текст после апострофа до конца строки поясняет алгоритм и не является его частью
,	Разделение списков значений, параметров процедуры и функции
( )	Задание индексов массива, выделение части выражения, задание списков параметров
=	Знак оператора присваивания
!	Обозначение типа данных `Single`, служебный знак символьного образца
#	Обозначение типа данных `Double`, служебный знак символьного образца
@	Обозначение типа данных `Currency`
$	Обозначение типа данных `String`
%	Обозначение типа данных `Integer`
&	Обозначение типа данных `Long`
?	Служебный знак символьного образца
*	Служебный знак символьного образца
[]	Ограничители множества значений для символьного образца
_	Признак переноса оператора на следующую строку
.	Десятичная точка, разделитель целой и дробной частей
;	Разделитель выражений в операторах ввода-вывода
:	Разделитель операторов в одной строке

Знаки операций представлены следующим набором ограничителей:

```
+  -  /  *  ^  =  >  <  \  '
```

В качестве разделителя выступает знак пробела.

Компилятор Basic-программ не различает строчные и прописные буквы, поэтому запись тех или иных идентификаторов программистом буквами разных регистров обычно используется для целей структурирования и удобочитаемости текста, что и очевидно из вышеприведенных примеров.

Переменные. В Visual Basic переменные именуются с помощью идентификаторов, длина которых может достигать 255 символов (они должны начинаться с буквы, за которой могут следовать другие буквы, цифры или символы подчеркивания). Регистр символов и наименований переменной значения не имеет.

Типы данных

В языке Visual Basic, как и в языке Pascal, типы данных (табл. 2.2) делятся на *простые* (или *базовые*) и *структурированные*.

К простым (базовым) типам в языке Visual Basic относятся:

- целый;
- вещественный;
- логический.

К стандартным структурированным типам относятся:

- дата;
- массив;
- строка;
- объект.

Целый тип данных. В стандарте языка Basic определен единственный целый тип данных `Integer`. В реализации языка Visual Basic целый тип данных представлен рядом разновидностей, приведенных в табл. 2.2.

Логический (булевский) тип данных. Данные логического типа (`Boolean`) в стандарте языка могут принимать одно из двух значений: `true` или `false`. Переменная или константа логического типа занимает в Visual Basic два байта, в которые записывается «0», если переменная или константа имеет значение `False`, и любое целое, отличное от «0», в противном случае.

Таблица 2.2. **Типы данных**

Название типа	Занимаемый размер, байты	Область изменения данных
	Целый тип	
`Byte` (целое без знака)	1	`0 .. 255`
`Integer` (целое со знаком)	2	`-32768 .. 32767`
`Long` (целое со знаком)	4	-2^{31} .. $2^{31} - 1$
	Логический тип	
`Boolean`	2	`true .. false`
	Вещественный тип	
`Single`	4	`-3.402823E38 .. -1.401298E-45` — для отрицательных значений; `1.401298E-45 .. 3.402823E38` — для положительных значений
`Double`	8	`-1.79769313486232E308 .. -4.94065645841247E-324` — для отрицательных значений; `4.94065645841247E-324 .. 1.79769313486232E308` — для положительных значений
`Currency`	8	-922 337 203 685 477.5808 .. 922 337 203 685 477.5807
`Decimal`	14	`-79 228 162 514 264 337 593 543 950 335 .. 79 228 162 514 264 337 593 543 950 335` — для целых чисел; `-7.9228162514264337593543950335 .. 7.9228162514264337593543950335` — для чисел с фиксированной точкой (28 знаков после десятичной точки); Минимальное, отличное от 0, число: +/-0.0000000000000000000000000001
	Тип данных Дата	
`Date`	8	От January 1, 100 до December 31, 9999
	Строковый тип	
`String` (переменной длины)	10 + длина строки	От 0 до приблизительно 2 миллиардов символов
`String` (фиксированной длины)	Длина строки	От 1 до приблизительно 65 400 символов

Когда прочие числовые типы данных преобразуются в тип `Boolean`, значение «0» воспринимается, как `false`, а любое другое значение становится значением `true`. Если значения типа `Boolean` преобразуются в значения других (числовых) типов, то значение `false` преобразуется в «0», а `true` в «–1».

Вещественный тип данных. В Visual Basic определены шесть стандартных вещественных типов. Каждый тип характеризуется своей областью изменения возможных значений.

Выбор конкретного типа для переменной связан с требуемой точностью вычислений.

Тип данных Дата. Переменные типа Дата (`Date`) представляют собой 8-байтовые представления в форме с плавающей точкой календарных дат в интервале от 1 января 100 года до 31 декабря 9999 года с составляющей времени в интервале от 0:00:00 до 23:59:59.

Константы типа `Date` должны справа и слева ограничиваться знаком «#», например `#January 1, 2002#`.

Строковый тип данных позволяет хранить последовательности символов — строки (`String`). Строки могут быть переменной и фиксированной длины

Теоретически такой тип данных позволяет хранить строковые переменные длиной до 2 млрд символов. Однако на конкретном компьютере это число может быть гораздо меньше из-за ограниченных объемов оперативной памяти или ресурсов операционной системы.

Строки фиксированной длины. Строки фиксированной длины представляют собой специальный тип строки, длина которой ограничена. Подобные переменные создаются с помощью оператора `Dim`. Например:

```
Dim ShortString As String * 10
Dim strShort As String * 10
MyStringVariable$="My string"
```

Тип данных `Variant`. Данный тип добавлен в Visual Basic 5 из версии 2.0. Переменная типа `Variant` может содержать данные любого типа. Если для переменной не объявлен тип данных, то по умолчанию используется тип данных `Variant`.

Тип информации, хранимой в переменной, при этом не имеет значения, поскольку `Variant` может принять любой тип дан-

ных (численный, дата/время, строковый): Visual Basic автоматически производит необходимые преобразования данных. С другой стороны, можно использовать встроенные функции для проверки типа данных, хранящихся в переменной типа Variant.

Использование такого типа данных, как Variant, замедляет работу программы, так как требуется время и ресурсы для выполнения преобразований типов.

Если в программе для некоторой переменной MyVar записаны операторы

```
MyVar = 5
MyVar = MyVar + 1
MyVar = "String value"
MyVar = UCase(MyVar),
```

то фактический тип переменной в каждом операторе будет определяться выражением в правой части.

Тип данных Массив. Язык позволяет определить две разновидности массивов: статический и динамический. Границы статического массива устанавливаются на этапе разработки и могут изменяться только в новой версии программы. Динамические массивы изменяют свои границы в ходе выполнения программы. С их помощью можно динамически задавать размер массива в соответствии с конкретными условиями.

Статический массив. Для объявления здесь используется оператор Dim с указанием максимального значения индекса массива в круглых скобках после его имени:

```
Dim NameArray (100) As String
```

В этом случае элементы переменной NameArray различают не по имени, а по индексу:

```
NameArray(4) = "Иванов"
```

Статические массивы определяются только глобально — их нельзя определить локально внутри процедуры.

В Visual Basic индексирование массива начинается с нуля, т. е. индекс 0 обозначает первый элемент массива, индекс 1 — второй и т. д.

Оператор Option Base позволяет задать индексацию массива с 1:

```
Option Base 1
```

Допустимыми значениями для Option Base являются только 0 и 1. Этот оператор служит для того, чтобы обеспечить совместимость Visual Basic с другими диалектами Basic, индексация в которых начинается с 1.

Для установки других границ массива необходимо использовать следующий синтаксис:

```
Dim <Имя переменной> ([<Нижний предел> To]
    <Верхний предел>)
```

Указанием верхней и нижней границ можно задать любые диапазоны индекса. Это удобно, если индекс несет определенную смысловую нагрузку (дата, номер заказа, возраст и т. п.):

```
Dim BirthDate (1980 To 2050)
```

Visual Basic позволяет также создавать многомерные массивы. При объявлении многомерного массива верхние границы каждой размерности разделяются запятыми:

```
Dim NameArray(10, 25) As String
```

Массив с именем NameArray может содержать 286 различных значений (11 × 26 = 286).

Динамический массив объявляется в том случае, если его размер заранее неизвестен. Объявление массива как динамического позволяет изменять его размер или размерность во время выполнения программы.

Динамический массив создается в два этапа. Сначала массив определяют без указания размера:

```
Dim DynArray() As Variant
```

Затем с помощью оператора ReDim устанавливают фактический размер массива:

```
ReDim DynArray (50, 10)
```

Синтаксис оператора ReDim:

```
ReDim Имя переменной (<Границы>) [As <Тип данных>]
```

Операции с массивами. Начиная с Visual Basic 6.0, в языке появилась возможность присваивать содержимое одного массива другому, например, для массивов newCopy и oldCopy:

```
newCopy = oldCopy
```

Тип данных, определяемый пользователем. Язык дает возможность определять типы данных, представляющие собой совокупность описания полей данных, аналогичную, например, записи языка Pascal.

Синтаксис определения пользовательского типа данных следующий:

```
Type <Имя типа> <Имя поля> As type
     [<Имя поля> As type]
....
End Type
```

После описания типа данных необходимо разместить переменную заданного типа с помощью оператора Dim:

```
Dim <Имя переменной> As <Имя типа>
```

Например:

```
Type StudentRecord           ' Определяем тип данных
     FirstName As String * 20
     LastName As String * 20
     Address As String * 30
     Phone As Long
     Birthday As Date
End Type
Dim MyRecord As StudentRecord  ' Объявляем переменную
                               ' Заполняем поля данных
MyRecord.FirstName = "Лютиков"
MyRecord.LastName = "Иван"
MyRecord.Address = "г. Москва, ул. Профсоюзная, д.5 кв. 10"
MyRecord.Phone = 1205643
MyRecord.Birthday = #12.09.86#
```

Идентификаторы типов данных

При объявлении переменных тип данных можно не указывать. Для того чтобы переменная была отнесена к определенному типу, можно использовать так называемые идентификаторы типов — специальные символы, добавляемые справа к идентификатору, задающему имя переменной (табл. 2.3).

Таблица 2.3. **Идентификаторы типов данных**

Тип данных	Знак	Пример
Integer	%	MyIntVar% = 5
Long	&	MyLongVar& = 317 689 654
Single	!	MySingleVar! = -3.40282
Double	#	MyDoubleVar# = 10^(-12)
Currency	@	MyCurrencyVar@ = 685 477.5807
String	$	MyStringVar$ = "My string"

2.3. Операции

Операции подразделяются на несколько групп:
- арифметические;
- операции над строками;
- операции с битами информации;
- сравнения;
- логические.

Приоритет выполнения операций

Если в выражении заданы операции более чем одной категории, то существует следующий порядок их выполнения: сначала выполняются арифметические операции, затем операции сравнения и последними выполняются логические операции.

Операции сравнения имеют один уровень приоритета и выполняются в порядке следования слева направо. Арифметические и логические операции выполняются в соответствии с порядком, приведенным в табл. 2.4.

Операция конкатенации строк (&) не относится к арифметическим операциям, но по приоритету выполнения находится как раз между арифметическими операциями и операциями сравнения (т. е. выполняется после арифметических операций, но перед операциями сравнения).

Таблица 2.4. **Приоритет выполнения операций**

Арифметические операции	Операции сравнения	Логические операции
Возведение в степень (^)	Равенство (=)	Not
Признак отрицательного числа (–)	Неравенство(<>)	And
Умножение и деление (*, /)	Меньше, чем (<)	Or
Деление нацело (\)	Больше, чем (>)	Xor
Остаток от деления (Mod)	Меньше либо равно (<=)	Eqv
Сложение и вычитание (+, –)	Больше либо равно (>=)	Imp
Конкатенация строк (&)	Like, Is	

Арифметические операции

Арифметические операции могут применяться только к операндам целых и вещественных типов (табл. 2.5).

В качестве операндов арифметических операций могут выступать стандартные арифметические (или тригонометрические) функции, т. е. функции с результатом целого или вещественного типа, аргументами которых являются выражения целого или вещественного типа. В табл. 2.6 приведены характеристики некоторых математических функций.

Таблица 2.5. **Арифметические и строковые операции**

Знак	Операция	Количество операндов	Тип операндов	Тип результата	Результат
^	Возведение в степень	2	Целый Хотя бы один вещественный	Целый Вещественный	x^y — возведение x в степень y. Значение x может быть отрицательным только при целом y
–	Признак отрицательного	1	Целый Вещественный	Целый Вещественный	Меняет знак операнда на противоположный
+	Сложение	2	Целый	Целый Вещественный	Сумма двух чисел
–	Вычитание	2	Целый Хотя бы один вещественный	Целый Вещественный	Разность двух чисел

Окончание табл. 2.5

Знак	Операция	Количество операндов	Тип операндов	Тип результата	Результат
*	Умножение	2	Целый Хотя бы один вещественный	Целый Вещественный	Произведение двух чисел
/	Деление	2	Целый или вещественный	Вещественный	Частное от деления двух чисел
\	Деление целых чисел	2	Целый	Целый	Целая часть от деления целых чисел: 25 \ 6 = 4
mod	Остаток от деления целых чисел	2	Целый или вещественный	Целый	Остаток от деления нацело: 25 mod 6 = 1 Если один из операндов — вещественный, перед выполнением операции происходит округление его до целого числа, например: 25 mod 6.7 = 4
+, &	Конкатенация	2	Строковый	Строковый	Строка, включающая содержание двух исходных

Таблица 2.6. **Некоторые математические и строковые функции ЯП Basic**

Функция	Назначение	Аргументы (параметры)	Результат
Abs(X)	Модуль (абсолютная величина) аргумента	Целый Вещественный	Целый Вещественный
Atn(X)	Арктангенс аргумента	Вещественный	Вещественный (вычисляется в радианах)
Cos(X)	Косинус аргумента	Вещественный	Вещественный Abs(Cos(X)) <=1
Exp(X)	Возведение числа *e* (основание натурального логарифма) в степень *X*	Вещественный, X <= 709.782712893; e считается равным 2.718282	Вещественный
Fix(X)	Целая часть числа. Для отрицательных чисел — минимальное целое число, большее или равное *X*	Целый или вещественный	Целый Например: Fix(-2.8) = -2 Fix(2.8) = 2

Продолжение табл. 2.6

Функция	Назначение	Аргументы (параметры)	Результат

`Int(X)`	Целая часть числа. Для отрицательных чисел — минимальное целое число, меньшее или равное *X*	Целый или вещественный	Целый
			Например:
			`Int(-2.8) = -3`
			`Int(2.8) = 2`

`Log(X)`	Натуральный логарифм аргумента	Целый или вещественный; *e* считается равным 2.718282	Вещественный
`Rnd`	Датчик случайных чисел		Вещественный, `0 ≤ Rnd < 1`
`Sgn(X)`	Индикатор знака числа	Целый или вещественный	–1, если `X < 0`; 0, если `X = 0`; 1, если `X > 0`
`Sin(X)`	Синус аргумента	Вещественный	Вещественный `Abs(Sin(X))<=1`
`Tan(X)`	Тангенс аргумента	Вещественный	Вещественный (в радианах)
`Sqr(X)`	Квадратный корень аргумента	Вещественный, `X >= 0`	Вещественный
`Pi`	Число π		Вещественный
`Left(S,m)` `Left$(S,m)`	Остаеляет в строке `m` символов, начиная с первого	`S` — строка, `m` — количество символов	Строка, содержащая первые `m` символов строки `S`
`Right(S,m)` `Right$(S,m)`	Оставляет в строке `m` последних символов	`S` — строка, `m` — количество символов	Строка, содержащая последние `m` символов строки `S`
`Mid(S,I[,m])` `Mid$(S,I[,m])`	Выделение подстроки	Строка `S`; целые значения `I` и `m`	Часть строки `S`, начиная с позиции `I`, длиной `m`. Если параметр `m` отсутствует или его значение больше длины строки, то выдается часть строки `S`, начиная с позиции `I` до конца строки
`StrComp` `(S1,S2[,cmp])`	Сравнение двух строк	Строка `S1`, строка `S2`, `cmp` – признак сравнения (0 — двоичное, 1 — символьное)	0, если `S1 = S2`; -1, если `s1 < s2`; 1, если `s1 > s2`; `null`, если `S1= null` или `S2 = null`

Окончание табл. 2.6

Функция	Назначение	Аргументы (параметры)	Результат
`Len(S)`	Вычисление длины строки	Строка `S`	Целое значение длины строки
`LCase(S)` `LCase$(S)`	Преобразование строки к нижнему регистру	Строка `S`	Новое значение строки `S` — все символы строчные
`InStr([I,]S1,` `S2[,cmp])`	Вычисляет позицию начала подстроки в строке	Подстрока `S2`, строка `S1`, целое значение `I` — позиция начала поиска, `cmp` — признак сравнения (0 —двоичное, 1 — символьное)	= 0, если `S2` не содержится в `S1`; целое > 0 — позиция `S2` в `S1`. Если параметр `I` отсутствует, поиск ведется от начала строки
`Trim(S)` `Trim$(S)`	Удаляет пробелы в начале и в конце строки	Строка `S`	Новая строка
`LTrim(S)` `Ltrim$(S)`	Удаляет пробелы в начале строки	Строка `S`	Новая строка
`RTrim(S)` `Rtrim$(S)`	Удаляет пробелы в конце строки	Строка `S`	Новая строка
`UCase(S)` `UCase$(S)`	Преобразование строки к верхнему регистру	Строка `S`	Новое значение строки `S` — все символы строчные
`Val(S)`	Преобразование строкового типа в числовой	Строка `S`	Целое или вещественное значение, соответствующее строковой записи `S`. Пробелы, знаки табуляции и конца строки пропускаются, преобразование ведется от начала строки до первого нецифрового символа

Результат тригонометрической функций `Atn(X)` представляет собой угол, измеряемый в радианах. Чтобы преобразовать радианы в градусы, необходимо умножить значение в радианах на константу `180/Pi` (180/π). Для обратного преобразования (градусов в радианы) необходимо умножить величину в градусах на константу `Pi/180`.

Значение функции Fix(X) эквивалентно выражению:

```
Sgn(X) * Int(Abs(X))
```

Чтобы вычислить выражение вида $\log_x y$, можно воспользоваться выражением для перехода к другому основанию логарифма:

```
Logxy = Log(Y) / Log(X)
```

Примеры записи арифметических выражений:

```
(Int(X)+sqr(Z))*2
Fix(Tan(X)-1)
Pi* R^2
(Cos(X)+Y)/Z
Log(X)/Log(2)+X/Y
P^Q)/R^(S+T)
```

Операции со строковыми типами

Visual Basic позволяет использовать знак операции «+» или «&» для объединения двух строковых операндов (см. табл. 2.5). Результатом операции S+T (или S&T), где S и T имеют строковый тип, будет конкатенация S и T — новая строка, результат добавления строки T в конец строки S.

Стандартные функции для работы со строками. Для работы с переменными строкового типа определены стандартные функции. Некоторые из них (наиболее часто используемые) приведены в табл. 2.6.

Рассмотрим примеры использования строковых функций.

1. Выделение первого слова в предложении (разделитель слов — знак «пробел»):

```
S_Sentence = Trim(S_Sentence) 'Удаление пробелов
                              'в начале строки
i = InStr(S_Sentence," ") 'Определение позиции первого пробела
                          ' в предложении S_Sentence
If i > 0 Then
      S_Word = Left(S_Sentence, i-1)
Else
      S_Word = S_Sentence
End If                    'В строковой переменной S_Word
                          ' - значение первого слова.
```


2. Удаление из строки всех цифр:

```
L = Len(S_Sentence)      'Вычисление длины строки
SDigit = "0123456789"    'Инициализация строковой переменной
                         'Sdigid, содержащей перечень всех цифр
I = 1
While I <= L
 Ch = Mid(S_Sentence,I,1) 'Выделение очередного
                          'символа строки S_Sentence
 J = InStr(SDigit, Ch)    'Проверка, является ли символ цифрой
 If J>0 Then
    S_Sentence = Left(S_Sentence, I - 1) +
    Right(S_Sentence, L - I)
                           'Удаление символа из строки
    L = L-1                'Уменьшение длины строки на 1,
                           'значение I не меняется
 Else
       I = I+1
 End If
Wend
```

3. Подсчет количества букв 'W' в строке (независимо от регистра):

```
N_w = 0;                          'Обнуление счетчика букв
S_Sentence = LCase(S_Sentence) 'Преобразование строки
                               'к нижнему регистру
L = Len(S_Sentence);              'Подсчет длины строки
For I = 1 To L
     If InStr(S_Sentence, 'w')>0
     Then N_w = N_w+1             'Увеличение счетчика букв на 1.
```

Операции сравнения

В результате выполнения операций сравнения получается значение логического типа: true или false. В табл. 2.7 приводятся значения результатов для каждой операции, выполняемой над <Операндом-1> (O1) и <Операндом-2> (O2), в зависимости от значений операндов. Если хотя бы один из операндов при выполнении любой операции принимает значение null, то результатом выполнения операции тоже будет null.

При сравнении операндов следует учитывать, что правила сравнения чисел или символьных строк всегда корректно действуют только в том случае, если сравниваемые операнды однотипны, например, оба операнда — числовые значения или же символьные строки.

Таблица 2.7. **Операнды и результаты операций сравнения**

Операция	Описание	True (истина), если	False (ложь), если
<	Меньше	O1 < O2, например 2 < 5 — истина	O1 >= O2, например 5 < 5 — ложь
<=	Меньше либо равно	O1 <= O2, например 2 <= 5 — истина	O1 > O2, например 5 <= 2 — ложь
>	Больше	O1 > O2, например 5 > 2 — истина	O1 <= O2, например 5 > 5 — ложь
>=	Больше либо равно	O1 >= O2, например 5 >= 2 — истина	O1 < O2, например 2 >= 5 — ложь
=	Равно	O1 = O2, например 5 = 5 — истина	O1 <> O2, например 2 = 5 — ложь
<>	Не равно	O1 <> O2, например 2 <> 5 — истина	O1 = O2, например 5 <> 5 — ложь

К операциям сравнения относят также и операции **Is** и **Like**, которые имеют специфические области применения.

Операция **Is** применяется для сравнения двух переменных A и B типа Object. Если обе переменные ссылаются на один физический объект в памяти, то в результате операции A **Is** B получается значение true. В противном случае результат равен false.

Операция **Like** служит для выявления соответствия между значением переменной строкового типа и так называемым образцом. Под образцом здесь понимается строка, содержащая специальные символы, трактующиеся в соответствии с правилами, приведенными в табл. 2.8.

Таблица 2.8. **Правила сравнения с образцом**

Символ в образце	Соответствие в строке
?	Любой символ, например "aBa" Like "a?a" возвращает true
*	Любое количество символов (или ни одного), например "BAT123456" Like "B?T*" возвращает true
#	Любая цифра (0—9), например "год 2002" Like "год ####" возвращает true
[список_символов]	Любой символ из списка, например "X" Like "[A-Z]" возвращает true; "X" Like "[A-WY-Z]" возвращает false
[!список_символов]	Любой символ, не находящийся в списке, например "9" Like "[!A-Z]"возвращает true; "X" Like "[!A-WY-Z]" возвращает false

Логические операции

Логические операции применяются к операндам логического типа, т. е. в качестве операндов выступают выражения, при вычислении дающие результат логического типа. Результат выполнения логических операций тоже логического типа. Вычисление логических выражений происходит в соответствии с таблицами истинности логических операций (см. табл. 1.7—1.9).

Примеры записи логических выражений:

```
(X>=0) and (X<=1)
((X>0) and (X<0.5)) or (X>3)
(N mod 2 = 0) imp (N>0)
```

При использовании логических выражений в качестве условий (например, в условных операторах и операторах цикла) значение логического выражения null приравнивается к значению false (см. также табл. 1.13, 1.14).

Операции с битами информации (побитовые). Операции побитового сравнения выполняются для операндов числового типа и в результате дают значение, получающееся путем побитового выполнения операции над операндами в соответствии с правилами, приведенными в табл. 1.15.

Рассмотрим примеры выполнения операций побитового сравнения. Пусть объявлены две переменные A и B типа Integer. В переменной A находится число 12, а в переменной B — число 9. Запишем значения переменных в двоичном побитовом представлении и применим операции побитового сравнения:

A = 12	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
B = 9	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
Not B = not 9 = -10	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0
A and B = 12 and 9 = 8	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
A or B = 12 or 9 = 13	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
A eqv B = 12 eqv 9 = -6	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
A imp B = 12 imp 9 = -5	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
A xor B = 12 xor 9 = 5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

2.4. Операторы языка

Оператор присваивания

Данный оператор предназначен для присваивания переменной значения.

Синтаксис:

```
[Let] <Идентификатор> = <Выражение>
```

Тип данных переменной, имя которой стоит в левой части оператора присваивания, должен быть совместим с типом данных выражения. Если для переменной не задан тип данных, то оператор присваивания одновременно с занесением значения в переменную определяет и ее тип данных для дальнейшего использования. Тип данных в этом случае соответствует типу данных выражения в правой части оператора.

Например:

Let `L = 2*Pi*R` — присвоить переменной `L` значение длины окружности радиуса `R`.

Аналогично:

```
L = 2*Pi*R
```

Let `I = I + 5` — увеличить на 5 числовое значение, находящееся в переменной `I`.

Аналогично:

```
I = I + 5
```

Оператор безусловного перехода

Оператор **GoTo** позволяет изменить стандартный последовательный порядок выполнения операторов и передать управление заданному оператору, которому в этом случае должна предшествовать метка. Эта же метка должна быть указана при операторе **goto**.

Синтаксис оператора безусловного перехода:

```
GoTo <Метка>
```

<Метка> представляет собой идентификатор или целое число (0—9999) — номер строки программного кода. Метка всегда должна стоять в начале строки и отделяться от первого оператора строки двоеточием. Значение метки должно быть уникально внутри модуля.

Пример:

```
10:  MyString = "Выполняется оператор строки номер 10"
     GoTo 30     'Передача управления на строку номер 30.
20:  MyString = "Выполняется оператор строки номер 20"
                 'Оператор строки номер 20 не выполняется
30:  MyString = "Выполняется оператор строки номер 30"
```

Условный оператор

В языке Visual Basic условный оператор реализован в двух формах:

- **If...Then...Else**;
- **Select Case**.

Оператор If...Then...Else имеет следующий синтаксис:

```
If <Логическое_выражение> Then [<Оператор-1>]
       [Else <Оператор-2>]
```

Такая синтаксическая конструкция должна быть обязательно записана в одной строке. Если в качестве <Оператора> выступает последовательность операторов, то операторы при записи должны отделяться друг от друга двоеточием («:»).

Рассмотрим пример.

Написать последовательность действий для решения следующей задачи.

Присвоить переменной y значение $\sin x$, если $x > 0$, и значение $\cos x$, если $x \le 0$, а также присвоить переменной z значение «1», если $x > 0$ и «0», если $x \le 0$:

```
If x > 0 Then y = sin(x): z =1 Else y = cos(x): z = 0.
```

Оператор **If...Then...Else** имеет и другую (так называемую блочную) форму записи:

```
If <Логическое_выражение> Then
     [<Оператор-0>]
[ElseIf <Логическое выражение-1> Then
     [<Оператор-1>]
[ElseIf <Логическое_выражение-2> Then
     [<Оператор-2>]
...
[ElseIf <Логическое_выражение-N> Then
     [<Оператор-N>]
[Else[<Оператор-N+1>]]
End If
```

Обязательными частями записи условного оператора в блочной форме являются первая и последняя строки. Количество условных ветвей, начинающихся со служебного слова **ElseIf**, теоретически не ограничено.

Такую форму удобно использовать в том случае, если реализация алгоритма требует программирования вложенных условных операторов. Изменим, например, условие представленной выше задачи следующим образом: присвоить переменной y значение $\sin x$, если $x < 1$, и значение $\cos x$, если $x \geq 2$, т. е. значение переменной x рассматривается на трех интервалах: $(-\infty, 1)$, $[1, 2)$ и $[2, +\infty)$.

Алгоритм решения задачи может быть описан с помощью блочной формы условного оператора:

```
If x < 1 Then
     y = sin(x)
ElseIf x >= 2 then
     y = cos(x)
End If
```

Далее рассмотрим запись алгоритма решения задачи, в которой значение y зависит от x следующим образом:

$$y = \begin{cases} \sin x, & x < -1; \\ 1 - \sin x, & -1 \leq x < 0; \\ 1 - \cos x, & 0 \leq x \leq 1; \\ \cos x, & x > 1. \end{cases}$$

Условный оператор для реализации алгоритма может быть организован так:

```
If x < -1 Then
     y = sin(x)
     ElseIf (x >= -1) and (x < 0) then
          y = 1 - sin(x)
     ElseIf (x >= 0) and (x <= 1) then
          y = 1 - cos(x)
     Else y = cos(x)
End If
```

Оператор Select Case аналогичен по действию оператору **Case** языка Pascal и похож на **do case** в FoxPro. Синтаксис оператора **Select Case** следующий:

```
Select Case <Выражение>
     [Case <Константа-1> [<Оператор-1>]
     ...
     [Case <Константа-N> [<Оператор-N>]
     [Case Else [<Оператор>]
End Select
```

<Выражение> в заголовочной части оператора представляет собой арифметическое, логическое или строковое выражение, результат вычисления которого последовательно сравнивается с константными значениями (числовыми или строковыми), заданными в **Case**-ветвях оператора. Если значение выражения совпало с одним из константных значений, то выполняется оператор (или группа операторов), записанный после заголовка соответствующей **Case**-ветви. Если же константное значение, равное значению выражения, не найдено, то выполняется оператор (или группа операторов), записанный после **Case Else**.

В случае использования в заголовочной части логического выражения **Case**-ветвей может быть три — с константными значениями `true`, `false` и `null`.

Между заголовками **Case**-ветвей может стоять любое количество строк операторов. Операторы, записанные в одной строке, должны отделяться друг от друга двоеточием.

При использовании оператора **Select Case** необходимо помнить о том, что значение выражения и константы должны быть одного типа.

Рассмотрим запись оператора **Select Case** для решения задачи — присвоить строке S значение дня недели для заданного числа D при условии, что в месяце 31 день и 1-е число — понедельник.

```
Select Case D mod 7
     Case 1
          S = "понедельник"
     Case 2
          S = "вторник"
     Case 3
          S = "среда"
     Case 4
          S = "четверг"
     Case 5
          S = "пятница"
     Case 6
          S = "суббота"
     Case 0
          S = "воскресенье"
End Select
```

С введением ограничений на область возможных значений переменной D и использованием полной формы оператора **Select Case** получим следующую запись алгоритма решения задачи:

```
If (D>=1) and (D<=31) then
     Select Case D mod 7
          Case 1
               S = "понедельник"
          Case 2
               S = "вторник"
          Case 3
               S = "среда"
          Case 4
               S = "четверг"
          Case 5
               S = "пятница"
          Case 6
               S = "суббота"
          Case Else
               S = "воскресенье"
     End Select
End If
```


Операторы цикла

В языке Visual Basic реализовано три разновидности оператора цикла:

- **For...Next**;
- **Do...Loop**;
- **While...Wend**.

Оператор For...Next. Синтаксис:

```
For <Переменная цикла> = <Начало>
To <Конец> [Step <шаг>][<Тело цикла>]
Next [<Переменная цикла>]
```

<Переменная цикла>, <Начало>, <Конец> и <Шаг> должны принадлежать числовому множеству значений. Использование конструкции **Step** <Шаг> позволяет управлять изменением значений переменной цикла. Значение шага изменения может быть как положительным, так и отрицательным. В случае отсутствия конструкции в заголовке цикла по умолчанию принимается значение шага, равное 1. В табл. 2.9 рассмотрено действие конструкции **Step** <Шаг> на примере простого цикла, увеличивающего значение переменной J (считаем, что начальное значение J равно 0).

Таблица 2.9. **Примеры циклов**

Оператор For	Значения переменной цикла	Значение J после выхода из цикла
For I = 1 To 10 Step 2 J = J+1 Next I	I = 1, 3, 5, 7, 9	J = 5
For I = 10 To 1 Step -2 J = J+1 Next I	I = 10, 8, 6, 4, 2	J = 5
For I = 1 To 10 J = J+1 Next I	I = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10	J = 10

Для значений <Начало> и <Конец> должно выполняться следующее правило:

- при отрицательном значении шага <Начало> ≥ <Конец>;
- при положительном значении шага <Начало> ≤ <Конец>.

Если указанные правила в заголовке цикла не выполняются, то не выполняются и операторы тела цикла.

Использование в теле цикла оператора **Exit For** позволяет программировать альтернативный путь для выхода из цикла. При его выполнении управление немедленно передается первому оператору, следующему за предложением **Next**, т. е. происходит выход из цикла. Например, в приведенном ниже операторе выход из цикла произойдет, как только очередное вычисленное значение переменной J станет больше 5 (т. е. при J = 7):

```
For I = 1 To 10
      J = J*2+1
      If J > 5 then Exit For
Next I
```

Оператор **For...Next** можно использовать для программирования вложенных циклов, например, используя следующую конструкцию:

```
For I = 1 To 10
     For J = 1 To 10
          For K = 1 To 10
          ...
          Next K
     Next J
Next I
```

Любая другая последовательность применения операторов **Next** будет некорректной, например, вложенный оператор, приведенный ниже, вызовет ошибку:

```
For I = 1 To 10
      For J = 1 To 10
           For K = 1 To 10
           ...
           Next J
      Next K
Next I
```

Если при этом в предложении **Next** не указывается переменная цикла, то очередное предложение **Next** считается компилятором относящимся к первому (по направлению вверх) предложению **For ... To**.

Рассмотрим примеры применения оператора **For ... To**.

1. Вычислить значение функции $y = N!$ ($y = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$) для $N = 100$:

```
y = 1;
     For i = 2 To 100
     y = y*i;
Next i.
```

2. Найти максимальный делитель D натурального числа k (за исключением самого k):

```
For i = k \ 2 To 1 Step -1
      if k mod i = 0 then
      D := I
      Exit For  ' Выход из цикла, если делитель найден
      End If
Next i
```

Оператор Do...Loop предназначен для повторения выполнения тела цикла до тех пор, пока значение некоторого логического выражения истинно или пока это значение не станет истинным.

Оператор может быть записан в двух формах: в форме проверки условия до выполнения тела цикла и в форме проверки условия после выполнения тела цикла

Синтаксис оператора с проверкой условия до выполнения тела цикла:

```
Do [{While | Until} <Условие>][<Тело цикла>]Loop
```

Синтаксис оператора с проверкой условия после выполнения тела цикла :

```
Do[<Тело цикла>] Loop [{While | Until} <Условие>]
```

Алгоритм выполнения оператора совпадает с алгоритмом выполнения операторов **While** и **Repeat** языка Pascal: максимальное число шагов цикла заранее не известно, а изменение значения переменных, входящих в логическое выражение, должно программироваться внутри тела цикла.

Необязательное <Условие> выполнения цикла может быть задано со словом **While**, и тогда <Тело цикла> выполняется до тех пор, пока значение условия — истина, или со словом **Until**, и тогда <Тело цикла> выполняется до тех пор, пока значение условия — ложь. Необязательность задания условия выполнения

цикла объясняется тем, что <Тело цикла> (как и <Тело цикла> оператора **For...Next**) может содержать оператор **Exit Do**, позволяющий завершить цикл принудительно.

Рассмотрим опять в качестве примера задачу нахождения максимального делителя (пример 2 предыдущего пункта). В табл. 2.10 приводятся различные варианты алгоритма решения задачи с использованием оператора **Do...Loop**.

Таблица 2.10. **Алгоритмы решения задачи о максимальном делителе**

С проверкой условия до выполнения тела цикла		С проверкой условия после выполнения тела цикла		С использованием оператора Exit Do
Условие While	Условие Until	Условие While	Условие Until	
D = 0 i := k \ 2 **Do While** D = 0 **If** k **mod** i = 0 **Then** D = i **End If** i = i - 1 **Loop**	D = 0 i := k \ 2 **Do Until** D<>0 **If** k **mod** i = 0 **Then** D = i **End If** i = i - 1 **Loop**	D = 0 i := k \ 2 **Do** If k mod i = 0 **Then** D = i **End If** i = i - 1 **Loop While** D=0	D = 0 i := k \ 2 **Do** **If** k **mod** i = 0 **Then** D = i **End If** i = i - 1 **Loop Until** D<>0	D = 0 i := k \ 2 **Do** **If** k **mod** i=0 **Then** D = i :**Exit Do** **End If** i = i - 1 **Loop**

Оператор While...Wend позволяет организовать цикл типа Пока.

Синтаксис оператора следующий:

```
While <Условие> [<Тело цикла>]
Wend
```

Рассмотрим пример. Член ряда с номером n для $n = 1, 2, 3, \ldots$ определяется выражением $\frac{1}{n^x}$. Написать последовательность операторов для вычисления суммы членов ряда от первого до (включительно) члена с наименьшим номером, не превосходящего 10^{-6}.

```
S = 0
N = 1
SN = 1                'Задание начального значения
                      'для члена ряда (больше, чем 10^-6)
While SN >= 10^(-6)
      SN = 1/ N^X
      S = S + SN
      N = N + 1
Wend
```


2.5. Процедуры и функции

В языке определены две разновидности подпрограмм — процедуры и функции. Каждое объявление процедуры или функции содержит обязательный заголовок, за которым следует последовательность операторов и обязательный признак завершения подпрограммы.

Передача управления в подпрограмму происходит с помощью оператора процедуры или функции.

Процедуру (функцию) в ЯП Basic нельзя объявить внутри другой процедуры (функции).

Объявление процедуры

Обобщенный синтаксис объявления процедур в Basic следующий:

```
[Private | Public] [Static]
Sub <Имя процедуры> [(<Список параметров>)]
    [<Операторы>]
    [Exit Sub] [<Операторы>]
End Sub
```

Необязательные операторы **Private** и **Public** имеют смысл для версий расширения Visual Basic:

- оператор **Public** делает процедуру доступной для использования не только модулем, в котором она объявлена, но и всеми другим модулями;
- оператор **Private** служит для объявления процедуры, доступной для использования только текущему модулю (и его процедурам).

Необязательный оператор **Static** позволяет сохранять значения локальных переменных процедуры между ее вызовами. Если предложение **Static** не используется, значения локальных переменных не сохраняются.

Рассмотрим простой пример объявления процедуры:

```
Sub ExampleProc
    VariableA = VariableA + 1
    Print VariableA
End Sub
```

Каждый раз при вызове процедуры с помощью оператора ExampleProc будет выводиться на экран число 1. Если же модифицировать объявление процедуры следующим образом:

```
Static Sub ExampleProc
    VariableA = VariableA + 1
    Print VariableA
End Sub,
```

то всякий раз при вызове процедуры на экран будет выводиться значение, увеличенное на единицу (1, 2, 3 и т. д.).

Оператор **Exit Sub** в теле процедуры может встречаться произвольное число раз и всякий раз вызывает немедленное завершение процедуры.

Оператор вызова процедуры имеет следующий синтаксис:

```
<Имя процедуры> [<Список параметров>]
```

Необязательный список фактических параметров (через запятую) приводится после имени процедуры без круглых скобок (в качестве разделителя имени и списка параметров выступает знак пробела):

```
Proc Par1, Par2, Par3
```

Объявление функций

Синтаксис объявления функции следующий:

```
[Public|Private][Static]
Function <Имя функции> [(<Список параметров>)]
     [As <Тип>]
     [<Операторы>][<Имя функции> = <Выражение>]
     [Exit Function]
     [<Операторы>][<Имя функции> = <Выражение>]
End Function
```

В отличие от синтаксиса объявления процедуры объявление функции может содержать в заголовке явное указание типа возвращаемого значения (предложение **As** <Тип>).

Использование операторов Public, Private и Static аналогично использованию этих операторов при объявлении процедуры.

Действие оператора Exit Function аналогично действию оператора Exit Sub.

Возвращаемое функцией значение должно быть представлено выражением в правой части оператора присваивания (<Имя функции> = <Выражение>).

Приведем пример объявления функции вычисления квадратного корня действительного числа.

```
Function CalculateSquareRoot (Arg As Double) As Double
    If Arg < 0 Then       ' Проверка параметра
        Exit Function     'Завершение функции,
                          'если параметр отрицательный
    Else
        CalculateSquareRoot = Sqr(Arg)  'Вычисление
                                        'возвращаемого значения
    End If
End Function
```

Оператор вызова функции отличается от оператора вызова процедуры тем, что список фактических параметров при вызове функции заключается в круглые скобки:

```
C = CalculateSquareRoot (Arg)
```

Объявление параметров. Процедуры и функции могут использовать параметры, список которых (при необходимости с указанием типа) размещают в скобках после имени подпрограммы, например:

```
Sub Factorial (N As Integer)
    ...
End Sub
```

При необходимости в объявлении указывается тип данных для параметров. В приведенном выше примере параметр N имеет тип Integer.

В версиях языка Visual Basic появилась возможность использовать обе категории вызова параметров: по значению и по наименованию. Обобщенный синтаксис объявления параметра в списке параметров следующий:

```
[Optional] [ByVal | ByRef] <Имя параметра>
[As <Тип параметра>]
[= <Значение по умолчанию>]
```

Передача параметров по значению. Для передачи параметров-значений перед именем параметра в заголовке процедуры следует указывать ключевое слово ByVal. В этом случае процедуре передается копия этого значения. При передаче параметров-значений ключевое слово ByVal должно указываться обязательно:

```
Function Factorial (ByVal N As Integer) As Integer
     F = 1
     For i = 2 to N
          F = F * i
     Next i
     Factorial = F
End Function
```

Передача параметров по наименованию. Для того чтобы передать параметр по наименованию, следует перед параметром в объявлении указать ключевое слово ByRef. По умолчанию (т. е. без указания способа передачи) параметры в Visual Basic передаются по наименованию.

Если параметр передается по наименованию, вызванная процедура получает физический адрес памяти передаваемой переменной и может изменять значение этого параметра.

Запишем алгоритм вычисления факториала в виде процедуры:

```
Sub Factorial (N As Integer, ByRef F As Integer)
F = 1
      For i = 2 to N
           F = F * i
      Next i
End Sub
```

Необязательные параметры. Если при вызове процедуры в качестве фактических параметров указать не все формальные параметры, то последует сообщение об ошибке. Однако в Visual Basic существует возможность в заголовке подпрограммы объявить так называемые необязательные параметры (т. е. параметры, которые при вызове подпрограммы можно не указывать в списке фактических параметров).

Для того чтобы параметр стал необязательным, перед его именем ставится ключевое слово Optional. Для параметров типа Optional (и только для них) можно указать значение по умолчанию (например, Optional Param = 5). После первого необязательного параметра все последующие должны быть также необязательными, например:


```
Sub SomeProc (Par1, Optional Par2 = "Two",
     Optional Par3 = "Three")
          Print Par1, Par2, Par3  'вывод значений параметров
                                  'на экран
End Sub
```

Рассмотрим теперь следующие операторы вызова процедуры:

```
SomeProc "One"
SomeProc "One","And"
SomeProc "A","B","C"
```

В результате выполнения первого оператора на экране появится значение "One Two Three", в результате второго — "One And Three" и в результате третьего — "A B C".

2.6. Организация ввода-вывода. Работа с файлами

Для работы с файлами в Basic определено понятие *канала ввода-вывода*. При открытии файлу ставится в соответствие канал с определенным номером. Таким образом, каждый открытый файл имеет собственный канал, с помощью которого записываются или считываются данные. В операциях ввода и вывода данных для задания связи с физическим файлом используется номер канала.

В Basic реализованы три типа доступа к файлам:

- последовательный (Sequential) — для чтения и записи текстовых файлов;
- произвольный (Random) — для чтения и записи структурированных файлов с записями фиксированной длины;
- двоичный (Binary) — для чтения и записи неструктурированных файлов.

При создании коммуникационных каналов система должна знать, какой тип доступа к каждому конкретному файлу нужно использовать и какова структура данных этого файла.

Общий алгоритм работы с внешними файлами состоит из нескольких этапов:

- определение номера канала ввода-вывода;

- открытие файла;
- чтение или запись данных;
- закрытие файла.

Определение номера канала ввода-вывода

Номер свободного канала, который можно использовать для работы с файлом, вычисляется с помощью функции FreeFile:

```
FreeFile [(<Область изменения>)]
```

Необязательный параметр <Область изменения> позволяет определить диапазон значений, из которого выбирается очередной свободный номер канала. Если его значение равно 0 (по умолчанию), то возвращается номер канала из диапазона 1—255, если 1, то из диапазона 256—511, например:

```
FInput = FreeFile(1)      'В переменную FInput занесется
                          'номер свободного канала
                          'из диапазона 256–511
```

Открытие файла

Связь внешнего файла с выбранным каналом осуществляется с помощью оператора открытия файла, общий синтаксис которого (для всех типов доступа) следующий:

```
Open <Имя файла> For <Тип> [Access <Доступ>]
[<Блокировка>] As [#]<Номер канала>
[Len=<Длина записи>]
```

Здесь:

<Имя файла> — полный идентификатор внешнего файла (например, c:\example\date.dat);

<Тип> — ключевое слово, специфицирующее тип доступа к файлу (Append, Input, Output — для последовательного доступа, Binary — для двоичного и Random — для произвольного доступа);

<Доступ> — необязательное указание уровня доступа к файлу (Read — только чтение, Write — только запись, Read Write — чтение-запись; по умолчанию устанавливается уровень Read Write);

<Блокировка> — необязательный параметр, определяющий возможность совместного использования файла несколькими процессами (Shared — разделенный доступ; Lock Read, Lock Write, Lock Read Write — локальный доступ);

<Номер канала> — номер выделенного канала ввода-вывода, знак "#" ставится при числовом константном значении (#1);

<Длина записи> — необязательное числовое значение (меньшее либо равное 32 767 байтам), задающее длину записи для файлов произвольного (random) доступа или размер текстового буфера для текстовых файлов последовательного доступа (для файлов двоичного доступа параметр игнорируется).

В случае работы с файлом последовательного доступа ключевое слово типа доступа одновременно задает и уровень доступа к данным:

Input — только чтение из файла;

Output — запись в файл;

Append — добавление к файлу.

При попытке открытия для чтения (Input) несуществующего файла выдается сообщение об ошибке. При открытии несуществующего файла для записи или добавления (Output или Append) создается новый файл. Если файл с указанным именем существует, то в режиме Output его содержимое удаляется, а в режиме Append файл открывается для добавления, например:

```
Open "c:\example\users.txt" For Append As #3
```

С помощью такого оператора канал ввода-вывода под номером 3 связывается с размещенным на диске файлом c:\example\users.txt для добавления информации.

Чтение (запись) данных

Процессы чтения-записи при работе с файлами различного типа доступа управляются разными процедурами.

Считывание данных из файла, открытого для последовательного или двоичного доступа, осуществляется с помощью оператора **Input**:

```
Input #<Номер канала>, <Список переменных>
```

<Список переменных> содержит разделенные запятой имена переменных, в которые заносятся значения, последовательно считываемые из файла. В списке запрещено использовать переменные типа массив или объект.

Элементы данных в файле должны быть записаны в том же порядке, в котором запрашивается их чтение. Двойные кавычки (" ") внутри вводимых данных игнорируются.

Считывание данных из текстового файла, открытого для последовательного доступа, может быть проведено также с помощью оператора **Line Input**:

```
Line Input #<Номер канала>, <Строковая переменная>
```

В <Строковую переменную> заносится последовательность символов текстового файла до признака конца строки (код возврата каретки — Chr(13) или возврат каретки — перевод строки — Chr(13) + Chr(10)). Признаки конца строки пропускаются и следующее считывание из файла осуществляется с начала следующей строки.

Запись данных в файл последовательного доступа должна происходить с использованием операторов **Print** и **Write**.

Синтаксис оператора **Print**:

```
Print #<Номер канала>, [<Список вывода>],
```

где <Список вывода> — необязательный параметр, который задает выражение (или список выражений) для вывода в файл последовательного доступа. Если параметр отсутствует, в файл выводится признак перевода строки.

В список вывода, помимо выражений, могут входить следующие параметры:

Spc(n) — задает вывод последовательности символов «пробел» длиной *n*;

Tab(n) — задает позицию вывода следующих данных (n — позиция от начала строки; при использовании Tab без параметра очередной вывод начинается с начала следующей зоны).

Для форматирования записываемой в файл информации следует по-разному разделять выражения в операторе **Print**. Если выводимые данные в операторе разделять запятыми, то в файле они будут разделены символами табуляции. Если же в операторе

для разделения данных использовать точку с запятой, то данные в файл записываются без разделителей.

Данные, записанные в файл с помощью оператора **Print**, должны быть прочитаны из файла с помощью операторов **Input** или **Line Input**.

Синтаксис оператора **Write**:

```
Write #<Номер канала>, [<Список вывода>]
```

<Список вывода> — необязательный параметр, который задает выражение (или список выражений) для вывода в файл последовательного доступа. Выражения в списке могут разделяться запятой, пробелом или двоеточием. Если параметр отсутствует, в файл выводится признак перевода строки.

Данные, записанные в файл с помощью оператора **Write**, должны быть прочитаны из файла с помощью оператора **Input**.

Для считывания данных из файла произвольного или двоичного доступа используется оператор **Get**:

```
Get [#]<Номер канала>, [<Количество записей>],
      <Имя переменной>
```

В соответствии с синтаксисом оператора из файла считывается указанное количество записей (при произвольном доступе) или байтов (при двоичном доступе) и помещается в переменную. Нумерация записей (байтов) внутри файла начинается с *единицы*. Если число записей (байтов) не указано, считывается одна запись (или один байт). Разделитель параметров при этом в операторе должен присутствовать:

```
Get #2 , , RecordBuffer
```

Запись данных в файл *произвольного* или *двоичного* доступа осуществляется с помощью оператора **Put**, синтаксис которого сходен с синтаксисом оператора **Get**:

```
Put [#]<Номер канала>, [<Количество записей>],
      <Имя переменной>
```

Чтение данных из файла с помощью оператора **Get** предполагает, что запись данных в файл проводилась с использованием оператора **Put**, и наоборот: данные, записанные в файл посредством оператора **Put**, должны быть прочитаны оператором **Get**.

Закрытие файла

С закрытием файла связано освобождение канала ввода-вывода и, возможно, дальнейшее использование этого канала для связи с другим внешним файлом.

Для закрытия файлов служат операторы **Close** и **Reset**:

```
Close [<Список номеров каналов>]
Reset
```

Оператор **Close** освобождает все перечисленные в списке каналы ввода-вывода. Если закрывается файл последовательного доступа, предварительно открытый на запись или добавление (Output или Append), то происходит запись информации и освобождение выделенного программного буфера. Использование оператора **Close** без параметров приводит к закрытию всех ранее открытых файлов.

Действие оператора **Reset** аналогично вызову оператора **Close** без указания параметров.

При работе с клавиатурой как с устройством ввода данных и с экраном дисплея как с устройством вывода используются операторы **Input** и **Print** без указания первого параметра (номера канала), вследствие этого не требуется использование операторов **Open** и **Close**, например:

```
Print "Hello!"
```

На экран будет выведена текстовая строка "Hello!".

Рассмотрим пример работы с файлом прямого доступа. Пусть необходимо формировать и обрабатывать файл, состоящий из записей следующей структуры:

```
Type StudentRecord
     Family As String * 20
     Mark As Single
End Type
```

Составим программу, позволяющую записывать данные в файл и затем читать содержимое произвольной записи по ее номеру:

```
Type StudentRecord
     Family As String * 20
     Mark As Single
End Type  'Объявляем буферную переменную для записи в файл
Dim MyRecord As StudentRecord
                  'Открываем (создаем) файл прямого доступа
Open "STUDENTS.DAT" For Random As #1 LEN = LEN(MyRecord)
b = "Y"           'Флаг завершения ввода в файл
While b = "Y"
      Input "Введите фамилию : ", MyRecord.Family
      Input "Введите оценку : ", MyRecord.Mark
      Put #1, 1, MyRecord          'Ввод очередной записи
      Input "Продолжить ввод (Y/N)? ", b
      b = UCASE(b)
Wend
Close #1                                  'Закрыть файл
Open "STUDENTS.DAT" For Random As #1 LEN = LEN(MyRecord)
Input "Введите номер записи : ", I%       'Ввод номера записи
                                          'для чтения
Seek #1, I%-1                    'Установка файлового указателя
                                 'перед нужной записью
Get #1, 1, MyRecord              'Чтение I-й записи
Print "STUDENT:", MyRecord.Family
Print "SCORE:", MyRecord.Mark
Close #1
```


Примеры

Несколько программ на ЯП TurboBasic

А. Упражнения на обработку массивов.

А.1. Расположить элементы массива целых чисел по убыванию количества делителей. Массив сгенерировать с помощью датчика случайных чисел.

```
05 REM РАСПОЛОЖИТЬ ЭЛЕМЕНТЫ МАССИВА ПО УБЫВАНИЮ КОЛИЧЕСТВА_
ДЕЛИТЕЛЕЙ
15 REM ЧИСЛА ОТ N ДО 3*N
20 DIM A%(300), B%(300) 'ОПИСАНИЕ ИСХОДНОГО МАССИВА A% И_
ДЕЛИТЕЛЕЙ B%
30 INPUT "ВВЕДИТЕ КОЛИЧЕСТВО ЭЛЕМЕНТОВ МАССИВА A>", N%
40 REM ГЕНЕРАЦИЯ МАССИВА :
45 RANDOMIZE(1)
50 FOR I%=1 TO N%
60 X=RND
70 A%(I%)=N%+N%*2*X
80 NEXT I%
90 REM FOR I%=1 TO N%
```

```
100 REM PRINT A%(I%)
110 REM NEXT I%
115 PRINT "BEFORE SORT"
120 FOR I%=1 TO N%
130 NDEL%=0 : A1%=A%(I%)
140 FOR J%=2 TO INT(A1%/2)+1
150 WK1%=INT(A1%/J%) : WK2%=WK1%*J%
160 IF WK2%=A1% THEN NDEL%=NDEL%+1
170 NEXT J%
175 B%(I%)=NDEL%
180 PRINT A1%, NDEL%
190 NEXT I%
200 REM СОРТИРОВКА МАССИВА B%
210 FOR I%=1 TO N%
220 MIN%=B%(I%): D%=I%
230 REM НАХОЖДЕНИЕ МИНИМАЛЬНОГО ЭЛЕМЕНТА МАССИВА ИЗ_
ОСТАВШИХСЯ ЭЛЕМЕНТОВ
240 FOR J%=I%+1 TO N%
250 IF B%(J%)<MIN% THEN MIN%=B%(J%) : D%=J%
260 NEXT J%
270 W1%=A%(I%) : A%(I%)=A%(D%) : A%(D%)=W1%
280 W1%=B%(I%) : B%(I%)=B%(D%) : B%(D%)=W1%
285 NEXT I%
290 REM ВЫВОД УПОРЯДОЧЕННЫХ МАССИВОВ
295 PRINT "AFTER SORT:"
300 FOR I%=1 TO N%
310 PRINT A%(I%), B%(I%)
320 NEXT I%
330 END
```

А.2. В массиве целых чисел удалить те элементы, сумма цифр которых не составляет число, кратное 4. Если в массиве все числа только такие, выдать сообщение. Массив сгенерировать с помощью датчика случайных чисел.

```
05 REM ОСТАВИТЬ ТОЛЬКО ТЕ ЧИСЛА, СУММА ЦИФР КОТОРЫХ ЕСТЬ ЧИСЛО,
06 REM КРАТНОЕ 4, ИНАЧЕ - ВЫДАТЬ СООБЩЕНИЕ
15 REM ЧИСЛА ОТ N ДО 2*N
20 DIM A%(300) 'ОПИСАНИЕ ИСХОДНОГО МАССИВА A%
30 INPUT "ВВЕДИТЕ КОЛИЧЕСТВО ЭЛЕМЕНТОВ МАССИВА A>", N%
40 REM ГЕНЕРАЦИЯ МАССИВА :
45 RANDOMIZE(1)
50 FOR I%=1 TO N%
60 X=RND
```



```
70 A%(I%)=N%+N%*2*X
80 NEXT I%
100 PRIZN%=1
120 FOR I%=1 TO N%
130 S1$=STR$(A%(I%)) : LN1%=LEN(S1$) : SUMN%=0
140 FOR K%=1 TO LN1%
150 S2$=MID$(S1$,K%,1) : SUMN%=SUMN%+VAL(S2$)
160 NEXT K%
170 PRINT A%(I%) SUMN%
220 WK1%=INT(SUMN%/4%) : WK2%=WK1%*4%
230 IF WK2% = SUMN% THEN PRIZN%=0 ELSE A%(I%)=0
240 NEXT I%
250 IF PRIZN%=1 THEN PRINT "NO NUMBERS"
330 END
```

Б. Упражнения на обработку с символьных строк

Б.1. Проверить строку, содержащую некоторую фразу (слова, разделенные пробелами, а в конце — точку), на некоторое условие.

```
15 REM МОЖНО ЛИ ИЗ БУКВ ПЕРВЫХ 3-Х СЛОВ ФРАЗЫ СОСТАВИТЬ 2_
ПОСЛЕДНИХ СЛОВА
17 REM ФРАЗЫ?
20 INPUT "ВВЕДИТЕ ИСХОДНУЮ СТРОКУ>", STR1$
40 REM ВВОД ДАННЫХ В СТРОКУ
45 PRINT STR1$
46 REM STOP
50 W1$="" : W2$=""
60 REM СТРОКИ С НАЧАЛА И С КОНЦА
70 LENS%=LEN(STR1$): CNTB%=0
80 FOR I%=1 TO LENS%
85 SYM$=MID$(STR1$,I%,1)
90 IF SYM$=" " THEN CNTB%=CNTB%+1
100 IF CNTB%=3 THEN GOTO 130
110 IF SYM$ > " " THEN W1$=W1$+SYM$
120 NEXT I%
130 PP%=INSTR(STR1$,".") : CNTB%=0
135 IF PP%=0 THEN PRINT "NO POINT" : STOP
140 FOR I%=PP%-1 TO 1 STEP -1
150 SYM$=MID$(STR1$,I%,1)
160 IF SYM$=" " THEN CNTB%=CNTB%+1
170 IF CNTB%=2 THEN GOTO 200
180 IF SYM$ > " " THEN W2$=W2$+SYM$
190 NEXT I%
200 PRINT W1$
210 PRINT W2$
```

```
220 FOR I%=1 TO LEN(W2$)
230 SYM2$=MID$(W2$,I%,1)
235 PRIZN%=0
240 FOR J%=1 TO LEN(W1$)
250 SYM1$=MID$(W1$,J%,1)
260 IF SYM2$=SYM1$ THEN PRIZN%=1
270 NEXT J%
280 IF PRIZN%=0 THEN GOTO 300
290 NEXT I%
300 IF PRIZN%=0 THEN PRINT "NO" ELSE PRINT "OK"
310 END
```

Б.2. Удалить из строки, содержащей некоторую фразу (слова, разделенные пробелами, а в конце — точку), слова, содержащие менее двух гласных букв латиницы.

```
15 REM УДАЛИТЬ ИЗ ФРАЗЫ СЛОВА, СОДЕРЖАЩИЕ МЕНЕЕ 2
17 REM ГЛАСНЫХ БУКВ ЛАТИНИЦЫ
20 INPUT "ВВЕДИТЕ ИСХОДНУЮ СТРОКУ>", STR1$
40 REM ВВОД ДАННЫХ В СТРОКУ
50 NEWS$="" : W1$=""
55 GLAS$="EYUIOAeyuioa"
56 PRINT "ГЛАСНЫЕ БУКВЫ-", GLAS$
60 REM НОВАЯ СТРОКА, ТЕКУЩЕЕ СЛОВО И СТРОКА ГЛАСНЫХ БУКВ
70 LENS%=LEN(STR1$): PP%=INSTR(STR1$,".")
75 IF PP%>0 THEN LENS%=PP%
80 FOR I%=1 TO LENS%
85 SYM$=MID$(STR1$,I%,1)
90 IF SYM$>" " AND I%<LENS% THEN W1$=W1$+SYM$ : GOTO 190
100 CNT%=0
110 FOR J%=1 TO LEN(W1$)
120 SYM$=MID$(W1$,J%,1)
130 FOR K%=1 TO LEN(GLAS$)
140 SYM2$=MID$(GLAS$,K%,1)
150 IF SYM$=SYM2$ THEN CNT%=CNT%+1
160 NEXT K%
170 NEXT J%
175 PRINT W1$, CNT%
177 IF CNT%>=2 THEN NEWS$=NEWS$+W1$+" "
180 W1$=""
190 NEXT I%
200 PRINT "REZULT STRING-",NEWS$
210 END
```


Контрольные вопросы

1. Охарактеризуйте типы данных, принятые в ЯП Basic.
2. В чем различие статических и динамических массивов?
3. Что такое идентификаторы типов данных?
4. Опишите операции над числовыми и строковыми данными.
5. В чем отличие побитовых и логических операций?
6. Какими операторами ЯП может реализоваться цикл типа До?
7. Какими операторами ЯП может реализоваться цикл типа Пока?

Задачи

1. Изменить простейшую диалоговую программу, приведенную в начале главы, таким образом, чтобы на сообщение **настроение плохое** она бы отвечала: **а у меня - хорошее**, и наоборот. При вводе других значений программа должна отвечать — **не поняла!**
2. Изменить программу п. 1 таким образом, чтобы на любые другие значения **nastr** (кроме **хорошее** и **плохое**) программа отвечала бы: **у меня тоже nastr.**
3. Используя вызов таймера, изменить простейшую диалоговую программу, приведенную в начале раздела таким образом, чтобы она приветствовала пользователя словами **доброе утро, добрый день, добрый вечер,** соответственно, с 8:00 до 12:00, с 12:00 до 17:00 и с 17:00 до 22:00.
4. Одномерный массив размерности N с помощью датчика случайных чисел заполняется числами из диапазона [10 000, 99 999]. Найти пятизначное число, которое в 45 раз больше произведения его цифр. Если в массиве нет таких чисел, напечатать сообщение.
5. Заданы 2 массива N*M и M*N. Поменять строки и столбцы в 1-м массиве и добавить их ко второму.
6. Задана текстовая строка, содержащая слова, разделенные пробелами. В конце каждой фразы стоит точка с пробелом. Можно ли из последних 3 слов 1-й фразы составить 3 первых слова 4-й фразы.
7. Задана текстовая строка, содержащая слова, разделенные пробелами. В конце каждой фразы стоит точка с пробелом. Найти самое длинное (короткое) слово 1-й (2-й ...) фразы и заключить его в скобки (кавычки).
8. Задана текстовая строка, содержащая слова, разделенные пробелами. В конце каждой фразы стоит точка с пробелом. Найти слово, содержащее более всего (менее всего) гласных (согласных) букв и напечатать его заглавными символами.

Глава 3
ЯЗЫК ПРОГРАММИРОВАНИЯ PASCAL

На протяжении многих лет язык программирования Pascal довольно часто упоминается как в учебной, так и в научной литературе. Созданный первоначально с педагогическими целями, Pascal оказался крайне удачным не только в силу того, что ему несложно научиться, но и как основа обсуждения языков программирования вообще.

3.1. Общие сведения и история языка

Предварительное описание языка программирования Pascal было опубликовано в 1968 г. его создателем — профессором кафедры вычислительной техники Швейцарского федерального института технологии Н. Виртом (свое название язык получил в честь великого французского математика и механика Блеза Паскаля, создавшего в 1642 г. первую механическую счетную машину — «паскалину», см., напр., [15]). Это был ЯП, представлявший собой развитие языков Algol-60 и Algol-W. В 1970 г. был разработан первый компилятор. Растущий интерес к созданию компиляторов для различных ЭВМ привел к распространению Pascal, и после двух лет его использования потребовалось внести в язык небольшие изменения. Поэтому в 1973 г. было опубликовано «Пересмотренное сообщение», где язык был уже определен в терминах множества символов ISO.

В начале 80-х годов Pascal еще более упрочил свои позиции с появлением трансляторов MS-Pascal и Turbo-Pascal для персональных ЭВМ. С этого времени язык Pascal становится одним из наиболее широко используемых языков программирования для персональных ЭВМ не только как рабочий инструмент пользователя, но и как средство обучения студентов высших учебных заведениях программированию.

Далее в развитии языка стала заметна тенденция его привязки к компьютеру IBM PC, стремление сделать его гибким и удобным инструментом. Следующий шаг усовершенствования — версия Object Pascal. Включив в себя понятие класса, эта версия языка поддерживает предыдущие версии Pascal в реализации фирмы Borland.

Общая структура программы на ЯП Pascal приводится на рис. 3.1. Она состоит из заголовка, разделов объявлений (деклараций) и тела программы. Рассмотрим уже знакомую диалоговую программу на ЯП Pascal (включающую только часть общих разделов):

```
Program Dialog;
Var Nam, Nastr: String;
Begin
     Repeat Writeln('Здравствуйте! Как Вас зовут?');
     Readln(Nam);
     If Nam = '*' Then Break;
     If Nam = '' Then Continue;
         Repeat Writeln('Добрый день, '+ Nam);
         Writeln('Как настроение?');
         Readln(nastr);
         If nastr ='' Then Continue;
         If nastr ='*' Then Break;
         Writeln('У меня тоже '+nastr+', '+Nam);
         Until nastr <> '*'
     Until Nam <> '*';
End;
```

Диалог с данной программой может выглядеть следующим образом.

```
Здравствуйте! как Вас зовут?
Гаврик
Добрый день, Гаврик
Как настроение?
Плохое
У меня тоже Плохое, Гаврик
```

Структурно программа включает:

- заголовок **Program**;
- раздел объявлений переменных **Var**;

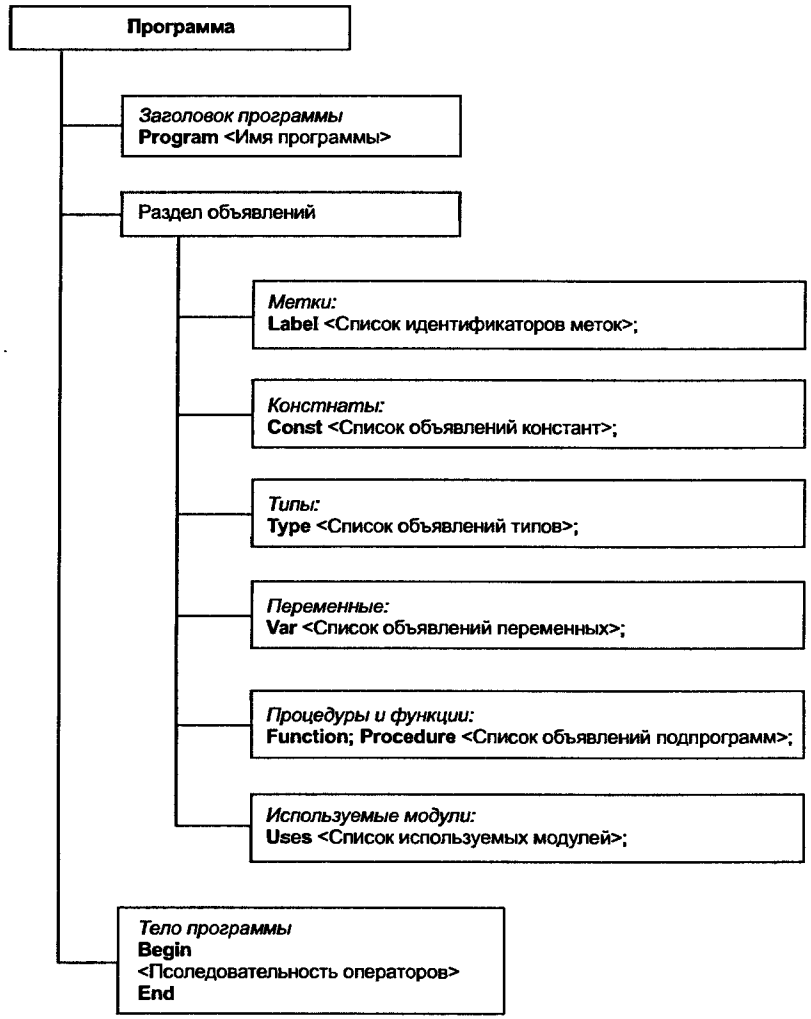


Рис. 3.1. Общая структурная схема программы на языке Pascal

- тело программы (ограничено операторными скобками **Begin** и **End**).

В программе объявлены две строчные переменные Nam и Nastr.

Программа состоит из двух вложенных циклов **Repeat ... Until**. Выход из внешнего и внутреннего циклов происходит

при условии ввода в строке Nam или Nastr символа «*» («звездочка»). С помощью оператора **Break** осуществляется выход во внешний цикл из внутреннего и из внешнего цикла на окончание работы.

При попытке ввести пустую строку в ответ на запрос программа возвращается в заголовок (оператор **Continue**), соответственно, внутреннего или внешнего цикла и повторяет запрос на ввод строки. Управление условными переходами осуществляется условным оператором **If**.

Процедуры Write(), Writeln() и Readln() осуществляют, соответственно, вывод на экран и ввод с клавиатуры строк текста.

Конкатенация строк осуществляется оператором «+».

Рассмотрим далее программу сортировки элементов одномерного массива по возрастанию.

```
Program Sort_Array;
Var W_Array  : Array [1..100] Of Integer;
      I, J, min, I_min, N : Integer;
Begin
      Write('Введите количество элементов массива (N<=100):');
      Readln(N);
      For I := 1 To N Do        // Ввод элементов массива
          Write('Введите элемент массива №', I , ':');
          Readln(W_Array[I])
End;
For I := 1 To N Do     // Внешний цикл прохода по элементам
    Begin I_min := I;    // Начальное задание индекса
                         // минимального элемента
        For J := I + 1 To N Do  //Внутренний цикл поиска
                                //максимального элемента в
                                //пределах от I + 1 до N
           If W_Array[J] < W_Array[I_ min] Then  I_ min:= J
           min := W_Array[I_ min] //Сохранение максимального
                                  //значения
                                  //Перестановка элементов:
                                  //I-го и максимального
           W_Array[I_min]:= W_Array[I];
           W_Array[I];:= min;
    End;
    For I := 1 To N Do      // Вывод отсортированного массива
        Writeln('Элемент массива №', I, '=', W_Array[I];
End;
```

Структурно программа выглядит следующим образом

- заголовок **Program**;
- раздел объявлений переменных **Var**;
- тело программы **Begin ...End**, в том числе:
  - — цикл ввода исходного массива;
  - — два вложенных цикла сортировки массива;
  - — цикл вывода результатов сортировки.

В результате работы программы на экране может быть отображен следующий диалог:

```
Введите количество элементов массива (N<=100):10
Введите элемент массива № 1:2
Введите элемент массива № 2:11
Введите элемент массива № 3:3
Введите элемент массива № 4:1
Введите элемент массива № 5:4
Введите элемент массива № 6:16
Введите элемент массива № 7:5
Введите элемент массива № 8:12
Введите элемент массива № 9:7
Введите элемент массива № 10: 8
Элемент массива № 1 =  16
Элемент массива № 2 =  12
Элемент массива № 3 =  11
Элемент массива № 4 =  8
Элемент массива № 5 =  7
Элемент массива № 6 =  5
Элемент массива № 7 =  4
Элемент массива № 8 =  3
Элемент массива № 9 =  2
Элемент массива № 10 =  1
```

В программе объявляются переменные — массив целых чисел, размерности не более 100 элементов (`W_Array`), целые элементарные данные — `I`, `J`, `min`, `I_min`, `N`.

Циклы в данном примере организуются вложенными операторами **For** `I := 1` **To** `N` **Do** и **For** `J := I + 1` **To** `N` **Do**.

Комментарии, как это очевидно из контекста, начинаются разделителем //.

3.2. Основные структурные единицы языка программирования Pascal

Лексика языка

Программа на ЯП Pascal представляет собой текстовый файл, не содержащий специальной разметки (плоский ASCII-файл).

Алфавит — набор символов, из которых можно составлять слова языка — названия переменных, файлов и пр., а также зарезервированные слова. Чтобы задать алфавит, необходимо определить, что такое символ:

```
<Символ> ::= <Буква>|<Цифра>|<Ограничитель>.
```

При записи текстов программ на языке Pascal разрешается использовать прописные (заглавные) и строчные буквы латинского алфавита и знак «_» («подчеркивание»), т. е. понятие <Буква> задается следующей формулой:

```
<Буква> ::= A|B|C|D|E|F|G|H|I|G|K|L|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|_|
            a|b|c|d|e|f|g|h|i|g|k|l|n|o|p|q|r|s|t|u|v|w|x|y|z|.
```

Для определения понятия <цифра> используется формула:

```
<Цифра> ::= 0|1|2|3|4|5|6|7|8|9.
```

Ограничители в тексте программы разделяют элементы фраз. Это могут быть знаки операций, скобки и другие служебные знаки или служебные слова, для которых не удалось подобрать служебные знаки.

```
<Ограничитель> ::= <Служебный знак>|<Служебное слово>
```

Служебные знаки, выполняющие в языке определенные функции, можно разделить на следующие группы:

```
<Служебный знак> ::= <Знак пунктуации>|
            <Знак операции>|<Разделитель>
```

Приведем определение каждой из групп:

```
<Знак пунктуации> ::= {|}|(*|*)|(|)|[|]|;|:|'|#|@|$|.|,|..|=|:=
```

Таблица 3.1. **Знаки пунктуации ЯП Pascal**

Знаки	Назначение
(* *) { }	Скобки комментария. Текст, заключенный между скобками, поясняет алгоритм и не является его частью
[ ]	Задание индексов массива, размера строки, элементов множества
( )	Выделение части выражения, задание списков параметров
;	Отделение одного предложения программы от другого, разделение параметров (в части объявления)
:	Отделение переменной или константы от типа (в части объявлений), отделение метки от оператора, следующего за ней
,	Разделение элементов списка, параметров процедуры и функции при вызове
@	Обозначение адреса (переменной, константы, процедуры, функции, метода)
$	Признак числа в шестнадцатеричной системе, обозначение директивы компилятора
#	Обозначение символа по его коду
..	Разделение границ диапазона в типе-диапазоне
:=	Знак оператора присваивания
=	Отдвление идентификатора типа (константы) от его описания (значения)
'	Апостроф — признак символа ипи строковой константы

Назначение знаков пунктуации приводится в табл. 3.1.

```
<Знак операции> ::= +|-|/|*|^|=|>|>=|<|<=
<Раэделитель> ::= <Пробел>|<Управляющий символ
(коды от 0 до 31)>
```

Идентификаторы. Для именования различных объектов программы служат языковые элементы (слова), называемые идентификаторами. Идентификатор определяется как последовательность букв и цифр, начинающаяся с буквы:

```
<Идентификатор> ::= <Буква>|<Идентификатор><Буква>|
      <Идентификатор><Цифра>
```


Компилятор Pascal-программы не различает строчные и прописные буквы, поэтому запись тех или иных идентификаторов программистом буквами разных регистров обычно используется для целей структурирования и удобочитаемости текста, как это и было сделано ранее в примерах Pascal-программ.

Список служебных (зарезервированных) слов ЯП Pascal приведен в табл. 3.2.

Таблица 3.2. **Зарезервированные слова ЯП Pascal**

And	Array	As	Asm	Begin	Case
Class	Const	Constructor	Destructor	Dispinterface	Div
Do	Downto	Else	End	Except	Exports
File	Finalization	Finally	For	Function	Goto
If	Implementation	In	Inherited	Initialization	Inline
Interface	Is	Label	Library	Mod	Nil
Not	Object	Of	Or	Out	Packed
Procedure	Program	Property	Raise	Record	Repeat
Resourcestring	Set	Shl	Shr	String	Then
Threadvar	To	Try	Type	Unit	Until
Uses	Var	While	With	Xor	

Переменные и константы

Данные делятся на переменные и константы.

Переменные — данные, значения которых могут изменяться в процессе выполнения программы. Например, для программы вычисления площади круга необходимо объявить две переменные: переменную R, в которую будем заносить значение радиуса окружности, и переменную S для вычисления площади круга по формуле

$$S = \pi R^2.$$

Константы — это данные, значения которых не меняются в процессе выполнения программы. В примере, описанном выше, константой является число π.

Каждая переменная и константа должна иметь свое уникальное имя (идентификатор). Для объявления переменных и констант в Pascal-программе выделены особые синтаксические разделы.

Раздел объявления констант начинается со служебного слова **Const** и содержит перечень всех используемых в программе констант, например, описание константы Radius, принимающей значение 4, в программе выглядит следующим образом:

```
Const
      Radius = 4;
```

Раздел объявления переменных начинается со служебного слова **Var** и содержит описание всех переменных:

```
Var
      Radius: Integer
```

Типы данных

С понятием данных тесно связано понятие *типа* данных. Тип — совокупность таких атрибутов данного, которая, с одной стороны, задает границы его изменения, а с другой, — определяет множество операций над ним.

Типы данных в ЯП Pascal принято делить на простые (или базовые) и структурированные.

К простым (базовым) типам в языке Pascal относятся:

- целый;
- вещественный;
- логический;
- символьный;
- перечисляемый;
- диапазон.

Структурированные типы описывают наборы однотипных или разнотипных данных. Типы данных, образующих набор, в свою очередь могут быть как простыми, так и структурированными. Если тип компонент набора является структурированным, то говорят, что получаемый в результате структурированный тип имеет более одного уровня структурирования. Структурированный тип может иметь неограниченное количество уровней структурирования.

К стандартным структурированным типам относятся:

- массив;
- запись;
- строка;
- множество;
- файл.

Целый тип данных. Целый тип присваивается данным (переменным и константам), которые во время работы программы могут принимать лишь целочисленные значения.

Например, программа вычисления функции «факториал» для натурального числа N ($y = N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$) оперирует с целыми числовыми значениями: N и y — натуральные числа.

В стандартном Pascal определен единственный целый тип данных Integer. В современных реализациях языка целый тип данных представлен рядом типов (табл. 3.3)

Например, для переменных X, Y, Z, описанных в разделе объявления переменных, как

```
Var
    X: Byte;
    Y: Smallint;
    Z: Word;,
```

операторы X:=2000; Y:=-40000; Z:=-2 будут некорректными, так как переменная X не может принимать значение, большее чем 255, Y не может быть меньше, чем -32768, Z должно быть положительным.

Логический тип данных. Данные логического типа (Boolean) в стандарте языка могут принимать одно из двух значений: true или false. Переменная или константа логического типа занимает 1 байт, в который записывается 1, если переменная или константа имеет значение true и 0 в противном случае.

Например, для переменной Flag, описанной в разделе объявления переменных как

```
Var
    Flag: Boolean,
```

корректны операторы:

```
Flag := true;
Flag := false;
```

Таблица 3.3. **Типы данных в ЯП Pascal**

Название типа	Занимаемый размер в байтах	Область изменения данных
	Целое данное	
Integer (целое со знаком)	2 (4)	-32768..32767 ($-2^{31}..2^{31}-1$)
Shortint (целое со знаком)	1	-128..127
Smallint (целое со знаком)	2	-32768..32767
Longint (целое со знаком)	4	$-2^{31}..2^{31}-1$
Byte (целое без знака)	1	0..255
Word (целое без знака)	2	$0..2^{16}-1$
Longword (целое без знака)	4	0..4294967295
	Логическое данное	
Boolean	1	True..False
ByteBool	1	True..False
WordBool	2	True..False
LongBool	4	True..False
	Вещественное данное	
Real	8 (15—16 значащих цифр)	$5.0 \cdot 10^{-324}..1 \cdot 7 \cdot 10^{308}$
Real48	6 (11—12 значащих цифр)	$2.9 \cdot 10^{-39}..1 \cdot 7 \cdot 10^{38}$
Single	4 (7—8 значащих цифр)	$1.5 \cdot 10^{-45}..3.4 \cdot 10^{38}$
Double	8 (15—16 значащих цифр)	$5.0 \cdot 10^{-324}..1.7 \cdot 10^{308}$
Extended	10 (19—20 значащих цифр)	$3.6 \cdot 10^{-4951}..1.1 \cdot 10^{4932}$
Comp	8 (19—20 значащих цифр)	$-2^{63}+1..2^{63}-1$
Currency	8 (19—20 значащих цифр)	-922337203685477.5808.. 922337203685477.5807

В современных реализациях языка (Turbo Pascal V 7.0, Object Pascal) добавлены еще три логических типа (для совместимости с другими языками программирования и со средой Windows) — табл. 3.4. Основные отличия этих типов от стандартного заключаются, во-первых, в фактическом размере (в байтах), а во-вторых, — в величине, соответствующей значению `true`. Для всех логических типов значению `false` соответствует число «0», записанное в соответствующее количество байтов. Значению же `true` в случае стандартного типа соответствует только «1», записанная в байт, а в случае других логических типов — любое число, отличное от «1».

Таблица 3.4. **Расширение типов `True`, `False`**

Boolean	ByteBool, WordBool, LongBool
false < true	false <> true
Ord(false) = 0	Ord(false) = 0
Ord(true) = 1	Ord(true) <> 0
Succ(false) = true	Succ(false) = true
Pred(true) = false	Pred(false) = true

Символьный тип данных. Данные стандартного символьного типа представляют собой символы ASCII. Переменная или константа символьного типа занимает 1 байт памяти. В соответствии с синтаксисом языка значение символа заключается в одинарные кавычки: `'P', 'a', 's', 'c', 'a', 'l'`.

При вызове функции `Ord(Ch)`, где `Ch` — значение символьного типа (`Char`), возвращается порядковый номер `Ch`. Любой символ ASCII-кода можно получить с помощью стандартной функции `Chr(X)`, где `X` — целое значение типа `byte` (или используя знак «#» перед числом, например `#30 = Chr(30)`).

Для следующих объявлений переменных

```
var
C:Char;
B:Byte;
```

справедливы операторы:

```
C := #3;
B := Ord(C);
B := 125;
C := Chr(B);
```

Перечисляемый тип. Перечисляемый тип данных задается упорядоченным набором идентификаторов, с которыми могут совпадать значения переменной (или константы) этого типа. Список идентификаторов указывается в круглых скобках, а сами идентификаторы разделяются запятыми:

```
<Перечисляемый тип> := <Идентификатор типа> =
      (<Список идентификаторов>);
<Идентификатор типа > := < Идентификатор>
<Список идентификаторов> := <Идентификатор>|
      <Идентификатор>, <Список идентификаторов>
```

Упорядочение наборов выполняется в соответствии с последовательностью, в которой перечисляются идентификаторы. Порядковый номер перечисляемой константы определяется ее позицией в списке идентификаторов при объявлении. Первый идентификатор в списке имеет порядковый номер 0, второй — порядковый номер 1 и т. д. Один и тот же идентификатор можно использовать в объявлении только одного перечисляемого типа.

Приведем примеры перечисляемого типа:

```
Type
     Color = (Red, Green, Blue, Black);
     Operator = (Plus, Minus, Multiply, Divide);
```

Порядковые типы. Типы логический, символьный, целый и перечисляемый относятся к так называемым порядковым типам, т. е. представляют собой множества значений, на которых определен порядок следования.

Для величин порядкового типа существуют стандартные процедуры и функции, позволяющие выполнить ряд действий. В табл. 3.5 приводятся эти процедуры и функции.

Тип Диапазон. Представляет собой диапазон значений из порядкового типа, называемого главным типом. Определение типа-диапазона задает все значения из главного типа, находящиеся между наименьшим и наибольшим значением, включая сами границы, и имеет следующий синтаксис:

```
<Тип-диапазон> := <Имя типа> = <Мин. значение>..
     <Макс. значение>;
<Имя типа> := <Идентификатор>
<Мин. значение> := <Константа>
<Макс. значение> := <Константа>
```


Таблица 3.5. **Стандартные процедуры и функции над величинами порядкового типа**

Название	Действие	Тип аргумента (параметра)	Тип результата	Примеры
`Ord(X)`	Возвращает целое число, которое показывает, какую позицию занимает значение X по отношению к другим значениям	Логический, символьный, целый, перечислимый	`Longint`	`Ord(false)=0; Ord('1')=34; Ord(plus)=0`
`Pred(X)`	Возвращает предшествующее значение X. При применении к первому элементу возникает ошибка	Логический, символьный, целый, перечислимый	Тот же, что и у аргумента	`Pred(true)=false; Pred(0)=-1; Pred(Minus)=Plus Pred(Plus)-error!`
`Succ(X)`	Возвращает последующее значение X. При применении к последнему элементу возникает ошибка	Логический, символьный, целый, перечислимый	Тот же, что и у аргумента	`Succ(false)=true; Succ('1')='2'; Succ(0)=1`
`Odd(X)`	Проверка аргумента на нечетность: возвращает `true`, если аргумент нечетный, и `false`, если аргумент четный	`Longint`	`Boolean`	`Odd(3)=true; Odd(6)=false;`
`Inc(X[,N])`	Процедура. Увеличивает значение переменной X на 1 (если второй параметр отсутствует) или на N	Тип X — целый, логический, символьный, перечислимый; Тип N — `Longint`	Тот же	`Inc(X)` аналогично `X:=X+1`, если X — целое; `Succ(X)`, если X — перечисляемое
`Dec(X[,N])`	Процедура. Уменьшает значение переменной X на 1 (если второй параметр отсутствует) или на N	Тип X — целый, логический, символьный, перечислимый; Тип N — `Longint`	Тот же	`Dec(X)` аналогично `X:=X-1`, если X — целое; `-Pred (X)`, если X-перечисляемое

Обе константы должны быть одного порядкового типа, при этом минимальное значение не должно превышать максимального.

Приведем примеры типов-диапазонов:

```
0 .. 999
-100 .. 100
Red .. Blue
```

Переменная типа диапазон имеет все свойства переменных главного типа, но ее значение на этапе выполнения должно принадлежать указанному интервалу.

Для задания границ диапазона разрешается использовать константные выражения, в связи с этим может возникнуть синтаксическая неопределенность. Рассмотрим следующие объявления:

```
Const
     X = 5;
     Y = 10;
Type
     Operator = (Plus, Minus, Multiply, Divide);
     Height = (X - Y) * 2  ..  (X + Y) * 2;
```

В соответствии с правилами синтаксиса языка любое определение типа, начинающееся с круглой скобки, воспринимается, как определение перечисляемого типа (см. определение типа `Operator`). Однако целью объявления `Height` является задание типа-диапазона. Решением этой проблемы является либо преобразование первого выражения, задающего минимальное значение, так, чтобы оно не начиналось с круглой скобки, либо задание другой константы, равной значению этого выражения, и затем использование этой константы в определении типа, например:

```
Type
     Height = 2 * (X - Y)  ..  (X + Y) * 2;
```

Вещественный тип данных — значения, записанные в памяти в виде чисел с плавающей точкой. Область возможного изменения значений определяется размером (в байтах), отводимым под конкретную реализацию типа.

Вещественный тип в стандартном ЯП Pascal называется `Real`. Помимо типа `Real` в современных реализациях Pascal определены еще шесть стандартных вещественных типов. Каждый тип характеризуется своей областью изменения возможных значений (см. табл. 3.3).

Тип `Comp` фактически является целым типом расширенного диапазона, но при этом на считается порядковым (т. е. к переменным типа `Comp` нельзя применять процедуры и функции, определенные только для порядковых типов — `Inc()`, `Dec()` и т. п.).

Выбор конкретного типа для переменной связан с требуемой точностью вычислений.

Для переменных вещественного типа определены две функции, позволяющие преобразовать переменную вещественного типа в переменную целого типа. В качестве аргументов функций выступают значения вещественного типа, а результат принадлежит целому типу. Названия и результат действия этих функций приведены в табл. 3.6.

Таблица 3.6. **Функции преобразования вещественного данного в целое**

Название	Назначение	Примеры
`Round(X)`	Округление вещественного числа до целого	`Round(3.456) = 3;` `Round(5.678) = 6;`
`Trunc(X)`	Выделение целой части числа	`Trunc(3.456) = 3;` `Trunc(5.678) = 5;`

Типизованные константы. Они могут быть использованы точно так же, как переменные того же самого типа, и могут появляться в левой части оператора присваивания. Отметим, что типизированные константы инициализируются только один раз — в начале выполнения программы. Таким образом, при каждом новом входе в процедуру или функцию локально объявленные типизированные константы заново не инициализируются.

Синтаксис объявления типизированной константы следующий:

```
<Типизированная константа> ::= <Идентификатор>:
        <Тип данных> = <Значение>;
```

Константы простого типа. Объявление типизированной константы простого типа содержит в своем описании указание на простой тип данных (например, `Integer`, `Real` или `Char`):

```
Const
      Maximum   : Integer = 999;
      Factor    : Real    = -0.1;
      Breakchar : Char    = #3;
```

Поскольку типизированная константа фактически представляет собой переменную с константным значением, она не может заменять обычную константу. Например, она не может использоваться в объявлении других констант или типов.

```
Const
      Min : Integer = 1;
      Max : Integer = 1000;
Type
      Vector = array [Min..Max] of Integer;
```

Объявление Vector является недопустимым, поскольку Min и Max являются типизированными константами.

3.3. Выражения, операции, операторы

Выражения и операции

Выражение — это синтаксическая единица языка, задающая порядок и способ вычисления некоторого значения. В соответствии с правилами формирования выражение представляет собой последовательность операндов, соединяющихся друг с другом знаками операций. Некоторые фрагменты выражения могут быть заключены в круглые скобки.

```
<Выражение> ::= <Операнд>|
      <Выражение><Операция><Операнд>|
      <Операнд><Операция><Выражение>|
      (<Выражение>)<Операция><Операнд>|
      <Операнд><Операция>(<Выражение>)
```

В качестве <Операнда> в конструкции выступают переменные, константы, функции:

```
<Операнд> ::= <Переменная>|<Константа>|<Функция>
```

Операции подразделяются на несколько групп:
- числовые/арифметические;
- операции отношения;
- логические;
- операции над битами информации.

Каждой группе операций соответствуют определенные типы переменных и констант.

По количеству операндов, участвующих в операциях, их делят на унарные (операции с одним операндом) и бинарные (операции с двумя операндами). В бинарных операциях используется обычное алгебраическое представление, например: A + B. В унарных операциях операция всегда предшествует операнду, например -B.

В более сложных выражениях порядок, в котором выполняются операции, соответствует приоритету операций (табл. 3.7).

Таблица 3.7. **Операции ЯП Pascal**

Операции	Приоритет	Категория
@, Not, +, -	Первый (высший)	Унарные операции
*, /, Div, Mod, And, Shl, Shr	Второй	Операции умножения
+, -, Or, Xor	Третий	Операции сложения
=, <>, <, >, <=, >=, In	Четвертый (низший)	Операции отношения

Для определения старшинства операций применяются три основных правила:
- операнд, находящийся между двумя операциями с различными приоритетами, связывается с операцией, имеющей более высокий приоритет;
- операнд, находящийся между двумя операциями с равными приоритетами, связывается с операцией, которая находится слева от него;
- выражение, заключенное в круглые скобки, перед выполнением вычисляется, как отдельный операнд.

Операции с равным приоритетом обычно выполняются слева направо, хотя иногда компилятор при генерации оптимального кода может переупорядочить операнды.

Числовые операции. Числовые (арифметические) операции могут применяться только к операндам целых и вещественных типов (см. табл. 3.8).

В качестве операндов числовых операций могут выступать также стандартные числовые функции, т. е. функции с результатом целого или вещественного типа, аргументами которых являются выражения целого или вещественного типа. В табл. 3.9 приведены характеристики некоторых таких функций.

Таблица 3.8. **Арифметические операции**

Знак	Операция	Число операндов	Тип операндов	Результат	Тип результата
+	Признак положительного числа	Унарная (1 операнд)	Целый Вещественный	Не меняет значения операнда	Целый Вещественный
-	Признак отрицательного числа	Унарная	Целый Вещественный	Меняет значение операнда на противоположный	Целый Вещественный
+	Сложение	Бинарная (2 операнда)	Целый Хотя бы один вещественный	Сумма двух чисел	Целый Вещественный
-	Вычитание	Бинарная	Целый Хотя бы один вещественный	Разность двух чисел	Целый Вещественный
*	Умножение	Бинарная	Целый Хотя бы один вещественный	Произведение двух чисел	Целый Вещественный
/	Деление	Бинарная	Целый или вещественный	Частное от деления двух чисел	Вещественный
div	Деление целых чисел	Бинарная	Целый	Целая часть от деления целых чисел	Целый
mod	Остаток от деления целых чисел	Бинарная	Целый	Остаток от деления целых чисел	Целый

Таблица 3.9. **Числовые функции ЯП Pascal**

Функция	Действие	Тип аргумента	Тип результата
Abs(X)	Модуль (абсолютная величина) аргумента	Целый Вещественный	Целый Вещественный
Arccos(X)	Арккосинус аргумента	Вещественный, Abs(X)<=1	Вещественный
Arcsin(X)	Арксинус аргумента	Вещественный, Abs(X)<=1	Вещественный
ArcTan(X)	Арктангенс аргумента	Вещественный	Вещественный
Cos(X)	Косинус аргумента	Вещественный	Вещественный
Cotan(X)	Котангенс аргумента	Вещественный	Вещественный
Hypot(X,Y)	Гипотенуза прямоугольного треугольника с катетами X, Y	Вещественный	Вещественный

Окончание табл. 3.9

Функция	Действие	Тип аргумента	Тип результата
Ceil(X)	Минимальное целое число, большее или равное X	Целый или вещественный	Целый Например: Ceil(-2.8) = -2 Ceil(2.8) = 3
Log10(X)	Десятичный логарифм аргумента	Вещественный	Вещественный
Log2(X)	Логарифм аргумента по основанию 2	Вещественный	Вещественный
LogN(N,X)	Логарифм X по основанию N	Вещественные	Вещественный
Max(A,B)	Максимальное из чисел A, B	Оба целые или оба вещественные	Совпадает с типом аргументов
Min (A,B)	Минимальное из чисел A, B	Оба целые или оба вещественные	Совпадает с типом аргументов
Tan(X)	Тангенс аргумента	Вещественный	Вещественный
Sqr(X)	Квадрат аргумента	Вещественный	Вещественный
Sqrt(X)	Квадратный корень аргумента	Вещественный	Вещественный
Pi	Число π		Вещественный
Power(X,EY)	Возведение X в степень Y	Вещественные	Вещественный

Примеры записи арифметических выражений:

```
(Max(X,Y)+sqrt(Z))*2;
Ceil(Tan(X)-1);
Pi* Sqr(R);
(Cos(X)+Y)/Z;
Log2(X)+X/Y;
Power(P,Q)/Power(R,(S+T))
```

Операции отношения. В результате выполнения операций отношения получается значение логического типа — `true` или `false` (см. табл. 3.10). При этом операнды, участвующие в операции, должны быть сравнимых типов, например: целого и целого; целого и вещественного; логического и логического и т. п.

Логические операции. Логические операции применяются к операндам логического типа. Результат выполнения логических операций тоже логического типа. Вычисление логических выражений происходит в соответствии с таблицами истинности логических операций (см. табл. 1.7—1.9).

Таблица 3.10. **Операции сравнения**

Операция	Описание	Примеры
=	Равно	X = Abs(X) → true, если X — положительное число, false, если X — отрицательное; A = A → true; 2 = 5 → false
<>	Не равно	X <> Abs(X) → true, если X — отрицательное число, false, если X — положительное; A <> A → false; 2 <> 5 → true
<	Меньше	A < A → false; 2 < 5 → true
<=	Меньше или равно	A <= A → true; Max(X,Y) <= X → false
>	Больше	A+1 > A → true; Min(X,Y) > X → false
>=	Больше или равно	A >= A → true; Max(X,Y) >= X → true

Примеры записи логических выражений:

```
(X>=0) and (X<=1);
((X>0) and (X<0.5)) or (X>3);
(N mod 2 = 0) and (N>0)
```

Битовые операции. Как это уже отмечалось в гл. 1 (см. табл. 1.15) в языках программирования могут присутствовать операции над битовыми строками. В табл. 3.11 приводятся описания данных операций в ЯП Pascal.

Таблица 3.11. **Операции с битами информации**

Операция	Описание	Число операндов	Тип операндов и результата	Примеры
Not	Битовое отрицание	Унарная (1)	Целый	Not 3 = -4; Not (-4) = 3
And	Битовое И	Бинарная (2)	Целый	3 And 5 = 1; 15 And 7 = 7
Or	Битовое ИЛИ	Бинарная	Целый	3 Or 5 = 7; 15 Or 7 = 15
Xor	Битовое исключающее ИЛИ	Бинарная	Целый	3 Xor 5 = 6; 15 Xor 7 = 8
Shl	Сдвиг влево	Бинарная	Целый	10 Shl 2 = 40; 11 Shl 2 = 44
Shr	Сдвиг вправо	Бинарная	Целый	10 Shr 2 = 2; 11 Shr 2 = 2

Операции I Shl J и I Shr J сдвигают значение I влево или вправо на J битов. Тип результата будет таким же, как тип I.

Операторы языка

Оператор присваивания. С помощью этого оператора переменной, имя которой указывается в левой части оператора, присваивается значение выражения, стоящего в правой части и вычисляемого перед присваиванием. Синтаксис оператора:

```
<Оператор присваивания> ::= <Переменная> := <Выражение>
```

В левой части оператора присваивания может стоять переменная любого типа (кроме типа File), но при этом типы переменной и выражения должны совпадать.

Например:

- вычислить длину окружности L (тип Real):

```
L := 2*Pi*R
R – радиус окружности (тип – Integer или Real);
Pi – константа, равная значению π;
```

- присвоить переменной Flag (тип Boolean) значение true, если переменная X (тип Real) находится в интервале (0;1), и false — в противном случае:

```
Flag := (X > 0) and (X < 1)
```

- присвоить переменной Rect (тип Boolean) значение true, если из отрезков длиной X, Y, Z (тип Real) можно построить треугольник, и false — в противном случае:

```
Rect := (X < Y + Z) and (Y < X + Z) and (Z < X + Y)
```

Оператор безусловного перехода. Оператор **Goto** позволяет изменить стандартный последовательный порядок выполнения операторов и передать управление заданному оператору, которому в этом случае должна предшествовать <Метка>. Эта же метка должна быть указана в операторе **Goto**.

Синтаксис оператора безусловного перехода:

```
<Оператор безусловного перехода> ::= Goto <Метка>;
<Метка> ::= <Идентификатор>|<Целое число от 0 до 9999>
```

Все метки должны быть перечислены в программном разделе объявления меток, который начинается со служебного слова **Label**:

```
Label 99, 100, MyLabel;
```

Одной меткой помечается только один оператор. Метка от оператора отделяется двоеточием:

```
<Помеченный оператор> ::= <Метка>:<Оператор>
```

При использовании оператора безусловного перехода должны соблюдаться следующие правила:

- метка, указанная в операторе перехода, должна находиться в том же блоке или модуле, что и сам оператор перехода. Другими словами, не допускаются переходы из процедуры или функции или внутрь нее;
- переход извне внутрь составного оператора (т. е. переход на более глубокий уровень вложенности) может вызвать некорректную работу программы, хотя компилятор не выдает сообщения об ошибке.

Следует также иметь в виду, что слишком частое применение оператора безусловного перехода ухудшает логику программы.

Пример:

```
If Odd(I) Then
Begin X := Z * X;
     Goto 1;
End;
I := I div 2;
X := Sqr(X);
1:;                    // Это пустой оператор
```

Пустой оператор. Пустой оператор не выполняет никакого действия в программе, но может иногда потребоваться для осуществления на него безусловного перехода.

Пустой оператор может отображаться в программе точкой с запятой, отделяющей пустой оператор от предшествующего, или меткой (см. пример выше).

Условный оператор. Назначение условного оператора — обеспечить ветвление алгоритма в зависимости от выполнения некоторого условия.

В языке Pascal существует две формы условного оператора — **If** и **Case**.

Условный оператор If имеет полную и неполную формы.

Условный оператор в неполной форме

```
If <Логическое выражение> Then <оператор>
```

выполняется следующим образом: сначала вычисляется <Логическое выражение>, затем, если его значение есть `true` (истина), выполняется оператор, стоящий за **Then**. Если значение логического выражения `false` (ложь), условный оператор действует как пустой.

В случае использования условного оператора в полной форме

```
If  <Логическое выражение> Then
    <Оператор-true>
Else <Оператор-false>),
```

<Оператор-true> выполняется, если значение логического выражения истинно, а <Оператор-false> — если ложно.

Рассмотрим пример.

Написать последовательность действий для решения следующей задачи.

Присвоить переменной y значение $\sin x$, если $x > 0$, и значение $\cos x$, если $x \le 0$.

В этом случае необходимо использовать условный оператор в полной форме:

```
if x>0 Then y:=sin(x) Else y:=cos(x);
```

Изменим условие задачи следующим образом: присвоить переменной y значение $\sin x$, если $x < 1$, и значение $\cos x$, если $x \ge 2$.

Переменная y должна поменять свое значение только в том случае, если переменная x принадлежит одному из интервалов: $(-\infty, 1]$ или $[2, +\infty)$. Если же лежит в интервале $[1, 2)$ переменная y не меняет своего значения.

Такой алгоритм может быть реализован последовательностью из двух условных операторов в неполной форме:

```
If x<1 Then y:= sin(x);
If x>=2 Then y := cos(x);
```

Второй оператор приведенной записи решения может быть использован в качестве <Оператора-false> в условном операторе в полной форме. Тогда решение можно записать следующим образом:

```
If x<1 Then y:= sin(x) Else If x>=2 Then y := cos(x)
```

Рассмотрим задачу поиска корней квадратного уравнения $ax^2 + bx + x = 0$.

Значения корней уравнения вычисляются по следующей формуле:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

и существуют при условии, что $D = b^2 - 4ac \geq 0$. Если же $D = 0$, то корень — один и вычисляется по формуле $x = \frac{-b}{2a}$.

Для решения задачи объявим следующие переменные:

a, b, c (тип Real) — коэффициенты уравнения;

D (тип Real) — значение дискриминанта;

X1 (тип Real) — первый корень уравнения;

X2 (тип Real) — второй корень уравнения.

Общий вид программы вычисления корней квадратного уравнения:

```
Program Square equation
Var a, b, c, D, X1, X2 : Real;
D := b*b + 4*a*c;                    //Вычисление дискриминанта
If D < 0 Then Writeln('Корней нет');
If D = 0 Then
     Begin X1:= -b/(2*a);            //Вычисление
                                     // единственного корня
          Writeln('Уравнение имеет один корень - ',X1)
     End;
If D > 0 Then
     Begin X1 := (-b + sqrt(D)) / (2*a);
           X2 := (-b - sqrt(D)) / (2*a);   //Вычисление
                                           // двух корней
           Writeln('Уравнение имеет два корня - ', X1, X2)
     End;
```

Заметим, что в приведенном примере в любом случае осуществляется проверка всех трех условий. Чтобы избежать этого, можно воспользоваться вложенными условными операторами (т. е. использовать условные операторы в качестве <Оператора-true> и <Оператора-false>). Тогда фрагмент программы можно записать следующим образом:


```
D := b*b + 4*a*c;
If D < 0 Then Writeln('Корней нет')
Else If D = 0 Then
Begin x1:= -b/(2*a);
      Writeln('Уравнение имеет один корень - ',x1)
End;
Else
Begin
      x1 := (-b + sqrt(D)) / (2*a);
      x2 := (-b - sqrt(D)) / (2*a);
      Writeln('Уравнение имеет два корня - ', x1, x2)
End;
```

Условный оператор Case предназначен для организации выбора одной из любого количества ветвей алгоритма в зависимости от значения некоторого выражения. Синтаксис оператора:

```
<Условный оператор Case> ::= Case <Выражение> Of
     <Последовательность Case-ветвей>
     Else <Оператор-Else> End;
<Последовательность Case-ветвей> ::=  <Case-ветвь>;|
     <Последовательность Case-ветвей> <Case-ветвь>
<Case-ветвь> ::= <Константа>: <Оператор>.
```

Оператор <**Case**-ветви> выполняется в том случае, если выражение в заголовке оператора **Case** принимает значение, равное константе <**Case**-ветви>.

Структура оператора, принятая в языке:

```
Case S Of
     C_V1: <Оператор-V1>;
     C_V2: <Оператор-V2>;
     ..................
     C_VN: <Оператор-VN>;
Else
     <Оператор-Else>
End;
```

Здесь S — выражение, значение которого вычисляется; C_V1, C_V2, ..., C_VN — константы, которые определяют разветвление

алгоритма и последовательно сравниваются со значением выражения S; <Оператор-V1>, ..., <Оператор-VN> — операторы ветвей; <Оператор-Else> выполняется в случае, если значение выражения не совпадает ни с одной из констант. При использовании оператора **Case** необходимо помнить о том, что значение выражения и константы должны быть одного типа.

Пример. Присвоить строке S значение дня недели для заданного числа D при условии, что в месяце 31 день и 1-е число — понедельник.

Для построения алгоритма воспользуемся операцией mod, позволяющей вычислить остаток от деления двух чисел, и условием, что 1-е число — понедельник. Тогда можно записать следующий оператор **Case**:

```
Case D mod 7 Of
     1: S := 'понедельник';
     2: S := 'вторник';
     3: S := 'среда';
     4: S := 'четверг';
     5: S := 'пятница';
     6: S := 'суббота';
     0: S := 'воскресенье';
End;
```

С помощью оператора вычисляется остаток от деления D на 7 и, в зависимости от полученного значения, в переменную S заносится строка, соответствующая дню недели.

В предложенной записи отсутствует оператор **Else**. Это объясняется тем, что выражение D mod 7 может принимать только одно из указанных значений. Запись алгоритма аналогична, например, следующей записи в полной форме:

```
Case D mod 7 Of
     1: S := 'понедельник';
     2: S := 'вторник';
     3: S := 'среда';
     4: S := 'четверг';
     5: S := 'пятница';
     6: S := 'суббота';
Else
     S := 'воскресенье';
End;
```


Оператор цикла. Обобщенный оператор цикла имеет следующий синтаксис:

```
<Оператор цикла> ::= <Заголовок цикла> <Тело цикла>
<Тело цикла> ::= <Оператор>
```

Заголовок цикла содержит сведения об условиях выполнения циклических действий, а тело цикла представляет собой последовательность самих действий.

В языке Pascal реализовано три разновидности оператора цикла — операторы **For**, **While** и **Repeat**.

***Оператор* For** — синтаксис:

```
<Оператор For> ::= For <Переменная цикла> :=
     <Выражение-начало> To <Выражение-конец>
     Do <Тело цикла>|
For <Переменная цикла> := <Выражение-начало>
     Downto <Выражение-конец> Do <Тело цикла>
```

Переменная цикла должна принадлежать счетному множеству значений (т. е. должна быть целого или перечисляемого типа). <Выражение-начало>, <Выражение-конец> и <Переменная цикла> должны быть совместимы по типу.

Выполнение оператора происходит по следующему алгоритму:

1. Вычисляется <Выражение-начало> (V_Start) и присваивается <Переменной цикла> i, вычисляется <Выражение-конец> (V_Fin).

2. Значение i сравнивается с V_Fin. Если $i \le V\_Fin$ (в случае использования оператора с перечислением **To**) или $i \ge V\_Fin$ (в случае использования оператора с перечислением **Downto**) — переход к п. 4, иначе — переход к п. 6.

3. Выполняется <Тело цикла>.

4. В случае использования оператора с перечислением **To** переменной цикла присваивается следующее большее значение (например, если i целого типа, то i:=i+1), а в случае использования оператора с перечислением **Downto** — следующее меньшее значение (например, если i целого типа, то i:=i-1). Переход к п. 2.

5. Завершение выполнения оператора.

Из представленного алгоритма видно, что если $V\_Start > V\_Fin$ (в случае использования оператора с перечислением **To**)

или V_Start < V_Fin (в случае использования оператора с перечислением **Downto**), то тело цикла ни разу не выполнится.

Количество шагов цикла (N_Step) может быть вычислено по следующим правилам:

- в случае оператора **To** — N_Step = V_Fin - V_Start + 1;
- для оператора **Downto** — N_Step = V_Start - V_Fin + 1.

Примеры.

1. Вычислить значение функции $y = N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$ для $N = 100$.

Введем переменную целого типа y и присвоим ей начальное значение 1:

```
y := 1;
```

Далее введем переменную целого типа i, которая будет возрастать от 2 до 100, и воспользуемся оператором цикла **For**:

```
For i := 2 To 100 Do y := y*i;
```

Число шагов цикла N_Step = 100 – 2 + 1 = 99.

2. Найти максимальный делитель натурального числа k (за исключением самого k).

Алгоритм поиска максимального делителя можно построить следующим образом:

- вычислить целую часть от деления k пополам и присвоить переменной i значение целой части;
- построить цикл с перечислением **Downto**, **V_Start=i**, **V_Fin=1** и оператором цикла, вычисляющим остаток от деления k на i и проверяющим значение этого остатка. Как только остаток от деления будет первый раз равен 0, необходимо присвоить максимальному делителю (D) значение i и закончить выполнение цикла.

Введем переменные целого типа k, i и D. Построим последовательность операторов, реализующую предложенный алгоритм:

```
For i := k div 2 Downto 1 Do
If k mod i = 0 Then
Begin
      D := i; Goto 1
End;
1: ;
```


Оператор **While** — синтаксис:

```
<Оператор While > ::= While <Условие> Do <Тело цикла>
```

<Условие> — это логическое выражение, истинность которого проверяется в начале каждого шага цикла. Цикл будет выполняться до тех пор, пока значение логического выражения истинно (цикл типа Пока). Если на первом же шаге значение логического выражения ложно, тело цикла не выполняется ни разу.

В отличие от оператора **For**, в операторе **While** максимальное число шагов цикла заранее не известно. Оператор предусматривает изменение значения переменных, входящих в логическое выражение, внутри тела цикла.

Рассмотрим пример 2 предыдущего пункта. Решение этой задачи предполагает заранее неизвестное число шагов, которое понадобится для нахождения максимального делителя, поэтому использование в алгоритме оператора **While** более оправданно.

```
D:=0; i := k div 2;
While D=0 do
begin if k mod i = 0 then D := i;
      i := i-1
end;
```

Оператор **While** будет выполняться до тех пор, пока D=0, т. е. пока не найден ни один делитель. Первый же найденный делитель будет максимальным, значение его занесется в переменную D, которая при этом станет отлична от нуля и сделает ложным логическое выражение в заголовке цикла.

Оператор **Repeat** — синтаксис:

```
<Оператор Repeat> ::= Repeat <Тело цикла>
     Until <Условие>
```

Оператор **Repeat** отличается от остальных двух операторов цикла тем, что проверка условия продолжения цикла стоит после тела цикла. Это обеспечивает выполнение тела цикла хотя бы один раз.

Служебные слова **Repeat** и **Until** играют роль операторных скобок, поэтому тело цикла, состоящее из нескольких операторов, не нужно оформлять как составной оператор (как это необходимо делать в предыдущих операторах цикла).

Тело цикла выполняется до тех пор, пока логическое выражение, формирующее условие, не станет равным истинному значению (цикл типа Пока).

Рассмотрим следующий пример. Вычислить сумму ряда величин $\frac{1}{n^x}$ от $n = 1$ до тех пор, пока очередной член ряда не будет меньше 10^{-6}.

```
Readln(X)
S:=0;
N:=1;
SN:=1;
Repeat S:=S+SN;
     Inc(N);   // Увеличение N на единицу (N:=N+1);
     SN := 1/Power(N,X);
               // Вычисление значения очередного члена ряда
Until SN<= Power(10,-6);
```

В этом цикле сначала происходит приращение суммы, а затем вычисление значения очередного члена ряда. Это обусловлено тем, что первый же член ряда, значение которого будет меньше, чем 10^{-6}, не должен быть добавлен в сумму.

Операторы управления циклом **Break** и **Continue** появляются в версиях Turbo Pascal 7.0 и Object Pascal.

Оператор **Break** предназначен для осуществления принудительного досрочного выхода из цикла. Например, цикл поиска максимального делителя D числа k может выглядеть следующим образом:

```
For i := k div 2 Downto 1 Do
If k mod i = 0 Then
Begin
     D := i;  //принудительный выход из цикла после
              // нахождения первого значения D
     Break
End;
```

Оператор **Continue** принудительно вызывает начало новой итерации цикла, даже если предыдущая еще не закончена. Цикл поиска максимального нечетного делителя числа k может быть таким:

```
For i := k div 2 Downto 1 Do
Begin If i mod 2 = 0 Then Continue;
```



```
              // Переход на следующую итерацию,
              // если i четное
     If k mod i = 0 Then
     Begin D := i;
           Break
     End;
End;
```

Составной оператор. Составные операторы задают порядок выполнения операторов, являющихся их элементами. Они должны выполняться в том порядке, в котором они записаны.

Составные операторы обрабатываются, как один оператор, что имеет решающее значение там, где синтаксис языка допускает использование только одного оператора (например, в условном операторе **If..Then..Else**). Операторы заключаются в ограничители (операторные скобки) **Begin** и **End** и отделяются друг от друга точками с запятой.

Синтаксис:

```
<Составной оператор> ::= Begin
     <Последовательность операторов> End;
<Последовательность операторов> ::=<Оператор>|
     <Последовательность операторов>;<Оператор>
```

Пример составного оператора:

```
Begin
     Z : = X;
     X : = 2*Y;
     Y : = Z;
End;
```

3.4. Структурированные типы данных

Структурированные типы описывают наборы однотипных или разнотипных данных, с которыми программа должна работать как с одной именованной переменной.

Например, если необходимо вводить и обрабатывать данные о средней температуре за каждые сутки 2007 г., удобнее не объявлять переменную для каждого отдельного значения, а хранить все данные под одним именем, т. е. объявить одну структуру из 365 переменных типа Real, поименовать ее, а к каждому отдельному значению иметь доступ по порядковому номеру.

Рассмотрим другой пример. Пусть необходимо вводить и обрабатывать следующие данные о студентах: фамилии, имена, отчества, даты рождения, годы поступления в вуз, номера студенческих билетов и т. п. В этом случае создается структура данных типа `Student`, в которую объединяются все нужные (разнотипные) сведения о студенте.

В языке Pascal определены следующие стандартные структурированные типы данных:

- массив;
- строка;
- запись;
- множество.

Массив

Массив представляет собой упорядоченную структуру однотипных данных, которые называются элементами массива. Структура данных типа `Массив` подходит, например, для решения задачи обработки среднесуточных температур, описанной выше.

Доступ к каждому отдельному элементу массива осуществляется с помощью индекса — в общем случае порядкового номера элемента в массиве.

Массивы могут быть как одномерными (адрес каждого элемента определяется значением одного индекса), так и многомерными (адрес каждого элемента определяется значением нескольких индексов).

Общий синтаксис объявления типа «*n*-мерный массив»:

```
<Имя>=Array
      [<Тип индекса-1>,<Тип индекса-2>,...,<Тип индекса-N>]
      Of <Тип элемента>
```

Из приведенного описания видно, что до объявления массива необходимо объявить типы индексов и элемента.

Индексы массива должны принадлежать типу-диапазону, т. е. принимать множество последовательных значений порядкового типа.

Элемент массива может принадлежать любому типу данных, в том числе и типу массив.

Рассмотрим следующие примеры:

```
Type
     Real_array = Array [1..365] Of Real;
```

Тип `Real_array` (массив из 365 вещественных чисел) можно использовать при решении задачи о среднесуточной температуре, введя, например, переменную `Temperature` типа `Real_array`:

```
Var
     Temperature : Real_array;
```

Доступ к отдельному элементу массива осуществляется через указание имени массива и значения индекса, заключенного в квадратные скобки:

```
Temperature[5] := 5.6;
```

(среднесуточной температуре за 5-й день года присваивается значение 5.6).

Значение индекса может быть задано выражением соответствующего типа:

```
Temperature[n*7] := 10;
```

В зависимости от значения переменной `n` (предположим, что `n` меняется от 1 до 52) значение 10 будет занесено в 7-, 14-, 21-, ..., 364-й элементы массива.

Рассмотрим фрагмент программы, позволяющий вычислить среднесуточную температуру за год.

Введем переменные `Sum_Temp` типа `Real` (для суммирования всех значений элементов массива), `Mid_Temp` типа `Real` (для вычисления средней температуры за год) и переменную-параметр цикла `i` типа `Integer`.

```
Sum_Temp := 0;          // первоначальное обнуление
                        // переменной
For i := 1 To 365 Do
      Sum_Temp := Sum_Temp + Temperature[i];
                        // цикл подсчета суммарной
                        // температуры
Mid_Temp := Sum_Temp/365;
                        // вычисление среднего значения
```

Для решения подобной задачи, возможно, более наглядным будет введение двумерного массива вещественных чисел, где первый индекс будет меняться от 1 до 12 (в соответствии с количеством месяцев в году), а второй индекс — от 1 до 31 (в соответствии с максимальным количеством дней в месяце):

```
Type
     Real_array = Array [1..12, 1..31] Of Real;
```

В этом случае общее количество элементов в массиве — 12 · 31 = 372, а доступ к отдельному элементу осуществляется через задание двух индексов:

`Temperature[2,7]` — значение среднесуточной температуры 7 февраля.

Для вычисления средней температуры за год необходимо объявить две переменные-параметры цикла — `i` и `j`, а фрагмент программы будет иметь следующий вид:

```
Sum_Temp := 0;       //первоначальное обнуление
                     //переменной
For i := 1 To 12 Do
     For j := 1 To 31 Do
     Sum_Temp := Sum_Temp + Temperature[i,j];
                     // вложенные циклы подсчета
                     // суммарной температуры
Mid_Temp := Sum_Temp/(12*31);
                     // вычисление среднего значения
```

Задание значений массиву-константе. При определении массива-константы значения элементов массива указываются в круглых скобках и разделяются запятыми. Если массив многомерный, то внешние скобки соответствуют самому левому индексу, вложенные в них — следующему, и т. д.

Например, для массивов, типы которых определяется предложениями:

```
Type
     Vect_array = Array [1..7] Of Integer;
     Matr_array = Array [1..3, 1..4] Of Integer;
```

могут быть заданы константы `C1` и `C2`.

```
Const
     C1: Vect_array = (1, 3, 5, 7, 9, 11, 13);
     C2: Matr_array =      ((1, 3, 5, 7),(2, 4, 6, 8),
                           (9, 11, 13, 15));
```


Константа `C2` соответствует следующей матрице:

$$C2 = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ 9 & 11 & 13 & 15 \end{pmatrix}.$$

Операции над массивами. Для одномерных массивов символов можно использовать *операции сравнения*, даже если это массивы не идентичных типов и имеют различный размер, например, для объявленных следующим образом массивов:

```
Var
     A : Array [1..15] Of Char;
     B : Array [1..10] Of Char;
```

можно написать условный оператор вида:

```
If A>B
Then
     Writeln(A)
Else
     Writeln(B);
```

Одному массиву можно присвоить значение другого, но только если они идентичных типов, например, если заданы массивы:

```
Var
     A1, A2 : Array [1..15] Of Real;
     B : Array [1..15] Of Real;
```

то оператор присваивания допустим только между массивами `A1` и `A2` (`A1 := A2`), даже несмотря на то, что размеры и типы элементов совпадают у всех трех массивов.

Стандартные функции. Для массивов целых и вещественных чисел в современных реализациях языка определены некоторые стандартные функции (табл. 3.12).

Таблица 3.12. **Стандартные функцин над массивами ЯП Pascal**

Функция	Описание	Тип элементов массива	Тип результата
`MaxIntValue(Data)`	Возвращает значение максимального элемента	Целые числа	Целый
`MaxValue(Data)`	Возвращает значение максимального элемента	Вещественные числа	Вещественный

Окончание табл. 3.12

Функция	Описание	Тип элементов массива	Тип результата
Mean(Data)	Возвращает среднее арифметическое элементов массива	Вещественных чисел	Вещественный
MinIntValue(Data)	Возвращает значение минимального элемента	Целые числа	Целый
MinValue(Data)	Возвращает значение минимального элемента	Вещественные числа	Вещественный
Sum(Data)	Возвращает сумму элементов массива	Вещественные числа	Вещественный
SumInt(Data)	Возвращает сумму элементов массива	Целые числа	Целый
SumOfSquares	Возвращает сумму квадратов элементов массива	Вещественные числа	Вещественный

Строка символов

Тип Строка Символов **(String)** определяет последовательность символов произвольной длины, записанную в одной строке программы и заключенную в одиночные кавычки (апострофы). Строку можно рассматривать как массив символов, однако, в связи с некоторыми особенностями использования строк по сравнению со стандартными массивами, символьный массив выделен в отдельный (строковый) тип данных.

Строка символов, ничего не содержащая между апострофами, называется пустой строкой. Два последовательных апострофа в строке символов обозначают один символ-апостроф.

К символам в строке можно иметь доступ как к компонентам массива, например для объявленной строки MyString:

```
var
    MyString : String;
```

можно программировать следующие действия:

```
MyString[1]:='H'; MyString[2]:='E'; MyString[3]:='L';
MyString[4]:='L'; MyString[5]:='O';
```


Эта последовательность действий будет аналогична оператору

```
MyString:= 'HELLO',
```

если в атрибуте длины строки в первом случае установить значение 5.

Атрибут длины строки содержится в символе с порядковым номером 0 и представляет собой размер строки, числовое значение которого определяется как Ord(string[0]).

В стандарте языка строковый тип имеет фиксированный или динамический атрибут длины, но в любом случае длина строки не может превышать 255 символов.

Фиксированный атрибут длины задается в квадратных скобках после слова **String** при объявлении типа. Например, строка, объявленная как

```
Var
    MyString : String[20];
```

может иметь длину не более 20 символов.

Строковый тип, объявленный без атрибута длины, имеет установленный по умолчанию атрибут длины, равный 255. Текущее значение атрибута длины можно получить с помощью стандартной функции Length.

В качестве расширения стандарта языка в современных его реализациях разрешено вставлять в строку символов управляющие коды. Символ «#» с целой константой без знака в диапазоне от 0 до 255 обозначает соответствующий этому значению символ в коде ASCII. Между символом «#» и целой константой не должно быть никаких разделителей.

Аналогично, если несколько управляющих символов входят в строку символов, то между ними не должно быть разделителей.

Приведем несколько примеров строк символов:

```
'Object Pascal'
'You''ll see!!!'
#13#10'Line 1'#13'Line 2'#7#7'The End!'#7#7
```

Длинные строки. В целях преодоления ограничений на размер стандартных строк в 32-разрядных реализациях языка (в визуальной среде разработки приложений Delphi) введена поддержка длинных строк. В табл. 3.13 приведены типы строк, которые могут быть объявлены в Object Pascal.

Длинные строки ограничиваются нулевым терминатором, т. е. завершаются байтом с нулевым значением, что обеспечивает их совместимость со строками языка C.

Служебное слово **String** является общим при объявлении и коротких, и длинных строк. Например, объявление

```
Var
    S: String;
```

создает переменную типа AnsiString, если компилятор поддерживает размещение длинных строк, и переменную типа ShortString — в противном случае.

Таблица 3.13. **Типы строк**

Тип	Максимальная длина	Описание
ShortString	255 символов	Соответствует стандартному типу String. Каждый элемент строки имеет стандартный символьный тип
AnsiString	$\approx 2^{31}$ символов	«Длинная» строка переменной длины. Память выделяется динамически. Каждый элемент строки имеет стандартный символьный тип
WideString	$\approx 2^{30}$ символов	«Длинная» строка переменной длины. Память выделяется динамически. Каждый элемент строки представлен символом стандарта UniCode

Константы строкового типа. Объявление типизированной константы строкового типа содержит максимальную длину строки и ее начальное значение:

```
Const
    HeadText : String [7] = 'Section';
    Newline  : String [2] = #13#10;
    TrueQue  : String [3] = `Yes';
```

Операции со строковыми типами. Pascal позволяет использовать знак операции «+» для объединения двух строковых операндов. Результатом операции S+T, где S и T имеют строковый тип, будет конкатенация S и T. Результат будет совместим с любым строковым типом (но не с символьным).

Любые два значения строковых данных можно сравнить, поскольку все значения строковых данных совместимы.

Операции отношения =, <>, <, >, <=, или >= применяются для сравнения строк в соответствии с порядком расширенного набора символов кода ASCII. Отношение между любыми двумя строковыми значениями устанавливается согласно отношению порядка между значениями символов в одинаковых позициях. Все операции отношения учитывают регистр.

В двух строках разной длины каждый символ более длинной строки, для которого нет соответствующего символа в более короткой строке, принимает значение «больше», например:

'xs' больше, чем 'x', но 'xxx' меньше, чем 'xy'.

Пустые строки могут быть равны только другим пустым строкам, и они являются строками с наименьшим значением. Значения символьного типа совместимы со значениями строкового типа, и при их сравнении символьное значение обрабатывается как строковое значение длиной 1.

Стандартные процедуры и функции для работы со строками. Для работы с переменными строкового типа определены стандартные процедуры и функции. Некоторые из них (наиболее часто используемые) приведены в табл. 3.14 (в столбце **Статус** значение F соответствует функции, P — процедуре).

Таблица 3.14. **Процедуры и функции над строками**

Имя	Статус	Назначение	Аргументы (параметры)	Результат
CompareStr()	F	Сравнение двух строк	S1, S2 — сравниваемые строки	< 0, если S1 < S2; = 0, если S1 = S2; > 0, если S1 > S2.
Concat()	F	Конкатенация (сцепление) двух или более строк	S1, S2, ..., Sn — сцепляемые строки	Строка, равная S1+S2+...+Sn
Copy()	F	Выделение подстроки	Строка S; целые значения Index и Count. Часть строки S, начиная с позиции Index, длиной Count	
Delete()	P	Удаление подстроки	Строка S; целые значения Index и Count	Новое значение строки S без фрагмента, начиная с позиции Index, длиной Count
Insert()	P	Добавление подстроки в строку	Строки Source и S, целое значение Index	Новое значение строки S после добавления в нее Source, начиная с позиции Index

Окончание табл. 3.14

Имя	Статус	Назначение	Аргументы (параметры)	Результат
`Length()`	F	Вычисление длины строки	Строка `S`	Целое значение длины строки
`LowerCase()`	F	Преобразование строки к нижнему регистру	Строка `S`	Новое значение строки `S` все символы строчные
`Pos()`	F	Вычисляет позицию начала подстроки в строке	Подстрока `Substr`, строка `S`	Ноль, если `Substr` не содержится в `S`; целое > 0 — позиция `Substr` в `S`
`SetLength()`	P	Назначает новую длину строки	Строка `S`, целое значение `NewLength`	Строка `S` с новым значением длины
`Str()`	P	Преобразует числовой тип в строковый	Целое или вещественное значение `X`, строка `S`	Строковая запись (в строке `S`) числа `X`
`StringOfChar()`	F	Формирует строку из последовательности символов	Символ `Ch`, целое значение `Count`	Строка символов `Ch` длиной `Count`
`Trim()`	F	Удаляет пробелы и управляющие символы в начале и в конце строки	Строка `S`	Новая строка
`TrimLeft()`	F	Удаляет пробелы и управляющие символы в начале строки	Строка `S`	Новая строка
`TrimRight()`	F	Удаляет пробелы и управляющие символы в конце строки	Строка `S`	Новая строка
`UpperCase()`	F	Преобразование строки к верхнему регистру	Строка `S`	Новое значение строки `S` — все символы строчные
`Val()`	P	Преобразование строкового типа в числовой	Строка `S`, целое или вещественное значение `V`, целое значение `Code`	Если `Code` = 0, то целое или вещественное значение `V`, соответствующее строковой записи `S`. Если `Code` > 0, то преобразование невозможно и значение `Code` указывает позицию ошибочного символа

Примеры использования строковых процедур и функций:

- выделение первого слова в предложении (разделитель слов — знак «пробел»):

```
S_Sentence := TrimLeft(S_Sentence);   //Удаление пробелов
                                      //в начале строки
i := Pos(' ', S_Sentence); // Определение позиции первого
                           // пробела в предложении S_Sentence
If i > 0 Then S_Word := Copy(S_Sentence, 1, i-1)
Else S_Word := S_Sentence;
                           // В строковой переменной S_Word
                           // - значение первого слова
```

- удаление из строки всех цифр:

```
L := Length(S_Sentence);   // Вычисление длины строки
i := 1;
While i <= L Do
If S_Sentence[i] In ['0'..'9'] Then
     Begin Delete(S_Sentence, i, 1);
     dec(L)                //Уменьшение длины строки
                           //на 1 в случае удаления цифры
     End
Else inc(i);
```

- подсчет количества букв 'W' в строке (независимо от регистра):

```
N_w := 0;                  // Обнуление счетчика букв
S_Sentence := LowerCase(S_Sentence);
                           // Преобразование строки к
                           // нижнему регистру
L := Length(S_Sentence); // Подсчет длины строки
For I := 1 To L Do
     if S_Sentence[i] = 'w' Then inc(N_w);
                           // Увеличение счетчика букв на 1
```

***Строка* `PChar`.** Для совместимости с другими языками программирования (например, C) и средой Windows в расширениях стандарта языка Turbo Pascal и Object Pascal введен еще один вид строк — строки, оканчивающиеся нулевым байтом (#0). Этим строкам соответствует стандартный тип `PChar`. Фактически тип `PChar` эквивалентен массиву символов (от 0 до N), где N — количество символов, не считая завершающего нулевого байта:

```
PChar = Array [0..N] Of Char;
```

В отличие от типа **String** символ с индексом 0 в **PChar**-строке является первым, а символ с индексом N — завершающим с кодом 0.

Для преобразования строчных типов в Object Pascal можно применять следующие действия над типами **String** и **PChar**, пусть объявлены переменные:

```
Var
     S_String: String;
     S_PChar: PChar;
```

Тогда справедливы следующие операторы присваивания:

```
S_String:= string(S_PChar);
S_PChar:= PChar(S_String);
```

В Object Pascal для работы с переменными типа **PChar** определены стандартные функции. Некоторые из них приведены в табл. 3.15.

Таблица 3.15. **Некоторые функции над переменнымн PChar**

Имя	Назначение	Аргументы (параметры)	Результат
StrCat()	Объединение двух строк (добавление к PChar-строке Dest PChar-строки Source)	PChar-строки Dest и Source	Новая PChar-строка
StrComp()	Сравнение двух PChar-строк	Str1, Str2 — PChar-строки	Целое число: <0, если Str1<Str2; =0, если Str1=Str2; >0, если Str1>Str2
StrCopy()	Копирование одной PChar-строки (Source) в другую (Dest)	PChar-строки Dest и Source	PChar-строка Dest
StrECopy()	Копирование одной PChar-строки (Source) в другую (Dest). Возвращается указатель на нулевой байт новой строки	PChar-строки Dest и Source	PChar-строка, начинающаяся с нулевого байта Dest
StrEnd()	Возвращается указатель на PChar-строку	PChar-строка Source	PChar-строка, начинающаяся с нулевого байта Source

Окончание табл. 3.15

Имя	Назначение	Аргументы (параметры)	Результат
StrLen()	Возвращается количество символов в PChar-строке (исключая нулевой байт)	PChar-строка Source	Целое число — количество символов в строке
StrLower()	Преобразование символов PChar-строки к нижнему регистру (все строчные)	PChar-строка Source	Преобразованная PChar-строка
StrMove()	Копирование заданного количества символов из одной PChar-строки (Source) в другую (Dest)	PChar-строки Dest и Source; количество символов Count	PChar-строка Dest
StrPCopy()	Копирование строки Source типа string в PChar-строку Dest	PChar-строка Dest, строка типа string Source	PChar-строка Dest
StrPos()	Возвращается указатель на начало PChar-строки Str2 внутри PChar-строки Str1	PChar-строки Str1, Str2	PChar-строка Str2 внутри Str1
StrRScan()	Возвращает указатель на последнее вхождение символа Chr в PChar-строку Str	PChar-строка Str, символ Chr	PChar-строка, начиная с последнего вхождения символа Chr в строку Str; nil — если вхождение не найдено
StrScan()	Возвращает указатель на первое вхождение символа Chr в PChar-строку Str	PChar-строка Str, символ Chr	PChar-строка, начиная с первого вхождения символа Chr в строку Str
StrUpper()	Преобразование символов PChar-строки к верхнему регистру (все прописные)	PChar-строка Source	Преобразованная PChar-строка

Примеры использования функций:

1. Рассмотрим фрагменты программы:

```
Var
   VPStr1, VPStr2: PChar; //Объявление переменных типа PChar
   S: String;
```

Следующий условный оператор проверяет вхождение PChar-строки CPascal в PChar-строку CTurbo и заносит значение в строку S:

```
If StrPos(VPStr1, VPStr2) <> nil Then
        S:='Подстрока найдена'
Else S:='Подстрока не найдена';
```

Если перед выполнением оператора VPStr1= 'Object Pascal ', а VPStr2='Pascal ', то в строку S занесется значение 'Подстрока найдена'.

2. Пример извлечения имени файла из полного пути.

```
Var             // Объявления переменных
     FileName, P: PChar;
     S: String;
```

Фрагмент программы:

```
P := StrRScan(FileName, '\');// Поиск последнего символа '\'
              // После выполнения оператора в переменной P
              // либо имя файла, либо nil
If P = nil Then
Begin
      P := StrRScan(FileName, ':');//Поиск последнего символа
                                   //':' После выполнения
                                   //оператора в переменной P
                                   //либо имя файла,либо nil
      If P = nil Then P := FileName;
End;
S := string(P);                    //Занесение в S имени файла
```

Запись

Тип Запись (**Record**) содержит установленное число компонентов или полей, которые могут быть различных типов. Объявление типа Запись задает для каждого поля идентификатор, который именует поле, и тип данных поля. Синтаксис:

```
<Запись> ::= Record <Список полей> End;
<Список полей> ::= <Поле>| <Список полей>;<Поле>
<Поле>::= <Список идентификаторов>:< Тип >
<Список идентификаторов> ::= <Идентификатор>|
     <Список идентификаторов>, < Идентификатор>
```


В соответствии с приведенным синтаксическим описанием поля в объявлении типа Запись отделяются друг от друга точкой с запятой, а при наличии в записи нескольких полей с одним и тем же типом данных идентификаторы этих полей могут быть описаны в одной строке и перечислены при этом через запятую. Количество полей записи не ограничено.

После объявления типа запись можно задать переменные или константы этого типа, например:

```
Type
TDateRec = Record
     Year: Integer;
     Month: 1 .. 12;
     Day: 1 .. 31;
End;
TPerson = Record
     FirstName, SecondName:string[20];
     Sex:(Man, Woman);
     Age:integer
End;
Var
     Person:TPerson;
     DateRec:TDateRec;
```

Доступ к полям записи осуществляется путем указания имени переменной (или константы) и имени поля, разделенным точкой, например, для переменных Person и DateRec, объявленных выше, операторы присваивания значений полям записываются следующим образом:

```
Person.FirstName:='Victor';
Person.Sex := Man;
Person.Age := 20;
DateRec.Year := 2002;
Daterec.Day := 21;
```

Объявление типа запись может иметь так называемую вариантную часть, воспринимаемую программой по-разному, в зависимости от текущего назначения. Физически вариантная часть представляет собой фрагмент памяти, который может рассматриваться в разных случаях, как разный набор полей. Вариантная часть в записи может быть только одна и в описании располагается в конце записи.

Синтаксис типа запись с вариантной частью следующий:

```
<Запись> ::= Record <Фиксированная часть>;
      <Вариантная часть> End;
<Фиксированная часть> ::= <Список полей>
<Вариантная часть> ::= <Заголовок вариантной части>
      <Список вариантов>
<Заголовок вариантной части> ::=
      Case <Идентификатор переменной выбора варианта> :
      <Перечисляемый тип > Of
<Список вариантов> ::= <Вариант>|
      <Список вариантов>;<Вариант>
<Вариант> ::= <Значение переменной выбора варианта>:
      (<Список полей>)
```

В случае записи с вариантной частью фиксированная часть может отсутствовать.

Переменная выбора варианта — это всегда переменная некоторого перечисляемого типа. Вариантная часть записи воспринимается как набор полей, соответствующий значению переменной выбора, поэтому доступ к информации может быть осуществлен более чем одним способом. При этом доступ к вариантным и фиксированным полям один и тот же.

Рассмотрим несколько примеров записей с вариантной частью:

```
Type
     TProfesion = (Student, Engeneer);
     TPerson = Record
     FirstName, LastName: String[20];
     BirthDay: DateRec;
Case Profesion: TProfesion Of
     Student: (Course: Integer;
     (Faculty : string[20]);
     Engineer: (Speciality: String[30])
End;
TFigure = (Rectangle, Triangle, Circle);
TColor = (Red, Green, Yellow);
TPolygon = Record
     X, Y: Real;
     Color: TColor;
Case Kind: TFigure Of
     Rectangle: (RHeight, RWidth: Real);
     Triangle: (TSize1, TSize2, Angle: Real);
     Circle: (Radius: Real);
End;
Var
     Student_Person, Engeneer_Person: TPerson;
     MyRect, MyTriangle: TPolygon;
```


В стандарте языка прежде чем использовать в программе один из вариантов, необходимо задать соответствующее значение переменной выбора варианта, например:

```
MyRect.Color := Red;
Kind := Rectangle;
MyRect.RHeight := 5.23;
```

В современных реализациях языка эту операцию делать не надо: нужный вариант однозначно определяется именем поля:

```
Student_Person.Course := 5;
Engeneer_Person.Speciality := 'Programmer';
MyTriangle.Angle := 1.0;
```

При объявлении типа записи с вариантной частью допускается не вводить специальную переменную выбора, а использовать стандартные перечисляемые типы и их возможные значения, например:

```
Var
     TConvert = Record
Case Integer Of
     1:(CPoint : longint);
     2:(Carray : array [1..4] of byte)
End;
```

Константы с типом **Запись**. Объявление константы с типом **Запись** содержит идентификатор и значение каждого поля, заключенные в скобки и разделенные точками с запятой. Синтаксис объявления константы следующий:

```
<Константа-запись> ::= <Идентификатор>:<Тип-запись> =
     (<Список значений полей>);
<Список значений полей> ::= <Значение поля>|
     <Список значений полей>;<Значение поля>
<Значение поля> ::= <Идентификатор поля>: <Значение>
```

Рассмотрим несколько примеров констант-записей:

```
Type
     TPoint = Record
     X, Y : Real;
End;
TVector = array [0..1] of TPoint;
     TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jly, Aug,
     Sep, Oct, Nov, Dec);
```

```
TDateRec = Record
     Day : 1..31;
     Month : TMonth;
     Year : 1950..2050;
End;
Const
     Origin : TPoint = (X : 0.0; Y : 0.0);
     Line   : TVector= ((X : -3.1; Y : 1.5),
     (X : 5.8; Y : 3.0));
     SomDay : TDateRec  = (Day: 2; Month: Dec;
     Year: 2002);
```

Поля должны указываться в том же порядке, как они следуют в объявлении типа запись. Если запись содержит вариантную часть, то можно указывать только поля одного выбранного варианта.

***Оператор над записями* (With).** В операциях над записями оператор **With** удобно использовать для краткого обращения к полям записи. В операторе **With** к полям одной или более конкретных переменных типа запись можно обращаться, используя только идентификаторы полей.

Оператор **With** имеет следующий синтаксис:

```
<Оператор With> ::= With <Идентификатор записи>
     Do <Оператор>;
```

Приведем пример оператора **With**:

```
With DateRec Do
     If Month = 12 Then
          Begin
               Month := 1;
               Year := Year + 1;
          End
     Else
          Month := Month + 1;
```

Это эквивалентно следующему условному оператору:

```
If DateRec.Month = 12 Then
     Begin
          DateRec.Month := 1;
          DateRec.Year := DateRec.Year + 1;
     End
Else
     DateRec.Month := DateRec.Month + 1;
```


В операторе **With** сначала проводится проверка каждого идентификатора переменной на соответствие полю указанной записи. Если идентификатор переменной совпадает с именем поля, то переменная всегда интерпретируется как поле записи.

Пусть объявлены следующие переменные:

```
Type
     TPoint = Record
     X, Y: Integer;
End;
Var
     X : TPoint;
     Y : Integer;
```

В этом случае как к X, так и к Y можно обращаться, как к переменной или как к полю записи. Например, оператор:

```
With X Do
     Begin
          X := 1;
          Y := 100;
     End;
```

эквивалентен последовательности операторов

```
X.X := 1;
X.Y := 100;
```

так как X и Y внутри оператора **With** воспринимаются как поля записи типа TPoint.

Множество

Диапазон значений типа Множество (Set) представляет собой множество всевозможных сочетаний объектов заданного порядкового типа. В этом случае заданный порядковый тип называется базовым типом.

Каждое возможное значение типа Множество является подмножеством возможных значений базового типа.

Переменная типа множество может принимать как все значения множества, так и ни одного. Синтаксис:

```
<Тип множество> ::= Set Of <Порядковый тип>;
```

Базовый тип не должен иметь более 256 возможных значений, и порядковые значения верхней и нижней границы базового типа не должны превышать диапазона от 0 до 255. В силу этого базовыми типами множества не могут быть типы `ShorInt`, `Integer`, `LongInt`, `Word`.

Любой множественный тип может принимать значение `[]`, которое называется пустым множеством.

Примеры множеств:

```
Type
     Number = Set Of '0'..'9';
     Digit = Set Of 0..9;
     Oper = Set Of (Plus, Minus, Multiply, Divide);
```

Тип множество можно ввести и непосредственно при объявлении переменных:

```
Var
     Digit :Set Of 0..9;
```

Значение переменной типа множество присваивается путем перечисления в квадратных скобках через запятую некоторых значений (или диапазонов значений) базового типа, например:

```
Digit := [1,3,5,7,9];
Digit := [0..5, 7, 8];
```

***Константы типа* Множество.** Объявление константы типа множества может содержать несколько элементов, заключенных в квадратные скобки и разделенных запятыми. Каждый элемент такой константы представляет собой константу или диапазон, состоящий из двух констант, разделенных двумя точками, например:

```
Type
     Digits  = Set Of 0..9;
     Letters  = Set Of 'A'..'Z';
Const
     EvenDigits : Digits = [0, 2, 4, 6, 8];
     Vowels     : Letters= ['A', 'E', 'I', 'O', 'U',
     'Y'];
     HexDigits  : Set Of '0'..'z' = ['0'..'9', 'A'..'F',
     'a'..'f'];
```

Операции над множествами. Над множествами (совместимых типов) определены как теоретико-множественные (алгебра множеств), так и логические (сравнение множеств) операции (табл. 3.16).

Таблица 3.16. **Операции над переменными типа множество**

Обозначение	Наименование	Комментарий
	Теоретико-множественные операции	
+	Объединение ($A \cup B$)	Значение C принадлежит A+B только в том случае, если C принадлежит A или B
-	Разность ($A \setminus B$)	C принадлежит A-B , если C принадлежит A, но не принадлежит B
*	Пересечение ($A \cap B$)	C принадлежит A*B, если C принадлежит как множеству A, так и множеству B
	Операции сравнения (логические)	
=	Равенство	Выражение A = B истинно только тогда, когда A и B содержат одни и те же элементы
<>	Неравенство	Выражение A <> B истинно только тогда, когда хотя бы один элемент множества A не содержатся во множестве B или наоборот
<=	Вхождение множества ($A \subset B$)	Выражение A <= B истинно, если каждый элемент множества A является также и элементом множества B
>=	Вхождение множества ($B \subset A$)	Выражение A >= B истинно, когда каждый элемент множества B является также и элементом множества A
In	Принадлежность множеству ($x \in A$)	Выражение x In A истинно, когда значение элемента порядкового типа является элементом типа множества, иначе выражение ложно

Тип данных Файл

В практике программирования часто встречаются задачи, при решении которых можно хранить обрабатываемые данные на внешнем носителе. В этом случае данные оформляются в виде файлов. Под файлом понимают любой логически непрерывный набор данных, размещенный на внешнем запоминающем устройстве (например, на магнитном диске). Так, исходные данные для работы программы могут быть организованы в виде

файла или совокупности файлов. Файлом может быть и результат работы программы.

Поддержка работы с внешними файлами внутри программы в языке Pascal осуществляется с помощью специализированных переменных, которым присваивается тип данных Файл.

Тип данных Файл описывает файл как линейную последовательность однотипных агрегатов (записей), размещенных на внешнем запоминающем устройстве, которые могут иметь любой тип данных за исключением типа файл или структурного типа, содержащего компоненту с типом файл. В отличие от массива длина файла, т. е. количество компонентов (записей), не задается, а место элемента не определяется индексом. Синтаксис объявления типа данных файл следующий:

```
<Тип файл> ::= <Идентификатор типа> = File Of
      <Тип компоненты>;
```

Если слово **Of** и тип компоненты опущены, то объявляется нетипизированный файл. Нетипизированные файлы используются для доступа к любому файлу на внешнем устройстве независимо от его внутреннего формата.

Стандартный файловый тип Text определяет файл, содержащий символы, объединенные в строки. Следует иметь в виду, что тип Text не идентичен типу **File Of Char**.

Приведем примеры объявлений файловых типов:

```
Type
    FileNumber = File Of Longint;// Файл длинных целых чисел
    FileDigit = File Of '0'..'9';// Файл символов-цифр
TPoint = record
    X,Y:real
End;
    FilePoint = File Of TPoint;  // Файл записей TPoint
```

Объявив файловый тип, можно определить переменные файлового типа:

```
Var
    File1: FileNumber;
    File2: FilePoint;
    File3: Text;       // Файл стандартного типа Text
    File4: File;       // Переменная нетипизированного файла
```

Переменные файлового типа имеют специфическое применение. Над ними нельзя выполнять никаких операций (например, присваивать значение, сравнивать), они используются лишь в качестве параметров специальных процедур работы с файлами.

3.5. Динамические данные

Рассмотренные выше типы данных описывали фрагменты оперативной памяти, которые выделялись по мере объявления в программе переменных того или иного типа. Такие переменные называются статическими, после объявления в разделе переменных под них отводится столько байтов оперативной памяти, сколько требует тип переменной. Размерами выделенной таким образом памяти уже нельзя управлять из программы, а можно лишь присваивать таким переменным некоторые значения.

Такое статическое объявление не всегда удобно при решении многих практических задач, например, когда речь идет об обработке заранее неизвестного количества однотипных данных, или о построении связанных структур, таких как списки, деревья и т. п. В этих случаях выделение максимальных размеров памяти приводит к неэффективной работе программы, а порой нельзя заранее предугадать, какое максимальное количество памяти потребуется для размещения данных.

В языке Pascal предусмотрена возможность размещения переменных в памяти по ходу выполнения программы. Для таких целей выделена особая область оперативной памяти, которая называется динамической, а переменные, размещаемые в ней, называются динамическими переменными. Таким образом, процессе обработки данные можно размещать и (после обработки) удалять из оперативной памяти.

Такая языковая возможность связана с наличием особых типов данных — указателей.

Тип Указатель

Тип данных Указатель задает множество значений, которые соответствуют адресам переменных определенного типа, называемого базовым типом. Синтаксис:

```
<Тип указатель> ::= ^<Базовый тип>;
<Базовый тип> ::= <Идентификатор типа>;
```

Например:

```
Type
     TCoord = Record
     X, Y: Real
End;
PTCoord = ^TCoord;
Var
     P1, P2, P3 : PTCoord;
     Point : TCoord;
     P : Pointer;
```

При объявлении переменной типа PTCoord резервируется память под физический адрес переменной типа TCoord (но не под сам тип TCoord).

Присвоить значение переменной типа указатель можно двумя путями:

- разместить в памяти новую (динамическую) переменную с помощью стандартной процедуры New с параметром типа указатель, в который после выполнения процедуры будет занесено значение адреса размещения;
- присвоить значение адреса размещения объявленной переменной базового типа (т. е. статической переменной), используя знак операции «@".

Процедура New отводит новую область в динамической памяти для переменной и сохраняет адрес этой области в переменной типа указатель.

Знак операции «@» устанавливает переменную типа указатель на область памяти, содержащую объявленную переменную.

Зарезервированное слово nil обозначает константу с пустым значением указателя.

Встроенный тип **Pointer** обозначает нетипизированный указатель, т. е. указатель, который не указывает ни на какой определенный тип. Как и значение, обозначаемое словом nil, значения типа **Pointer** совместимы со всеми другими типами указателей.

Для приведенных выше объявлений справедливы следующие операторы:

```
New(P1);
P2 := @Point;
New(P);
P3 := P;
```


В ЯП Pascal определен ряд стандартных процедур и функций для переменных типа указатель (табл. 3.17).

Таблица 3.17. **Перечень процедур и функций над указателями**

Название	Статус	Назначение	Параметры	Результат
Addr(X)	F	Возвращает адрес статической переменной	X — параметр любого типа (включая процедурный)	Указатель типа Pointer
Ptr(Address)	F	Преобразует значение целого тила в указатель	Address — параметр типа Integer	Указатель типа Pointer
New(P)	P	Размещает в памяти динамическую переменную типа, базового для параметра P. Размер выделяемой памяти совпадает с размером базового типа	P — параметр типа указатель	В параметр P заносится значение адреса переменной базового типа
Dispose(P)	P	Освобождение памяти, на которую указывает параметр. Размер освобождаемой памяти совпадает с размером базового типа	P – параметр типа указатель	После выполнения процедуры значение P не определено
GetMem(P,Size)	P	Выделяет память заданного размера под переменную	P — параметр типа указатель (включая тип Pointer); Size — параметр типа Integer	В параметр P заносится значение адреса переменной
FreeMem(P[, Size])	P	Освобождает память заданного размера, занятую переменной, на которую указывает параметр. Если размер не указан, освобождается память, предварительно выделенная с помощью процедуры GetMem	P — параметр тила указатель (включая тип Pointer); Size — параметр типа Integer	После выполнения процедуры значение P не определено

Указатели, определяюшие адреса переменных разного типа, сами являются переменными разного типа, и для них недопустимо использование оператора присваивания. Исключение составляет тип **Pointer**, который совместим по присваиванию с любым другим типом указатель.

К указателям одного типа можно применять операции сравнения = и <>.

Чтобы получить непосредственно значение переменной, адрес которой содержится в указателе, необходимо после имени указателя поставить знак «^», например:

```
Point := P1^;
P2^ := Point;
```

Константы с типом **Указатель.** Объявление константы типа Указатель обычно содержит знак операции «@" и идентификатор константного значения некоторого типа, например:

```
Type
     TDirection = (Left, Right, Up, Down);
     TNodePtr = ^Node;
     TNode = Record
          Value : TDirection;
          Next : TNodePtr;
End;
Const
     N1: TNode = (Value: Down;Next: nil);
     N2: TNode = (Value: Up; Next: @N1);
     N3: TNode = (Value: Right;Next: @N2);
     N4: TNode = (Value: Left;Next: @N3);
     DirectionTable: TNodePtr = @N4;
```

В приведенном примере необходимо обратить внимание на то, что синтаксис языка разрешает объявлять тип-указатель перед объявлением базового типа, например, в случае, когда речь идет о формировании последовательностей структур данных типа Запись со ссылками друг на друга.

3.6. Процедуры и функции

При разработке программ на алгоритмических языках широко используется понятие подпрограммы. Подпрограмма — это средство, позволяющее многократно использовать в разных местах основной программы один раз описанный фрагмент алгоритма.

Структура подпрограммы аналогична структуре программы и может содержать все разделы объявлений, принятые в языке.

Объявления (типы, переменные, константы), использующиеся любой подпрограммой, относятся к одной из двух категорий — категории локальных объявлений и категории глобальных объявлений. Локальные объявления принадлежат процедуре, описаны внутри нее и могут использоваться только ею. Глобальные объявления принадлежат программе в целом и доступны как самой программе, так и всем ее подпрограммам. Обмен данными между основной программой и ее подпрограммами обычно осуществляется посредством глобальных переменных.

Если имя глобального объявления совпадает с именем локального, то внутри подпрограммы объявление интерпретируется как локальное, и все изменения, вносимые, например, в значение такой переменной, актуальны только в рамках подпрограммы.

В языке Pascal существуют две разновидности подпрограмм — процедуры и функции. Каждое объявление процедуры или функции содержит обязательный заголовок, за которым следуют разделы локальных объявлений (аналогичных разделам объявлений программы) и составной оператор (блок), реализующий алгоритм подпрограммы.

Вызов процедуры на исполнение активизируется с помощью оператора процедуры. Функция активизируется при вычислении выражения, содержащего вызов функции, а возвращаемое функцией значение подставляется в это выражение.

Объявление процедур

Синтаксис объявления процедуры следующий:

```
<Объявление процедуры > ::= <Заголовок процедуры>;
     <Тело процедуры>;
<Заголовок процедуры> ::= Procedure <Идентификатор>
     <Список формальных параметров>
<Тело процедуры> ::= <Раздел локальных объявлений>
     <Составной оператор>
```

В заголовке процедуры указывается имя процедуры и описывается список формальных параметров (если он присутствует).

За заголовком может следовать раздел локальных объявлений (меток, типов, констант, переменных, вложенных процедур и функций). Раздел локальных объявлений может отсутствовать в

том случае, если процедура использует только глобальные объявления.

Запуск процедуры на исполнение осуществляется с помощью оператора процедуры, в котором содержатся имя процедуры и фактические параметры.

Последовательность операторов, реализующих алгоритм процедуры, записывается внутри составного оператора (блока) процедуры. Если в каком-либо операторе внутри блока процедуры используется идентификатор самой процедуры, то процедура будет выполняться рекурсивно, т. е. при выполнении будет обращаться сама к себе.

Приведем пример объявления процедуры:

```
      // Преобразование целого числа в строку
      // восьмеричного представления
Procedure OctString(Nmb: Integer; Var S: string);
      // Заголовок процедуры со списком формальных параметров:
      // Nmb - исходное целое число; S - строка для записи
      // результата преобразования
Var
      P: Integer; // Объявление локальной переменной P
Begin
      P := Abs(Nmb);
      S := '';
      Repeat
            S := S + chr(P mod 8);
            P := P div 8;
      Until P = 0;
      If Nmb < 0 Then S := '-' + S;
End;
```

В дальнейшем для вызова процедуры из основной программы или подпрограммы необходимо записать оператор процедуры со списком фактических параметров, которые должны совпадать по количеству и типам с формальными параметрами процедуры. Например, в результате выполнения фрагмента программы:

```
OctString(InpNmb,ResultString);
ResultString := 'Число '+str(InpNmb)+
' в восьмеричной с.с. равно '+ResultString;
```

в переменной `ResultString` типа `String` формируется запись старого и нового представления целого числа `InpNmb`.

Объявления функций

Функция — это подпрограмма, вычисляющая и возвращающая некоторое значение. Оператор функции может быть использован в качестве операнда при построении выражения.

Синтаксис объявления функции следующий:

```
<Объявление функции> ::= <Заголовок функций >;<Тело функции>;
<Заголовок функции> ::= Function <Идентификатор>
      <Список формальных параметров>: <Тип результата >
<Тип результата> ::= <Идентификатор типа>
<Тело функции> ::= <Раздел локальных объявлений>
      <Составной оператор>
```

Основное отличие заголовка функции от заголовка процедуры (помимо используемого для объявления служебного слова) в том, что заголовок функции дополнительно содержит указание на тип возвращаемого функцией значения — тип результата.

Оператор функции при ее вызове обычно стоит либо в правой части оператора присваивания, либо входит на правах операнда в выражение, либо указывается в качестве фактического параметра при вызове другой подпрограммы. Вызов функции содержит идентификатор функции и список фактических параметров, совпадающий по размеру и типам со списком формальных параметров. После выполнения тела функции возвращается значение, тип которого совпадает с типом результата функции.

Операторная часть тела функции содержит операторы, реализующие алгоритм получения результата объявленного типа, при этом в операторной части должен находиться по крайней мере один оператор присваивания, в котором в левой части стоит идентификатор функции. Результатом выполнения функции будет последнее значение, присваиваемое идентификатору функции. Если такого оператора присваивания нет или он не выполняется, то возвращаемое функцией значение не будет определено.

Если идентификатор функции используется для вызова функции внутри операторной части тела функции, то функция выполняется рекурсивно.

Приведем пример объявления функции:

```
      // Функция вычисления суммы квадратов первых
      // N чисел натурального ряда
```

```
Function SumSqr(N: Integer): Integer;
Var
     S, i: Integer;
Begin
     S := 1;
     For i := 2 To N Do
          S := S+i*i;
     SumSqr := S;       // Присваивание значения результату
                        //функции
End;
```

Для вызова функции из основной программы или из другой подпрограммы (например, при вычислении значения выражения) необходимо в выражении указать оператор функции со списком фактических параметров:

```
LineLen := Log10(X)/SumSqr(I);
```

Здесь `Log10(X)` — вызов стандартной функции вычисления десятичного логарифма фактического параметра `X`;

`SumSqr(I)` — вызов функции вычисления суммы квадратов с фактическим параметром `I`.

Переменная `Result` в функции

Реализация языка в версии Object Pascal облегчает программирование функций путем автоматического объявления в каждой из них локальной переменной `Result`. Эта переменная имеет тот же тип, что и результат функции. Присваивание значения переменной Result аналогично определению значения функции. Преимущество использования этой переменной при вычислении значения функции в том, что локальная переменная `Result` может стоять в правой части оператора присваивания и не вызывает при этом рекурсивного вычисления функции (в отличие от идентификатора функции, появление которого в вычисляемом выражении означает рекурсивный вызов функции).

Например, представленная выше функция вычисления суммы квадратов натурального ряда может быть преобразована следующим образом:

```
Function SumSqr(N: Integer): Integer;
Var
     i: Integer;
```



```
Begin
     Result := 1;
     For i := 2 To N Do
          Result := Result +i*i; // После выхода из
               //цикла в переменной Result - значение функции
End;
```

Заметим, что уменьшилось количество локальных переменных, необходимых для реализации алгоритма, и сам алгоритм стал короче на один оператор.

Формальные и фактические параметры

Объявление процедуры или функции содержит список формальных параметров. Каждый параметр из списка формальных параметров является локальным по отношению к процедуре или функции, для которой он объявлен. Это означает, что глобальные переменные, имена которых совпадают с именами формальных параметров, становятся недоступными для использования в процедуре или функции.

Отдельные объявления параметров в списке разделяются точкой с запятой. Синтаксис списка формальных параметров следующий:

```
<Список формальных параметров> ::= <Объявление параметра>|
     <Список формальных параметров>; <Объявление параметра>
```

Все формальные параметры можно разбить на две категории:

- параметры, вызываемые подпрограммой по своему значению (т. е. параметры, которые передают в подпрограмму свое значение и не меняются в результате выполнения подпрограммы);
- параметры, вызываемые подпрограммой по наименованию (т. е. параметры, которые становятся доступными для изменения внутри подпрограммы).

Главное различие этих двух категорий — в механизме передачи параметров в подпрограмму. При вызове параметра по значению происходит копирование памяти, занимаемой параметром, в стек и использование в дальнейшем в операторах подпрограммы локальной копии параметра. Основное значение параметра

(глобальное по отношению к подпрограмме) при этом остается без изменения. Следует отметить, что использование такого механизма при передаче, например, массивов большой длины может отрицательно влиять на быстродействие программы и заполняет стек лишней информацией.

При вызове параметра по наименованию в процедуру передается адрес памяти (глобальной по отношению к подпрограмме), в которой размещено значение параметра, и в качестве локальной переменной выступает ссылка на глобальное размещение параметра, обеспечивающая доступ к самому значению.

Объявление формального параметра обязательно содержит имя (идентификатор) параметра и, как правило, его тип, отделяющийся от имени двоеточием. Несколько однотипных параметров могут объединяться в одно объявление. При этом их имена перечисляются через запятую.

При обращении к подпрограмме формальные параметры заменяются на соответствующие по типу и категории фактические параметры вызывающей программы или подпрограммы.

Параметры-значения. Формальные параметры-значения относятся к первой из перечисленных выше категорий и действуют как переменные, локальные по отношению к процедуре или функции. Изменения формальных параметров-значений не влияют на значения соответствующих фактических параметров. Синтаксис объявления параметра-значения следующий:

```
<Объявление параметра-значения> ::= <Список идентификаторов>:
      <Тип параметра>
<Тип параметра> ::= <Идентификатор типа>
<Список идентификаторов> ::= <Идентификатор>|
      <Список идентификаторов>,<Идентификатор>
```

Фактический параметр, соответствующий параметру-значению в операторе процедуры или вызове функции, должен быть выражением, а его значение не может быть файлового типа. Фактический параметр должен быть совместим по присваиванию с типом формального параметра-значения.

Рассмотрим пример объявления функции с параметром-значением:

```
// Функция вычисления количества запятых в строке
Function NmbComma( S: String ): Integer;
Var i, L, Nmb :Integer;
```



```
Begin
     Nmb := 0;
     L := Length(S);
     For i := 1 To L Do
          If S[i] = ',' Then inc(Nmb);
     NmbComma := Nmb;
End;
```

Параметры-переменные. Формальные параметры-переменные относятся ко второй категории параметров и служат для модификации внутри подпрограммы значений соответствующих фактических параметров. Формальный параметр-переменная представляет фактическую переменную во время выполнения процедуры или функции, поэтому все изменения значения формального параметра отражаются на фактическом параметре. Синтаксис объявления параметров-переменных следующий:

```
<Объявление параметров-переменных> ::=
     Var <Список идентификаторов>:<Тип параметра>|
     Var <Список идентификаторов>
```

Внутри подпрограммы любое упоминание формального параметра-переменной обеспечивает доступ к самому фактическому параметру. Тип фактического параметра должен быть тождественен типу формального параметра-переменной (это ограничение можно обойти через использование параметров-переменных без типа). Файловые типы могут передаваться только как параметры-переменные.

Рассмотрим пример объявления процедуры с параметром-переменной:

```
//Процедура замены запятых на точки с запятой в строке
Procedure NmbComma(Var S: String);
Var i, L:Integer;
Begin
     L := Length(S);
     For i := 1 To L Do
     If S[i] = ',' Then S[i] = ';';
End;
```

В результате применения этой процедуры к строке `S_Sentence`:

```
NmbComma(S_Sentence);
```

будет изменено содержимое строки. Если до вызова процедуры в строке находилось значение 'лимон, апельсин, банан', то после вызова процедуры значение S_Sentence будет равно 'лимон; апельсин; банан'.

Как видно из синтаксической формулы, объявление параметров-переменных может не сопровождаться указанием типа. Когда формальный параметр является параметром-переменной без типа, соответствующий фактический параметр может быть переменной любого типа, а ответственность за правильность использования параметра ложится при этом на программиста.

Внутри процедуры или функции такой параметр-переменная не имеет типа, то есть он не совместим с переменными всех других типов до тех пор, пока ему не присвоен определенный тип.

Рассмотрим пример передачи параметров-переменных без типа:

```
Function VarEqual(Var Source, Dest; EqSize: Word): Boolean;
    // Функция сравнения переменных Source и Dest длины EqSize
Type
    ArrBytes = Array [0 .. 1000] Of Byte;
    // Объявление вспомогательного локального типа данных
Var
    N: Integer;
Begin
    N := 0;
    While (N < EqSize) And (ArrBytes(Dest)[N] =
          ArrBytes(Source)[N])
          Do Inc(N);
    Equal := N = Size;
End;
```

Эту функцию можно использовать для сравнения любых двух переменных, размер которых не превышает 1000 байт. Например, в программе присутствуют следующие объявления:

```
Type
     TVector = Array[1 .. 10] Of Integer;
     TPoint = Record
          X, Y: Integer;
End;
Var
     Vector1, Vector2: TVector;
     Point1, Point2: TPoint;
```


Тогда вызов функции:

```
VarEqual(Vector1, Vector2, SizeOf(Vector))
```

обеспечит сравнение массивов Vector1 и Vector2;

```
VarEqual(Vector1, Vector2, SizeOf(Integer)*10)
```

— сравнение первых 10 элементов массивов Vector1 и Vector2;

```
Equal(Point1, Point2, SizeOf(TPoint))
```

— сравнение переменных Point1 и Point2;

```
Equal(Vector1[1], Point2.Y, SizeOf(Integer))
```

— сравнение значений Vector1[1] и Point2.Y.

Параметры-константы. Формальные параметры-константы носят пограничный характер между двумя категориями. С одной стороны, это параметры, которые передаются в подпрограмму по наименованию, т. е. ссылкой на глобальное размещение фактического параметра, но, с другой стороны, внутри подпрограммы действует запрет на изменение значения параметра. Использовать параметры-константы удобно вместо параметров-значений, когда параметр характеризуется большим размером занимаемой памяти. Параметры-константы введены в описание языка только начиная с версии Turbo Pascal 7.0. Синтаксис объявления параметров-констант следующий:

```
<Объявление параметров-переменных> ::=
     Const <Список идентификаторов>:<Тип параметра>|
     Const <Список идентификаторов>
```

В рассмотренном ранее примере объявления функции с параметром-значением более эффективным будет использование параметра-константы, который обеспечит передачу в процедуру адреса размещения строки:

```
// Функция вычисления количества запятых в строке
Function NmbComma(Const S: String): Integer;
Var i, L, Nmb :Integer;
Begin
     Nmb := 0;
     L := Length(S);
     For i := 1 To l Do
          If S[i] = ',' Then inc(Nmb);
     NmbComma := Nmb;
End;
```

Аналогично объявлению параметров-переменных в объявлении параметров-констант может отсутствовать указание типа. В этом случае фактический параметр может быть переменной любого типа, и использование его внутри подпрограммы предполагает предварительное преобразование к конкретному типу.

3.7. Структура программ и модулей

Как уже отмечалось выше, программа на языке Pascal представляет собой последовательность (см. рис. 3.1):

```
<Программа>::= <Заголовок> <Раздел объявлений>
      <Тело программы>
```

В ЯП Turbo Pascal и Object Pascal имеются следующие особенности в структуре программы по равнению со стандартом языка:

- заголовок программы не обязателен и не обрабатывается компилятором;
- порядок размещения разделов объявлений произволен (в отличие от стандарта, где этот порядок строго фиксирован);
- в программе может быть указано несколько одинаковых разделов объявлений;
- существует раздел объявления используемых модулей (в стандарте языка этот раздел отсутствовал).

Заголовок программы

Заголовок программы содержит служебное слово **Program**, имя программы и ее параметры. Синтаксис заголовка следующий:

```
<Заголовок программы> ::= Program <Идентификатор>;|
     Program <Идентификатор>(<Список параметров
     программы>);
```


Список параметров программы в стандарте языка представлял собой список идентификаторов используемых программой файлов ввода-вывода, перечисленных через запятую. В современных реализациях языка список параметров игнорируется.

Заголовок программы в современных реализациях языка является чисто декоративной деталью и компилятором не используется.

Раздел объявлений

Раздел включает в себя объявления:

- меток;
- констант;
- типов;
- переменных;
- процедур и функций;
- используемых модулей.

Раздел объявления меток. Синтаксис раздела объявления меток следующий:

```
<Раздел объявления меток> ::= Label
     <Список идентификаторов меток>;
<Список идентификаторов меток> ::= <Идентификатор метки>|
     <Список идентификаторов меток>,<Идентификатор метки>
```

Например:

```
Label 1, A1, 99;
```

Раздел объявления констант. Синтаксис раздела объявления констант следующий:

```
<Раздел объявления констант> ::= Const
     <Список объявлений констант>;
<Список объявлений констант> ::= <Объявление константы>|
     <Список объявлений констант>; <Объявление константы>
<Объявление константы> ::= <Объявление простой константы>|
     <Объявление типизированной константы>
<Объявление простой константы> ::= <Идентификатор> =
     <Значение>
<Объявление типизированной константы> ::= <Идентификатор>:
     <Тип>= <Значение>
```

Например:

```
Const
     CInt = 999;
     OddDigits : Set Of 0..9 = [1, 3, 5, 7, 9];
     RArr : Array [1..5] Of Real = (1.2, 0.5, -3.0,
          75.12, 123.675);
```

Раздел объявления типов. Синтаксис раздела объявления типов следующий:

```
<Раздел объявления типов> ::= Type <Список объявлений типов>;
<Список объявлений типов> ::= <Объявление типа>|
     <Список объявлений типов>; <Объявление типа>
<Объявление типа> ::= <Идентификатор> = <Тип>
<Тип > ::= <Простой тип>|<Структурированный тип>
```

Например:

```
Type
     TPoint = Record
          X, Y, Z: Real
     end;
     TArrPoint = Array [1..1000] Of TPoint;
     TFilePoint = File Of TPoint;
```

Раздел объявления переменных. Синтаксис раздела объявления переменных следующий:

```
<Раздел объявления переменных> ::=
Var <Список объявлений переменных>;
<Список объявлений переменных> ::= <Объявление переменных>|
     <Список объявлений переменных>; <Объявление переменных>
<Объявление переменных> ::= <Список идентификаторов> : <Тип>
<Список идентификаторов> ::= <Идентификатор>|
     <Список идентификаторов>, <Идентификатор>
<Тип > ::= <Простой тип>|<Структурированный тип>|
     <Идентификатор типа>|
```

Например:

```
Var
     PointA, PointB: TPoint;
     LineLen: Real;
     i, j, k: Integer;
     Smess: String;
     ChrMess: Set of 'A'..'Z';
```


Раздел объявления процедур и функций. Синтаксис раздела объявления подпрограмм следующий:

```
<Раздел объявления подпрограмм> ::= <Список объявлений
     подпрограмм>;
<Список объявлений подпрограмм> ::= <Объявление подпрограммы>|
     <Список объявлений подпрограмм> <Объявление
     подпрограммы>
<Объявление подпрограммы> ::= < Объявление процедуры>|
     <Объявление функции>
```

Примеры:

```
     // Функция вычисления факториала натурального числа
Function Factorial(N:Integer): Integer;
Var i, F:Integer;
Begin
     F := 1;
     For i := 2 To 100 Do F := F*i;
     Factorial := F
End;
     // Процедура удаления пробелов из строки
Procedure DeleteBlank(Var S:String);
Var i: Integer;
begin
     i := pos(' ',S);
     While i > 0 do
     Begin Delete(S, i, 1);
          i := pos(' ',S)
     End;
end;
```

Раздел объявления используемых модулей (предложение Uses). Предложение **Uses** идентифицирует все модули, используемые программой. Синтаксис раздела объявления используемых модулей следующий:

```
<Раздел объявления используемых модулей> := Uses
      <Список модулей>;
<Список модулей> ::= <Имя модуля>|
      <Список модулей>, <Имя модуля>
```

Например:

```
Uses
     Windows, Messages, SysUtils, Classes,
     Graphics, Controls, Forms, Dialogs,
     StdCtrls;
```

Тело программы

Тело программы представляет собой составной оператор — начинается словом **Begin** и заканчивается словом **End** с точкой. Точка является признаком конца программы.

Модули

Модули являются в Turbo Pascal основой модульного программирования. Они используются для создания библиотек, которые могут включаться в различные программы (при этом становится необязательным иметь в наличии исходный код), а большие программы могут подразделяться на логически связанные модули.

```
<Модуль> ::= <Заголовок модуля>;<Интерфейсный раздел>
      <Раздел реализации><раздел инициализации>
```

Заголовок модуля. В заголовке модуля определяется имя модуля:

```
<Заголовок модуля> ::= Unit <Идентификатор модуля>
```

Имя модуля используется при ссылке на модуль в предложении **Uses**. Это имя должно быть уникальным, так как два модуля с одним именем не могут использоваться одновременно.

Интерфейсный раздел. В интерфейсном разделе объявляются те константы, типы, переменные, процедуры и функции, которые являются глобальными, иначе говоря, доступными основной программе (программе или модулю, которые используют/вызывают данный модуль). Основная программа имеет доступ к этим элементам, как если бы они были объявлены в блоке, который включает главную программу.

```
<Интерфейсный раздел > ::= Interface
      <Предложение Uses >;<Раздел объявления констант>;
      <Раздел объявления типов>;
      <Раздел объявления переменных>;
      <Раздел заголовков процедур и функций>;
<Раздел заголовков процедур и функций> ::=
      <Список заголовков подпрограмм>
<Список заголовков подпрограмм > ::= <Заголовок подпрограммы>|
      <Список заголовков подпрограмм >;
```



```
<Заголовок подпрограммы>
<Заголовок подпрограммы > ::=
<Заголовок процедуры>|<Заголовок функции>
```

Интерфейсный раздел только перечисляет заголовки процедур и функций. Полные описания процедур и функций находятся в разделе реализации.

Раздел реализации. В разделе реализации приведены описания всех глобальных процедур и функций. В нем также описываются константы, типы, переменные, процедуры и функции, являющиеся глобальными для модуля, но локальными по отношению к основной программе.

```
<Раздел реализации> ::= Implementation <Предложение Uses>;
      <Раздел объявления меток>; <раздел объявления констант>;
      <Раздел объявления типов>;
      <Раздел объявления переменных>;
      <Раздел объявления процедур и функций>
```

Заголовки процедур и функций могут быть продублированы из интерфейсного раздела. Необязательно задавать список формальных параметров, но если список указан, то в случае несоответствия объявления в интерфейсном разделе и разделе реализации компилятор выдаст ошибку.

Раздел инициализации. Раздел инициализации является последним разделом модуля. Он может состоять либо из зарезервированного слова **End** (в этом случае модуль не содержит кода инициализации), либо из операторной части, которая должна выполняться для инициализации модуля.

```
<Раздел инициализации> ::= End| <Составной оператор>
```

Разделы инициализации модулей, которые используются программой, выполняются в том же порядке, в каком модули указаны в предложении **Uses**.

3.8. Организация ввода-вывода данных. Работа с файлами

Ввод-вывод данных в языке Pascal осуществляется путем взаимодействия программы с внешними файлами. Внешний файл — это поименованный файл на диске или устройство вво-

да-вывода (например, клавиатура, принтер или сканер). Во внешних файлах сохраняются результаты работы программы и располагаются данные, служащие источником информации, необходимой для функционирования программы.

Связь с внешними файлами осуществляется через файловые переменные, т. е. переменные файлового типа. Pascal обеспечивает доступ к трем различным категориям файлов:

- текстовым файлам (для связи с такими файлами необходимо объявить переменную типа `Text`);
- типизированным файлам (для связи с ними объявляется переменная типа **`File Of`** <Тип>);
- нетипизированным файлам или файлам без типа (связываются с программой через переменную типа **`File`**).

Работа с каждой из категорий имеет свою специфику, но есть и общие правила обеспечения процессов чтения-записи для всех категорий файлов.

Работу с внешними файлами поддерживают стандартные процедуры и функции ввода-вывода (табл. 3.18). Перед использованием файловой переменной любого типа ее необходимо связать с внешним файлом с помощью вызова процедуры `Assign()` (в расширении языка — Object Pascal для Delphi — эта процедура называется `AssignFile`). После того как связь с внешним файлом установлена, все операции ввода или вывода для внешнего файла осуществляются через присоединенную к нему файловую переменную.

Перед непосредственным чтением-записью данных внешний файл необходимо «открыть». Существующий файл можно открыть с помощью процедуры `Reset()`, а новый файл можно создать и открыть с помощью процедуры `Rewrite()`. Кроме того, текстовые файлы могут быть открыты процедурой `Append()` для добавления данных в конец файла, но при этом они доступны только для записи.

Типизированные и нетипизированные файлы всегда допускают как чтение, так и запись, независимо от того, были они открыты с помощью процедуры `Reset` или с помощью процедуры `Rewrite()`.

Когда начинается выполнение программы, всегда автоматически открываются стандартные текстовые файловые переменные `Input` и `Output` (ввод и вывод). `Input` — это доступный только для чтения текстовый файл, связанный с клавиатурой, а

Таблица 3.18. **Стандартные процедуры и функции ввода-вывода**

Наименование	Статус	Описание
`Append(F)`	P	Открывает существующий текстовый файл для дозаписи. Файл при этом открывается только для записи, внутренний файловый указатель устанавливается на конец файла. Если строка имени файла пустая (`AssignFile(F,'')`), осуществляется связь со стандартным файлом ввода
`AssignFile(F,Name)` `Assign(F,Name)`	P	Присваивает имя внешнего файла (`Name`) файловой переменной `F`. Если строка имени пустая, осуществляется связь со стандартным файлом ввода или вывода
`BlockRead(F,Buf,N[,NTransf])`	P	Читает `N` или менее (если при чтении достигнут конец файла) записей из нетипизированного файла, с которым связана файловая переменная `F`. Результат чтения помещается в переменную `Buf`. Необязательный параметр `NTransf` указывает на количество фактически прочитанных записей. Максимальный размер считываемых данных равен `Size*N`, где `Size` — размер записи, задающийся процедурами `Reset()` и `Rewrite()`. После выполнения операции внутренний указатель файла перемещается на количество считанных записей (`NTransf`)
`BlockWrite(F,Buf,N[,NTransf])`	P	Записывает `N` или менее (если при записи будет до конца заполнен диск) записей в нетипизированный файл, с которым связана файловая переменная `F`. Данные для записи берутся из переменной `Buf`. Необязательный параметр `NTransf` указывает на количество фактически перемещенных записей. Максимальный размер перемещаемых данных равен `Size*N`, где `Size` — размер записи, задающийся процедурами `Reset()` и `Rewrite()`. После выполнения операции внутренний указатель файла перемещается на количество помещенных в файл записей (`NTransf`)
`ChDir(SPath)`	P	Меняет текущую директорию/каталог на директорию с именем, задающимся параметром `SPath`. Если параметр `SPath` содержит имя диска, то меняется и текущий диск
`CloseFile(F)` `Close(F)`	P	Закрывает ранее открытый внешний файл, с которым связана файловая переменная `F`. При необходимости в содержимое файла вносятся произведенные изменения
`Eof(F)`	F	Возвращает статус конца файла, с которым связана файловая переменная `F`. Если параметр опущен, рассматривается стандартный файл ввода. Принимает значение `true`, если обработана последняя запись файла или файл является пустым. В остальных случаях принимает значение `false`

Продолжение табл. 3.18

Наименование	Статус	Описание
`Eoln(F)`	F	Возвращает статус конца строки для текстового файла, с которым связана файловая переменная `F`. Если параметр опущен, рассматривается стандартный файл ввода. Принимает значение `true`, если текущим элементом файла является признак конца строки или `Eof(F)` принимает значение `true`. В остальных случаях принимает значение `false`
`Erase(F)`	P	Удаляет внешний файл, с которым связана файловая переменная `F`. Перед удалением файл обязательно должен быть закрыт с помощью процедуры `CloseFile` (`Close`)
`FilePos(F)`	F	Возвращает текущую позицию (в количестве записей) указателя внутри типизированного или нетипизированного файла, с которым связана файловая переменная `F`. Если указатель находится в начале файла, возвращает значение 0. Применяется только к открытым файлам
`FileSize(F)`	F	Возвращает текущий размер (в количестве записей) типизированного или нетипизированного файла, с которым связана файловая переменная `F`. Применяется только к открытым файлам. Для пустых файлов возвращает значение 0
`Flush(F)`	P	Освобождение буфера текстового файла вывода. Информация из буфера записывается во внешний текстовый файл, с которым связана файловая переменная `F`. Не работает для файлов, открытых на чтение
`GetDir(D, CPath)`	F	Возвращает в значение параметра `CPath` текущую директорию указанного устройства. Номер устройства задается параметром `D: D=0` — текущий диск, `D=1` — диск `a:`, `D=2` — диск `b:` и т. д.
`IOResult`	F	Возвращает целое значение статуса последней операции ввода-вывода
`MkDir(Path)`	P	Создает новую директорию, путь к которой задается параметром `Path`. В строку `Path` не должно входить имя файла
`Read F,v1[,v2,...,vn])`	P	Читает одно или более значений из файла в одну или более переменных. Используется для текстовых и типизированных файлов. Параметр `F` может отсутствовать. Связь в этом случае осуществляется со стандартным текстовым файлом ввода

Продолжение табл. 3.18

Наименование	Статус	Описание
`Readln F,v1[,v2,...,vn])`	P	Читает одно или более значений из текстового файла в одну или более переменных и переводит текущий указатель файла на начало следующей строки. Используется только для текстовых файлов. Параметр `F` может отсутствовать. Связь в этом случае осуществляется со стандартным текстовым файлом ввода. Вызов процедуры в форме `Readln(F)` переводит внутренний указатель файла на начало следующей строки (или на конец файла, если следующей строки не существует)
`Rename(F,NewNm)`	P	Переименовывает внешний файл, с которым связана файловая переменная `F`. Параметр `NewNm` содержит новое имя файла
`Reset(F[,Size])`	P	Открывает на чтение и запись существующий файл, с которым связана файловая переменная `F`. Внутренний указатель файла устанавливается на начало файла. Если строка имени файла пустая (`AssignFile(F,'')`), осуществляется связь со стандартным файлом ввода. Если файловая переменная `F` имеет тип `Text`, то файл открывается только на чтение. После вызова процедуры `Reset()` значение функции `Eof(F)` всегда `false` (за исключением случая пустого файла — тогда `Eof(F)=True`). Необязательный параметр `Size` (тип — целое) используется только для файлов без типа и задает размер пересылаемой записи в байтах. По умолчанию `Size = 128`
`Rewrite(F[,Size])`	P	Создает и открывает на чтение и запись новый файл, имя которого задается процедурой `AssignFile()` (`Assign`) для файловой переменной `F`. Если файл с таким именем уже существует, то он удаляется и на его месте создается новый пустой файл. Внутренний указатель файла устанавливается на начало пустого файла. Если строка имени файла пустая (`AssignFile(F,'')`), осуществляется связь со стандартным файлом вывода. Если файловая переменная `F` имеет тип `Text`, то файл открывается только на запись. После вызова процедуры `Rewrite()` значение функции `Eof(F)` всегда `true`. Необязательный параметр `Size` (тип — целое) используется только для файлов без типа и задает размер пересылаемой записи в байтах. По умолчанию `Size = 128`

Продолжение табл. 3.18

Наименование	Статус	Описание
`RmDir(SPath)`	P	Удаляет пустую директорию, путь к которой указан параметром `SPath`. Если указан путь к несуществующей директории, не пустой директории или к директории, установленной как текущая, выдается сообщение об ошибке
`Seek(F, N)`	P	Устанавливает текущую позицию указателя внутри типизированного или нетипизированного файла, с которым связана файловая переменная `F`, на запись с номером `N`. Нумерация записей ведется со значения 0. Применяется только к открытым файлам. `Seek(F, FileSize(F))` перемещает внутренний указатель на конец файла
`SeekEof(F)`	F	Возвращает статус конца текстового файла, с которым связана файловая переменная `F`. Если параметр опущен, рассматривается стандартный файл ввода. Отличается от `Eof(F)` тем, что стоящие в конце файла символы пробела и табуляции игнорируются
`SeekEoln(F)`	F	Возвращает статус конца строки для текстового файла, с которым связана файловая переменная `F`. Если параметр опущен, рассматривается стандартный файл ввода. Отличается от `Eoln(F)` тем, что стоящие в конце строки символы пробела и табуляции игнорируются
`SetTextBuf (F,Buf[,Size])`	P	Назначает буфер ввода-вывода для внешнего текстового файла, с которым связана файловая переменная `F`. Параметр `Buf` указывает на буфер, который будет использоваться для ввода-вывода. Необязательный параметр `Size` задает размер буфера. Если размер не указан, используется вся переменная `Buf`. Если процедура не используется, ввод-вывод организуется с помощью стандартного буфера размером в 128 байт. Назначение действует до следующей процедуры `AssignFile()` (`Assign`). Процедура не должна применяться к уже открытому файлу, для которого имели место операции ввода-вывода (это может привести к потере данных)
`Truncate(F)`	P	Удаляет часть файла, с которым связана файловая переменная `F`, начиная с текущей позиции до конца. Используется только для типизированных и нетипизированных файлов. Применяется только к открытым файлам

Окончание табл. 3.18

Наименование	Статус	Описание
`Write(F,v1[,v2,...,vn])`	P	Записывает одно или более значений в файл, с которым связана файловая переменная `F`. Используется для текстовых и типизированных файлов. Параметр `F` может отсутствовать. Связь в этом случае осуществляется со стандартным текстовым файлом ввода
`Writeln(F,v1[,v2,...,vn])`	P	Записывает одно или более значений в файл, с которым связана файловая переменная `F`, и затем записывает признак конца строки (только для текстовых файлов). Используется только для текстовых файлов. Параметр `F` может отсутствовать. Связь в этом случае осуществляется со стандартным текстовым файлом вывода. Вызов процедуры в форме `Writeln(F)` приводит к записи в файл признака конца строки

`Output` — это доступный только для записи текстовый файл, связанный с дисплеем.

Любой файл представляет собой линейную последовательность элементов (записей). Каждая запись файла имеет свой порядковый номер. Первая запись файла считается нулевой и имеет порядковый номер 0. Для каждого файла определено понятие текущей позиции внутри файла, т. е. порядковый номер записи (или записей), к которой будут обращены стандартные процедуры чтения-записи.

Обычно доступ к файлам организуется последовательно, т. е. когда некоторая запись считывается с помощью стандартной процедуры `Read()` или записывается с помощью стандартной процедуры `Write()`, текущая позиция файла перемещается к следующей по порядку записи файла. Однако к типизированным и нетипизированным файлам можно организовать прямой доступ с помощью стандартной процедуры `Seek()`, которая перемещает текущую позицию файла к записи с заданным порядковым номером. Текущую позицию в файле и текущий размер файла можно определить с помощью стандартных функций `FilePos()` и `FileSize()`.

Когда программа завершит обработку файла, его необходимо закрыть с помощью стандартной процедуры `Close` (в расширении языка — Object Pascal для Delphi — эта процедура называется `CloseFile`). После завершения процедуры `Close()` связан-

ный с файловой переменной внешний файл обновляется, а файловая переменная может быть использована для обеспечения работы с другим внешним файлом.

По умолчанию, при всех обращениях к стандартным функциям и процедурам ввода-вывода автоматически производится проверка на наличие ошибок. При обнаружении ошибки программа прекращает работу и выводит на экран сообщение об ошибке. С помощью директив компилятора {$I+} и {$I-} эту автоматическую проверку можно включить или выключить. Когда автоматическая проверка отключена (т. е. когда процедура или функция была скомпилирована с директивой {$I-}), ошибки ввода-вывода, возникающие при работе программы, не приводят к ее останову. Результат выполнения операции ввода-вывода при этом можно проверить с помощью стандартной функции IOResult.

Текстовые файлы

При открытии текстового файла внешний файл интерпретируется особым образом: считается, что он представляет собой последовательность символов, сгруппированных в строки, где каждая строка заканчивается признаком конца строки (End Of Line). Признак конца строки представлен символом перевода каретки (CR, Carriage Return), за которым, возможно, следует символ перевода строки (LF, Line Feed).

Для текстовых файлов существует специальный вид операций чтения и записи (Read() и Write()), которые позволяют считывать и записывать последовательности значений, тип которых отличается от типа Char и String. Такие значения автоматически переводятся в символьное представление и обратно. Рассмотрим, например, оператор процедуры

```
Read(F, i, R),
```

где i — переменная типа Integer, а R — переменная типа Real. Использование этой процедуры приведет к считыванию двух последовательностей цифр и знаков, допустимых при записи чисел. Эти последовательности в файле должны быть разделены пробелом, знаком табуляции или признаком конца строки. Первая последовательность будет интерпретирована как десятичное

число, значение которого будет занесено в переменную i, а вторая последовательность будет преобразована в вещественное число, которое будет занесено в переменную R.

Как было отмечено ранее, имеются две стандартные файловые переменные текстового типа — Input и Output. Стандартная файловая переменная Input — это доступный только для чтения файл, связанный со стандартным файлом ввода операционной системы (обычно это клавиатура), а стандартная файловая переменная Output — это доступный только для записи файл, связанный со стандартным файлом вывода операционной системы (обычно это дисплей). Перед началом выполнения программы файлы Input и Output автоматически открываются, как если бы были выполнены следующие операторы:

```
Assign(Input,''); Reset(Input);
Assign(Output,''); Rewrite(Output);
```

После выполнения программы эти файлы автоматически закрываются.

Для некоторых стандартных процедур (например, Read(), ReadLn(), Write(), WriteLn()) не требуется явно указывать в качестве параметра файловую переменную. Если этот параметр опущен, то по умолчанию будут рассматриваться стандартные файловые переменные Input и Output, в зависимости от назначения процедуры или функции. Например, Read(X) соответствует Read(Input,X), а Write(X) — Write(Output,X).

В качестве примера рассмотрим следующую программу. Пусть на диске в текущей директории есть файл с именем filereal.txt, в котором в нескольких строках записаны последовательности действительных чисел в символьном представлении:

```
0.54 1.7  4.56 0.2
1.32 1.524     18   0.92
7.7.
```

Необходимо вычислить сумму чисел и вывести результат на экран.

```
Program SumReal;
Var
      FInput: Text;    // Файловая переменная
      X: Real;         // Переменная для чтения числа из файла
```

```
      Sum: Real;       // Переменная для вычисления суммы
Begin
      {$I-}            // Включение внутренней проверки
                       // правильности операций ввода-вывода
      Sum: = 0;
      Assign(FInput, 'filereal.txt');
      Reset (FInput);
      If IOResult <> 0 Then
            Writeln('Ошибка при открытии файла')
                              // Вывод на экран сообщения
      Else
      Begin
            While Not EoF(FInput) Do  // Внешний цикл -
                                      //до конца файла
            Begin
                  While Not Eoln (FInput) Do //Внутренний цикл
                                             //- до конца строки
                  Begin
                        Read(FInput, X);
                        Sum: = Sum + X
                  End;
                  Readln(FInput)
            End;
            Writeln('сумма=; Sum:8:3); // Вывод на экран
            // значения суммы в формате с фиксированной
            //точкой (общая длина числа - 8 знаков, 3 знака
            //после точки
            Close(FInput)
      End
End.
```

Типизированные файлы

Использование стандартных процедур `Read()` и `Write()` для типизированных файлов отличается от их использования для текстовых файлов тем, что переменные, в которые читается или из которых записывается информация, должны иметь тот же тип данных, что и компоненты (записи) типизированного файла. Таким образом, при чтении или записи типизированных файлов всегда происходит перемещение данных одинаковой длины.

Функция `FileSize()` для типизированных файлов возвращает количество записей, находящихся в файле, а функция `FilePos()` позволяет определить номер текущей записи файла.

Для решения такой задачи предложим программу `Students_List`.


```
Program Students_List;
Label 1, 2;
Type TStud = Record   //Тип данных, представляющий запись
                      //на одного студента
     Family: String[30];
     Name: String[15];
     Patr: String[20];
     Group: String[5];
     Course: Integer;
     Subject: String[30];
     Mark: Integer;
End;
Var
     RStud: TStud;
     FStud: File Of TStud; //Переменная типизированного файла
     ch :Char;
Begin
{$I-}
Assign(FStud, 'Students.rec');
Rewrite(FStud);            //Открытие файла на запись
If IOResult<> 0 Then
     Begin
          Writeln('Ошибка открытия выходного файла ');
          Goto 2
     End;
Repeat
     With RStud Do         //Ввод с клавиатуры значений
                           //полей очередной записи
     Begin
          Write('Фамилия (30) : '); Readln(Family);
          Write('Имя (15) : '); Readln(Name);
          Write('Отчество (20) : '); Readln(Patr);
          Write('Группа (5): '); Readln(Group);
          Write('Курс : '); Readln(Course);
          Write('Предмет (20): '); Readln(Subject);
          Write('Оценка : '); Readln(Mark);
     End;
     Write(FStud,RStud);   //Занесение в файл очередной
                           //записи
     If IOResult<> 0 Then
     Begin
          Writeln(' Ошибка записи в файл ');Goto 1
     End;
     //Запрос на продолжение или окончание ввода данных
     Write('Продолжить ввод (д/н)? '); readln(ch);
Until ch = 'н';
1: Close(FStud);    //Закрытие файла
2:
End.
```

Нетипизированные файлы

Нетипизированные файловые переменные используются в основном для обеспечения прямого доступа к любому файлу на диске независимо от его типа и структуры.

Любой нетипизированный файл объявляется со словом **File** без атрибутов, например:

```
Var
     Datafile : File;
```

Для нетипизированных файлов в процедурах `Reset()` и `Rewrite()` допускается указывать дополнительный параметр, чтобы задать размер записи, использующийся при передаче данных.

По умолчанию длина записи равна 128 байт. Предпочтительной длиной записи является длина записи, равная 1, поскольку это единственное значение, которое точно отражает размер любого файла (если длина записи равна 1, то неполные записи невозможны).

За исключением процедур `Read()` и `Write()` для всех нетипизированных файлов допускается использование любой стандартной процедуры, которую разрешено использовать с типированными файлами. Вместо процедур `Read()` и `Write()` здесь используются соответственно процедуры `BlockRead()` и `BlockWrite()`, позволяющие пересылать данные с высокой скоростью.

Рассмотрим задачу создания предметных ведомостей с подсчетом среднего балла за экзамен на основе файла, полученного в предыдущем примере.

Необходимо получить текстовый файл, представляющий собой последовательный вывод оформленной в виде ведомости информации об экзамене по каждому предмету, наименование которого встречается в файле `Students.rec`.

Предлагается следующий алгоритм обработки файла `Students.rec`:

- прочитать файл и построить в памяти однонаправленный список записей типа `TStud_List`;
- построить цикл обработки списка (до тех пор, пока он не станет пустым) по следующему принципу:
  - по первой записи списка построить заголовок ведомости и запомнить наименование предмета, курс и номер группы, удалить запись из списка;

  - далее, для каждой следующей записи списка провести сравнение значений соответствующих полей (предмет, курс, группа) с сохраненными ранее значениями. Если значения совпадают, то информация о фамилии, имени, отчестве и оценке студента заносится в выходной текстовый файл и запись удаляется.

Алгоритм реализован в программе `List_Prep`:

```
Program List_Prep;
Label 1;
Type TStud = Record
     Family : String[30];
     Name   : String[15];
     Patr   : String[20];
     Group  : String[5];
     Course : Integer;
     Subject: String[30];
     Mark   : Integer;
End;
TPStud_List = ^TStud_List;
TStud_List = Record
     RecStud : TStud;
     Next: TPStud_List;
End;
Var
     Fstud: File;
     BList,
     EList,
     CurList: TPStud_List;
     ch     : Char;
     Gr     : String[5];
     SumMark,
     NStud,
     Cr     : Integer;
     Subj   : String[30];
     Flist  : Text;
Procedure DelRec(rec:TPSTud_List);//Процедура удаления записи
                                  //из списка
Begin
     If EList <> nil Then EList^.Next := Rec^.Next;
     If BList = rec Then BList := rec^.Next;
     CurList := Rec^.Next;
     Dispose(rec);
End;
Begin
     {$I-}
     Assign(FStud, 'Students.rec');
```

```
Reset(FStud, SizeOf(TStud));  //Открытие входного файла
If IOResult <> 0 Then
Begin
     Writeln('Ошибка открытия входного файла');Goto 1
End;
Assign(FList, 'Students.txt');
Rewrite(FList);   //Открытие файла для вывода результата
If IOResult <> 0 Then
Begin
     Writeln('Ошибка открытия выводного файла');Goto 1
End;
new(BList);
BlockRead(FStud, BList^.RecStud,1);  //Чтение первой
                //записи, организация начала списка
If IOResult <> 0 Then
Begin
     Writeln('Ошибка чтения входного файла');Goto 1
End;
BList^.Next := nil;
EList := BList;
                              // Цикл создания списка
While not Eof(FStud) Do
Begin
     new(CurList);
     BlockRead(FStud, CurList^.RecStud,1);
     If IOResult <> 0 Then
     Begin
          Writeln('Ошибка чтения входного файла');
          Goto 1
     End;
     EList^.Next := CurList;
     EList:=CurList;
     EList^.Next:= nil
End;
                              // Цикл обработки списка
While BList <> nil Do
Begin
     CurList := BList;
     With CurList^.RecStud Do
     Begin
          Subj := Subject;
          Gr   := Group;
          Cr   := Course;
          SumMark := Mark;
          NStud := 1;         // Формирование
                              // заголовка ведомости
          Writeln(FList,'Курс      : ', Cr);
          Writeln(FList,'Группа  : ', Gr);
          Writeln(FList,'Предмет: ', Subj);
          Writeln(FList);  Writeln(FList);
```



```
          Writeln(FList, Family,
          ' ':(30-ord(Length(Family))), Name,
          ':(15-ord(Length(Family))), Mark:3);
     End;
     BList:= CurList^.Next;
     Dispose(CurList);
     CurList := Blist;
     EList := nil;
// Цикл выбора записей для занесения в текущую ведомость
     While CurList <> nil Do
     Begin
          With CurList^.RecStud Do
          If (Cr = Course) And (Gr = Group)
          And (subj = Subject)
          Then
          Begin
               Writeln(FList,Family,' ':
               (30-ord(Length(Family))),Name,
               ' ':(15-ord(Length(Family))),,Mark:3);
               inc(NStud);
               inc(SumMark,Mark);
               DelRec(CurList)
          End
          Else
          Begin
               EList := CurList;
               CurList := CurList^.Next
          End;
     End;
     Writeln(FList);  // Завершение формирования
                      // очередной ведомости
     Writeln(FList,'Средний балл      : '
     , (SumMark/NStud):3:2);
     Writeln(FList,'Кол-во студентов: ', NStud);
     Writeln(FList);
     Writeln(FList);
End;
Close(FStud); Close(FList);      // Закрытие файлов
1:
End.
```

***Переменная* FileMode**. Переменная FileMode, определенная в модуле System, устанавливает код доступа, когда типизированные и нетипизированные файлы (но не текстовые) открываются с помощью процедуры Reset.

По умолчанию задается значение FileMode, равное 2, которое разрешает и чтение, и запись. Присваивание другого значе-

ния переменной FileMode приводит к тому, что все последующие вызовы процедуры Reset() будут использовать этот режим.

Диапазон возможных значений FileMode задает:

FileMode = 0 — только чтение;

FileMode = 1 — только запись;

FileMode = 2 — чтение-запись.

Новые файлы, созданные с помощью процедуры Rewrite(), всегда открываются в режиме Read/Write, соответствующем значению переменной FileMode = 2.

Контрольные упражнения

1. Записать на языке Pascal арифметические выражения, приведенные в математической записи:

   1) $\left(a+\frac{b}{c}\right)\cdot d$;

   2) $\ln x + y^k x$;

   3) $\sqrt{\sin^2 x(x^2+1)+\cos^2(x^3+2{,}5)}$;

   4) $2a\cdot \operatorname{arctg} x$;

   5) $\arcsin x$;

   6) $\frac{p^q}{r^{s+t}}$;

   7) $\frac{\sin x+\cos x}{x^2+1}$;

   8) $2e^{-x}$;

   9) $\frac{2+\cos a}{\sqrt{a^2+1}}$.

2. Вычислить результаты выражений:

   1) A*2+A >= A*A;

   2) Cos(Abs(X)) = Cos(X);

   3) Sin(X) <= 1;

   4) Int(X) <= X;

   5) Ln(100^2) = 4;

   6) X mod 3 > 3;

   7) A >= A div 5 * 5.

3. При каких значениях X значения выражений истинны:

   1) Sin(X) = X;

   2) X mod 2 = 0;

   3) X*X = X+X;

   4) Sqr(X) > X.

4. Записать логические выражения, которые принимают значение true только при выполнении указанных условий:

   1) переменная X принимает значение в интервале [2, 5);

   2) переменная X принимает значение вне интервала (0, 1);

   3) переменная X принимает значение в одном из интервалов (0, 0,3) и [–5, –2,5);

   4) значение целой переменной N делится на 2 и на 3;

   5) значение целой переменной N делится на 5 или на 7;

   6) значение положительной целой переменной N при делении на 5 дает в остатке 3;

   7) ближайшее целое число к значению X четно и отлично от 0.

5. Записать в виде оператора присваивания следующие действия

   1) присвоить переменной Flag значение true, если переменная X принимает значение в одном из промежутков (0; 2) и (5; 25];

   2) присвоить переменной Flag значение true, если значение целочисленной переменной X четно (нечетно);

   3) присвоить переменной Flag значение true, если значение целочисленной переменной X делится на 5 и на 3;

   4) присвоить переменной Flag значение true, если треугольник со сторонами A, B, C равнобедренный.

6. Записать алгоритмы решения следующих задач двумя способами, используя или только условные операторы либо же операторы присваивания и условные операторы в неполной форме:

   1) присвоить переменной y значение 0, если $x \ge 0$, или значение $x^2 - 3x + 5$, если $x < 0$;

   2) присвоить переменной y значение $\sin x + 1$, если $x < 1$, значение $\cos x$, если $x > 2$, и значение x в остальных случаях;

   3) присвоить переменным max и min соответственно наибольшее и наименьшее из значений a, b;

   4) присвоить переменной y значение $\frac{1}{(1+x)^2}$, если $x \ge 1$, а в остальных случаях значение x^2+2, если $x < 0$, и значение $2x+1$, если $x \ge 0$.

7. Разработать процедуры и функции для решения следующих задач обработки массивов:

   - подсчитать количество отрицательных элементов одномерного массива вещественных чисел;

- подсчитать сумму квадратов положительных элементов одномерного массива вещественных чисел;
- подсчитать сумму квадратов отрицательных элементов одномерного массива вещественных чисел;
- преобразовать одномерный массив вещественных чисел, присвоив каждому элементу квадрат его значения;
- преобразовать одномерный массив вещественных чисел, уменьшив каждый элемент на абсолютную величину среднего значения элементов массива;
- преобразовать одномерный массив вещественных чисел, занеся в каждый элемент сумму всех предыдущих элементов (в первый элемент при этом необходимо поместить значение 0);
- подсчитать сумму квадратов диагональных элементов двумерного массива вещественных чисел;
- подсчитать максимальную сумму элементов в строках двумерного массива вещественных чисел;
- подсчитать минимальную сумму элементов в столбцах двумерного массива вещественных чисел;
- преобразовать двумерный массив вещественных чисел, занеся значение 0 во все элементы с двумя четными индексами.

8. Разработать процедуры и функции для решения следующих задач обработки строк:
   - подсчитать количество слов в строке (разделителем считать символ пробела);
   - подсчитать количество предложений в строке (разделителями считать точку, вопросительный и восклицательный знаки);
   - подсчитать количество гласных (согласных) букв в строке;
   - подсчитать количество знаков препинания (знаки препинания — запятая, точка, точка с запятой, двоеточие);
   - преобразовать строку, убрав лишние пробелы (оставить по одному пробелу между словами);
   - удалить из строки повторы слов.

Глава 4
ЭЛЕМЕНТЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ РАЗРАБОТКИ ПРОГРАММ

В истории применения компьютеров вычислительная техника всегда использовалась на пределе своих возможностей. Каждое новое достижение в аппаратном либо в программном обеспечении приводит к попыткам расширить сферу применении ЭВМ, что влечет за собой постановку новых задач, для решения которых, в свою очередь, нужны новые вычислительные средства.

Основа для массового промышленного программирования была создана с разработкой новых методов построения программ. Одной из первых и наиболее широко применяемых технологий программирования стало структурное программирование. Этот метод до сих пор не потерял своего значения для многих классов задач.

Основой ***структурного подхода*** являются два основополагающих принципа:

- использование процедурного стиля программирования;
- последовательная декомпозиция алгоритма решения задачи сверху вниз.

В соответствии с этим подходом задача решается путем выполнения следующей последовательности действий. Первоначально задача формулируется в терминах ввода данных — вывода результата: на вход программы подаются некоторые данные, программа работает и выдает ответ. После этого начинается последовательное расчленение (декомпозиция) всей задачи на отдельные более простые действия. При этом на любом этапе декомпозиции программу можно проверить, применяя механизм так называемых «заглушек» — процедур, имитирующих

вход и/или выход процедур нижнего уровня. «Заглушки» позволяют проверить логику верхнего уровня до реализации следующего, т. е. на каждом шаге разработки программы существует работоспособный «каркас», который постепенно обрастает деталями.

Структурное программирование ясно определило значение модульного построения программ (т. е. разбиения монолитных программ на группу отдельных модулей) при разработке больших проектов, но в языках программирования единственным способом структуризации программы оставалось составление ее из подпрограмм и функций.

Объектно-ориентированное программирование появилось и получило широкое распространение именно благодаря попыткам разрешения следующих проблем, возникавших в процессе проектирования и разработки программных комплексов:

- развитие языков и методов программирования не успевало за все более растущими потребностями в прикладных программах. Единственным реальным способом снизить временные затраты на разработку был метод многократного использования разработанного программного обеспечения, т. е. проектирование новой программной системы на базе разработанных и отлаженных ранее модулей, которые выступают в роли своеобразных «кирпичиков», ложащихся в фундамент новой разработки;
- ускорение разработки программного обеспечения требовало решения проблемы упрощения их сопровождения и модификации;
- не все задачи поддаются алгоритмическому описанию по требованиям структурного программирования, т. е. в целях упрощения процесса проектирования необходимо было решить проблему приближения структуры программы к структуре решаемой задачи.

Решение перечисленных проблем в рамках создания объектно-ориентированного подхода к программированию и породило три его основных достоинства:

- упрощение процесса проектирования программных систем;
- легкость сопровождения и модификации;
- минимизацию времени разработки за счет многократного использования готовых модулей.

4.1. Основные понятия объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) — подход, использующий объектную декомпозицию, основанную на выделении объектов и связей между ними. В отличие от процедурного подхода к программированию, когда описание алгоритма решения некоторой задачи представляет собой последовательность действий, объектно-ориентированный подход предлагает описывать программные системы в виде взаимодействия объектов и, прежде всего, создать некоторый инструментарий, присущий решаемой задаче, а затем уже программировать в терминах этой задачи.

Объектно-ориентированное программирование является мощным средством для моделирования отношений между объектами практически в любой предметной области. Особенно удобно и легко в объектах выразить взаимодействие между различными элементами графического интерфейса пользователя.

Динамика поведения объектно-ориентированной системы описывается в терминах обмена сообщениями между объектами. «Чистое» объектно-ориентированное программирование поддерживает всего одну операцию: послать объекту `O` сообщение `Msg` с параметрами $P_1 \ldots P_n$. Как само сообщение, так и его параметры — это, в свою очередь, объекты, которые могут быть результатами обработки других сообщений.

Объекты и классы

В объектно-ориентированном программировании базовыми единицами являются объекты.

Объект — это сущность, сочетающая в себе как совокупность данных, так и действий над ними. Каждый объект характеризуется своим состоянием, которое определяется текущими значениями его атрибутов (характеристических свойств). Атрибутами объекта могут быть не только атомарные величины (числа, символы, логические значения), но и объекты. Все атрибуты объекта делятся на две группы — «внутренние» атрибуты (те, которые доступны только самому объекту) и атрибуты, входящие в интерфейс объекта (их часто называют свойствами).

Интерфейс объекта — это описание того, как он может взаимодействовать с окружающим миром. Интерфейс объекта определен полностью его свойствами и методами. Методы — это действия над данными объекта.

Объект сохраняет свое состояние до тех пор, пока оно не будет изменено через вызов методов этого объекта. Это существенно ограничивает возможность несанкционированного или некорректного доступа к объекту. Управление состояниями объекта через вызов методов в конечном итоге определяет поведение объекта.

Объект может посылать сообщения другим объектам и принимать сообщения от них. Сообщение является совокупностью данных определенного типа, передаваемых объектом-отправителем объекту-получателю, имя которого указывается в сообщении. Получатель реагирует на сообщение выполнением некоторого метода.

Таким образом, совокупность объектов образует среду, в которой вычисления выполняются путем обмена сообщениями между объектами. Состояние вычислительной среды в ООП характеризуется совокупным состоянием объектов, это в принципе отличает объектно-ориентированные вычисления от вычислений, заданных на процедурно-ориентированных языках.

Класс. Объекты с одинаковыми возможностями (данными и методами) объединяет термин класс. Класс описывает общее поведение и характеристики набора аналогичных друг другу объектов. Объект — это экземпляр класса или, другими словами, переменная, тип которой задается классом. Объекты в отличие от классов реальны, т. е. существуют и хранятся в памяти во время выполнения программы. Соотношения между объектом и классом аналогичны соотношениям между переменной и типом.

Объектно-ориентированное программирование позволяет программировать в терминах классов: определять классы, конструировать производные классы (подклассы) на основе существующих классов, создавать объекты, принадлежащие классу (экземпляры класса). Каждый класс задается своим описанием на языке ООП, которое включает информацию, необходимую для создания объектов данного класса и организации работы с ними.

У каждого объекта есть ссылка на класс, к которому он относится. При приеме сообщения объект обращается к классу для обработки данного сообщения. Если сам класс не располагает

методом для обработки сообщения, оно может быть передано вверх по иерархии наследования.

Так как объекты взаимодействуют исключительно за счет обмена сообщениями, они могут ничего не знать о реализации обработчиков сообщений друг у друга. Таким образом, взаимодействие описывается исключительно в терминах сообщений или событий, которые достаточно легко привязать к конкретной предметной области. В этом состоит свойство абстракции (т. е. описания взаимодействия исключительно в терминах предметной области).

Свойства объектов

Структуру и поведение объектов в ООП определяют свойства инкапсуляции, наследования и полиморфизма.

Инкапсуляция. Объединение всех данных и методов объекта (включая данные и методы объектов-предков) называется *инкапсуляцией*. Механизм инкапсуляции скрывает данные и методы объекта от внешнего вмешательства или неправильного использования, облегчая тем самым понимание работы программы, а также ее отладку и модификацию, так как только в очень редких случаях разработчика интересует внутренняя реализация объектов — главное, чтобы объект обеспечивал функции, которые он должен предоставить.

Взаимодействие с объектом происходит через интерфейс. Обычно интерфейс определяет единственный способ входа в объект и выхода из него, детали реализации остаются инкапсулированными. Интерфейс объекта составляют его свойства, методы и события. Только содержимое интерфейса предоставляется данным объектом в распоряжение других объектов, благодаря этому предотвращается доступ других объектов к внутренним переменным состояния объекта. Таким образом, инкапсуляция обеспечивает использование объекта, не требуя знания того, как именно он устроен внутри.

Данные и методы внутри объекта могут обладать различной степенью открытости (или доступности). Некоторые из них объявляются общедоступными, другие доступны только из методов самого объекта. Обычно открытые данные используются для

того, чтобы обеспечить контролируемый интерфейс с его закрытой частью.

Наследование позволяет определять новые классы в терминах существующих классов и повторно использовать уже созданную часть программного кода в других проектах. Посредством наследования формируются связи между объектами, а для выражения процесса наследования используют понятия «родители» и «потомки». В программировании наследование служит для сокращения избыточности кода, и суть его заключается в том, что уже существующий интерфейс вместе с его программной частью можно использовать для других объектов. При наследовании могут также проводиться изменения интерфейсов.

Наследование — это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него.

Обычно, если объекты соответствуют конкретным сущностям реального мира, то классы являются абстракциями, выступающими в роли понятий. Между классами, как между понятиями, существуют иерархические отношения, связывающие класс с его родителем и потомком, которые и реализуются механизмом наследования.

Наследование выполняет в ООП несколько важных функций:

- моделирует концептуальную структуру предметной области (в виде иерархии классов);
- позволяет использовать одни и те же писания для задания разных классов;
- обеспечивает программирование больших систем путем пошаговой конкретизации классов.

Обычно в объектно-ориентированных языках существует класс, являющийся вершиной всей иерархии наследования (как правило, называемый `Object`), а одним из свойств объекта является ссылка на объект-предок, которому переадресуются все сообщения, не обрабатываемые данным объектом.

Полиморфизм характеризует способность различных объектов по-разному обрабатывать одинаковые сообщения. При этом различные объекты используют одинаковую абстракцию, т. е. могут обладать свойствами и методами с одинаковыми именами. Однако обращение к ним будет вызывать различную реакцию для различных объектов. На практике это означает, что можно создать общий интерфейс вызова для группы близких по смыслу действий.

Большое достоинство полиморфизма состоит в том, что при использовании объекта можно вызывать определенное свойство или метод, не заботясь о том, как объект выполняет задачу. Таким образом, полиморфизм означает присваивание методу одного имени или обозначения, которое совместно используется объектами различных классов, при этом каждый объект реализует действие способом, соответствующим его классу.

Проиллюстрируем свойства ООП на примере известной задачи размещения и перемещения некоторого количества мерцающих разноцветных точек на плоскости экрана.

Точка на экране характеризуется координатами `X`, `Y` , имеет цвет, может быть видимой или невидимой и может перемещаться по экрану.

Очевидно, что основой изображения точки является ее положение (позиция) на экране (например, значения координат `X` и `Y` относительно левого верхнего угла экрана). Таким образом, может быть объявлен класс `Позиция` со свойствами — координатами `X` и `Y` , имеющими тип целое число, и методом `НазначитьКоординаты`:

```
Позиция(X , Y, НазначитьXY)
```

Далее объявим класс `Точка`, который может быть описан следующим образом:

```
Точка(X , Y, Видимость, Цвет, НазначитьXY, НазначитьЦвет,
      Зажечь, Погасить, Переместить)
```

При таком объявлении свойства и методы класса `Позиция` полностью входят в класс `Точка`. Используя механизм наследования, опишем класс `Точка` как потомка класса `Позиция`:

```
Точка (Позиция, Видимость, Цвет, НазначитьЦвет, Зажечь,
       Погасить, Переместить)
```

Интерфейс такого объекта составляют только методы. Атрибуты `X`, `Y`, `Видимость`, `Цвет` получают конкретные значения при вызове интерфейсных методов `НазначитьXY`, `Переместить` (для атрибутов `X`, `Y`), `НазначитьЦвет` (для атрибута `Цвет`), `Зажечь`, `Погасить` (для атрибута `Видимость`). Тогда для создания объекта такого класса (например, `Точка1`) необходимо активизировать процедуру

размещения объекта и присвоить атрибутам конкретные значения, активизировав соответствующие методы, например:

```
Точка1 = Создать.Точка;
Точка1. НазначитьXY(3,5);
Точка1.НазначитьЦвет(Красный);
Точка1.Зажечь;
```

В результате таких действий на экране появится красная точка с координатами $X = 3$, $Y = 5$.

Чтобы проиллюстрировать свойство полиморфизма, объявим класс цветных кругов, которые задаются координатой центра и радиусом. Этот класс должен наследовать все возможности класса Точка, но методы НазначитьЦвет, Зажечь, Погасить, Переместить, по результату действия являясь теми же самыми, фактически не могут быть реализованы одинаковой последовательностью команд в случае с точкой и кругом. Эта ситуация разрешается путем использования полиморфизма — методы классов имеют одинаковое имя (и, соответственно, одинаковый ожидаемый результат), а внутреннее описание методов будет различно для разных классов. Тогда класс Круг может иметь следующее описание:

```
Круг (Точка, Радиус, Назначить цвет, Зажечь, Погасить,
Переместить)
```

Компоненты

Использование при программировании готовых библиотек классов повышает скорость разработки программ и существенно экономит усилия разработчиков. Однако любая такая библиотека перед использованием требует изучения своей структуры и возможностей и, кроме того, должна быть написана на том же языке программирования, что и разрабатываемая программа (конечно, существуют способы сопряжения различных языков программирования, но не всегда ими можно воспользоваться). Эти недостатки послужили причиной появления концепции компонентов.

Фирмой Microsoft с целью стандартизации программных компонентов была разработана технология ActiveX, в основе которой лежит COM (Component Object Model — модель компонентного объекта).

Эта системная технология объединяет совокупность средств, с помощью которых объекты, разработанные различными разработчиками на разных языках программирования и работающие в разных средах, могут взаимодействовать друг с другом без какой-либо модификации их исполняемых модулей (двоичных кодов). Сутью данной технологии является то, что программы строятся из компонентов, представляющих собой объекты. При этом компоненты — непосредственно исполняемые файлы, и тем самым они не связаны с конкретными языками программирования. Компонент достаточно зарегистрировать в операционной системе и он будет доступен любой программе, исполняющейся на данной машине.

Компонент — это объект, объединяющий состояние и интерфейс (способ взаимодействия), который позволяет включать компоненты в различные современные среды разработки приложений. При этом не важно, на каком языке программирования реализован компонент. Он должен просто удовлетворять определенным внешним параметрам и быть нейтральным по отношению к языку программирования, чтобы его можно было использовать в программе на любом языке, поддерживающем компонентную технологию. Так, например, компоненты стандарта ActiveX могут быть одинаково успешно включены в программу, реализованную в среде Visual Basic, и в приложение, разработанное средствами Delphi.

У компонента имеются два типа интерфейсов — интерфейс стадии проектирования и интерфейс стадии выполнения. Интерфейс проектирования позволяет включать компоненты в современные среды разработки приложений, а интерфейс выполнения управляет работой компонента во время выполнения программы.

Состояние компонента может быть изменено только с помощью изменения его свойств и вызова методов.

Разработка любого приложения состоит из двух взаимосвязанных этапов:

- проектирования и создания функционального интерфейса приложения (т. е. набора визуальных компонентов, которые будут обеспечивать взаимодействие пользователя и вычислительной среды);
- программирования процедур обработки событий, возникающих при работе пользователя с приложением.

Проектирование интерфейса. На этом (первом) этапе формирования общего вида главного окна при выполнении приложения и способов управления работой приложения, для каждого компонента необходимо определить его внешний вид, размеры, способ и место размещения в области окна приложения (т. е. реализовать интерфейс разработки и интерфейс выполнения).

Компоненты, доступные проектировщику на этапе разработки приложения, разбиты на функциональные подгруппы. С точки зрения внутренней структуры компоненты разбиваются на три группы. На рис. 4.1 представлена графическая интерпретация этого разбиения.

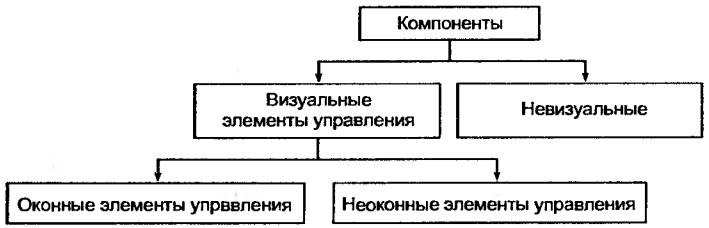


Рис. 4.1. Иерархия групп компонентов схожей внутренней структуры

Визуальные компоненты (элементы управления) характеризуются наличием свойств размеров и положения в области окна и на стадии разработки приложения обычно находятся на форме в том же месте, что и во время выполнения приложения (например, кнопки, списки, переключатели, надписи). Визуальные компоненты имеют две разновидности — «оконные» и «неоконные» (графические):

- оконные визуальные компоненты (самая многочисленная группа компонентов) — это компоненты, которые могут получать *фокус ввода* (т. е. становиться активными для взаимодействия с пользователем) и содержать другие визуальные компоненты.
- неоконные (графические) визуальные компоненты не могут получать фокус и содержать другие визуальные компоненты (например, надписи и графические кнопки).

Невизуальные компоненты на стадии разработки не имеют своего фиксированного местоположения и размеров. Во время выполнения приложения некоторые из них иногда становятся видимыми (например, стандартные диалоговые окна открытия и сохранения файлов), а другие остаются невидимыми всегда (например, таблицы базы данных).

Важной характеристикой компонента, как и любого объекта, являются его *свойства* — атрибуты, определяющие его состояние и поведение. Различают три типа свойств компонента:

- свойства *времени проектирования*. Установленные для них значения будут использоваться в момент первого отображения компонента и в дальнейшем могут быть изменены во время выполнения приложения;
- динамические свойства. Изменением их значений можно управлять только изнутри программного кода (во время выполнения приложения);
- свойства только для чтения, которые могут быть прочитаны и использованы при выполнении программы, но не могут быть изменены.

Непосредственное программирование процедур обработки событий, исходящих от компонентов (второй этап). Основная задача при разработке таких процедур — запрограммировать реакцию на все возможные изменения состояний объектов.

Объектно-ориентированные языки программирования

Объектно-ориентированные языки программирования обязательно должны содержать конструкции, позволяющие объявлять классы, создавать и уничтожать объекты, принадлежащие классам и т. п. Эти языки можно разделить на две группы:

- «чистые» объектно-ориентированные языки, в наиболее классическом виде поддерживающие объектно-ориентированную технологию:
  - Simula (1962);
  - Smalltalk (1972);
  - Beta (1975);
  - Cecil (1992);
  - Java (1995);
- «гибридные», появившиеся в результате внедрения объектно-ориентированных конструкций в популярный процедурный язык программирования:
  - Object Pascal (1984);
  - C++ (1983).

Первым «подлинным» языком ООП считается Smalltalk, который был разработан в Пало-Альто, в лаборатории компании Xerox.

В настоящем учебном пособии в качестве иллюстраций к ООП рассматриваются два примера — интегрированная среда разработки приложений Delphi (в совокупности с СП Object Pascal) и Visual Basic.

Предваряя возражения читателей по поводу того, что это не чисто объектные среды, ответим, что они выбраны с целью иллюстрирования развития уже рассмотренных ЯП и СП. Это, конечно, системы процедурного типа с внедренными элементами ООП, о чем и говорится в названии данной главы.

Среда Delphi (по всей видимости, благодаря своим высоким характеристикам) получила свое название от греческого города, где жил знаменитый дельфийский оракул.

Программный продукт Delphi позволяет создавать, компилировать, тестировать и редактировать проект приложения в единой среде программирования и удачно сочетает в себе несколько передовых технологий:

- высокопроизводительный компилятор в машинный код (скорость компилирования 120 тыс. строк/мин). В процессе работы над приложением разработчик выбирает готовые компоненты и проектирует их визуальное размещение и взаимодействие. После выполнения компиляции получается исполнительный код программы;
- объектно-ориентированную модель компонентов (основных объектов, которые группируются в 270 классов) с отсутствием ограничения по типам создаваемых объектов. Delphi содержит полный набор визуальных инструментов, предназначенных для скоростной разработки приложений на базе готовых компонентов. Число компонентов и область их применения непрерывно растут, так как в их создании могут принимать участие и другие фирмы;
- визуальное построение приложений. Визуальные компоненты пишутся на языке программирования Object Pascal;
- масштабируемые средства построения БД. Одно и то же приложение можно использовать как для локального, так и для клиент-серверного варианта. Среда Delphi включает в себя локальный сервер для поддержки разработки приложений по технологии «клиент — сервер».

В качестве базового языка в среде принят язык программирования Pascal. Сама среда Delphi разработана с использованием Delphi.

Пользователи Delphi — это, во-первых, профессионалы-разработчики информационных систем, и, во-вторых, — программисты, применяющие среду для быстрого решения своих задач.

Visual Basic — основные возможности ЯП были вкратце рассмотрены выше (см. гл. 1).

4.2. Интерфейс среды Delphi

В момент первой загрузки среды Delphi (далее будет представлена версия программного продукта Delphi 7) интерфейс первоначально содержит шесть окон (рис. 4.2):

- главное окно Delphi с заголовком, содержащим имя открытого проекта (на рисунке `Delphi — Project1`, рис. 4.2);
- окно `Обозревателя Дерева Объектов` (`Object Tree View`, рис. 4.2, *1*);
- окно `Инспектора Объектов` (`Object Inspector`, рис. 4.2, *2*, рис. 4.4);
- окно `Конструктора Формы` с заголовком `Form1`, рис. 4.2, *3*, рис. 4.5;
- окно `Редактора Кода`, озаглавленное `Unit1.Pas`, которое первоначально перекрывается окном формы, рис. 4.2, *4*, рис. 4.6;
- окно `Проводника Кода` (`Exploring Unit1.pas`, рис. 4.2, *5*).

Окна Delphi можно перемещать, убирать с экрана, а также изменять их размеры.

Delphi — однодокументная среда, т. е. позволяет одновременно работать только с одним приложением.

Главное окно

Главное окно Delphi, занимающее верхнюю часть экрана, содержит:

- строку `Заголовка`, в которой отображается имя открытого проекта (рис. 4.2, *6*);

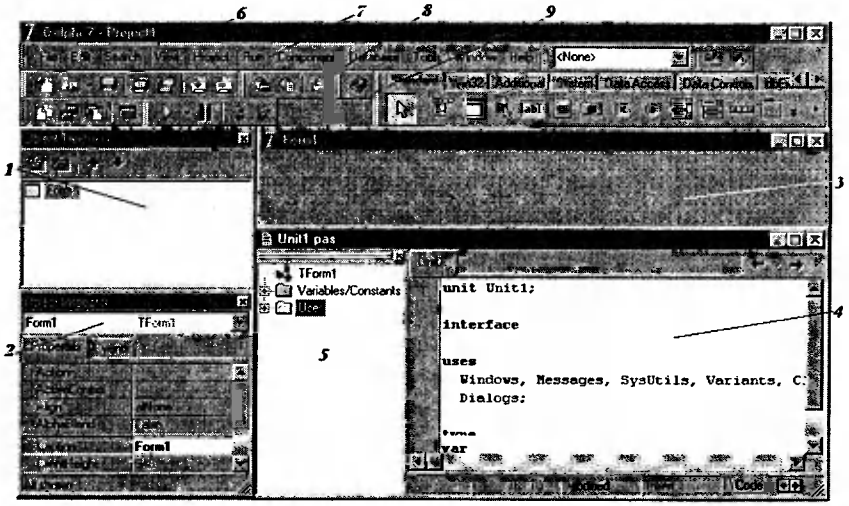


Рис. 4.2. Исходный вид интерфейса Delphi: *1* — Обозреватель Дерева Объектов; *2* — Инспектор Объектов; *3* — Конструктор Формы; *4* — Редактор Кода; *5* — Проводник Кода; *6* — строка Заголовка; *7* — строка Главного Меню; *8* — Панель Инструментов; *9* — Палитра Компонентов

- строку Главного Меню с набором команд для управления разработкой и тестированием приложений (рис. 4.2, *7*);
- Панель Инструментов — кнопок, представляющих собой краткую форму функций меню (рис. 4.2, *8*);
- Палитру Компонентов, отображающую компоненты, с помощью которых создается приложение (рис. 4.2, *9*, рис. 4.3).

Панель Инструментов. Кнопками Панели Инструментов можно вызывать наиболее часто используемые команды, представленные в Главном Меню. Вызвать команды Главного Меню можно также с помощью соответствующих комбинаций клавиш алфавитно-цифровой и функциональной клавиатур.

Кнопки сгруппированы в шесть панелей инструментов:

- Стандартная панель;
- Панель просмотра;
- Панель отладки;
- Пользовательская панель;
- Рабочий стол;
- Интернет.

Можно управлять отображением панелей инструментов и изменять состав кнопок на них. Эти действия выполняются с помощью контекстного меню панелей инструментов или с помощью функции Главного меню (View\ToolBars\Customize...).

Палитра Компонентов используется для выбора и размещения на форме компонентов графического интерфейса. Группы компонентов размещаются на разных вкладках.

В Delphi используется открытая компонентная архитектура, которая позволяет добавлять компоненты в каждую группу и создавать новые группы компонентов. Разные конфигурации среды могут отличаться набором представленных компонентов, однако, всегда разрешено добавлять новые компоненты, независимо от того, где они были созданы.

Компоненты являются «строительными блоками», из которых конструируются формы приложений.

Палитру Компонентов можно настраивать с помощью диалогового окна Palette Properties (Свойства Палитры), которое вызывается командой Properties (Свойства) контекстного меню Палитры Компонентов или командой Component\Configure Palette (Компонент\Настройка палитры) Главного меню. Окно позволяет выполнять операции по удалению, добавлению и перемещению отдельных компонентов и целых страниц (рис. 4.3).

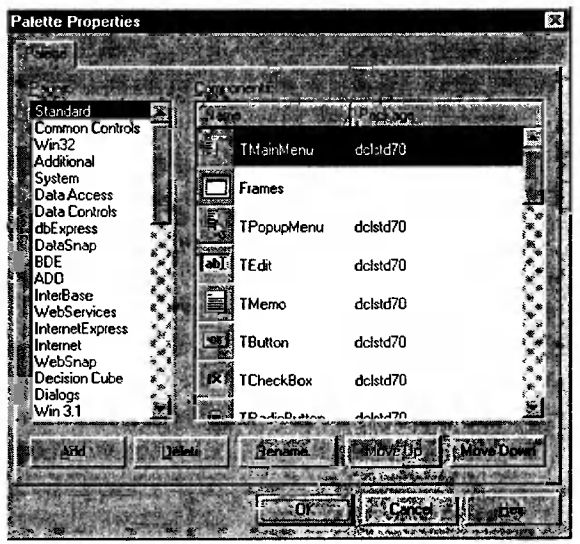


Рис. 4.3. Диалоговое окно свойств Палитры Компонентов

Окно Обозревателя дерева объектов (Object Tree View) после запуска среды находится под главным окном и отображает древовидную структуру объектов текущей формы.

Окно Инспектора объектов (Object Inspector) предназначено для управления компонентами (их размещением и поведением на форме) на этапе проектирования интерфейса. Оно содержит две страницы — Свойства (Properties) и События (Events). Каждый компонент имеет свой набор свойств и событий, определяющий его индивидуальные характеристики и особенности, а также возможности по его использованию в процессе работы приложения.

Страница Свойства (список свойств) отображает и позволяет менять характеристики текущего (выделенного) объекта в окне формы. Когда в среде Delphi создается новое приложение, первоначально Инспектор Объектов отображает свойства текущей формы (см. рис. 4.2). Если при проектировании интерфейса нужно изменить что-нибудь, связанное с определенным компонентом, то это выполняется в Инспекторе Объектов. К примеру, можно изменить заголовок и размер компонента TLabel, изменяя свойства Caption, Height и Width (рис. 4.4).

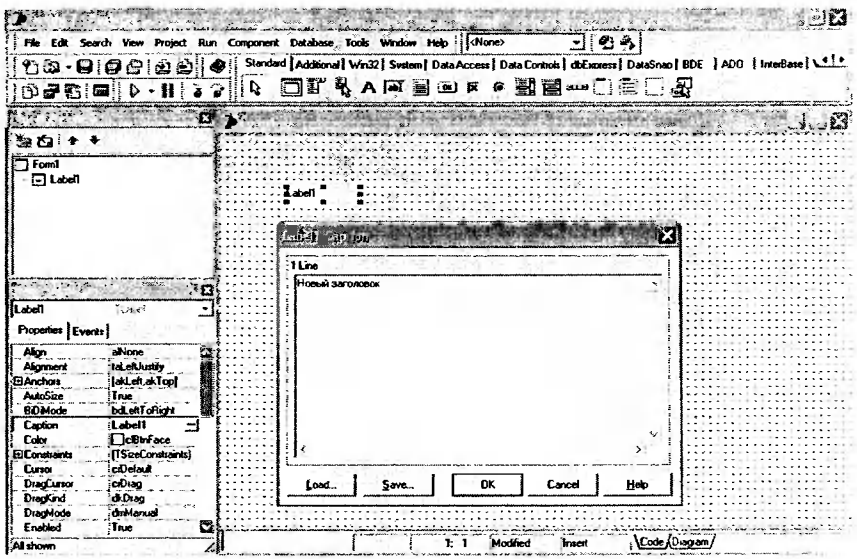


Рис. 4.4. Изменение свойства Caption для TLabel

На странице События (список событий) задаются действия, которые должны выполняться объектом при наступлении определенных событий. Страница событий связана с Редактором Кода — если курсор манипулятора или текстовый курсор расположен в некоторой строке на правой половине страницы, то при двойном нажатии клавиши манипулятора «мышь» автоматически в окне Редактора Кода создается заготовка для процедуры обработки соответствующего события.

Если для какого-либо события объявлена процедура, то в дальнейшем при работе с приложением она выполняется автоматически при возникновении этого события. Такие процедуры называют обработчиками, так как они служат для обработки событий.

Окно Конструктора формы — это окно разрабатываемого приложения, внешний вид и наполнение которого определяются при проектировании. Компоненты графического интерфейса помещаются на форму и располагаются на ней по желанию проектировщика (рис. 4.5). Приложение может иметь несколько форм.

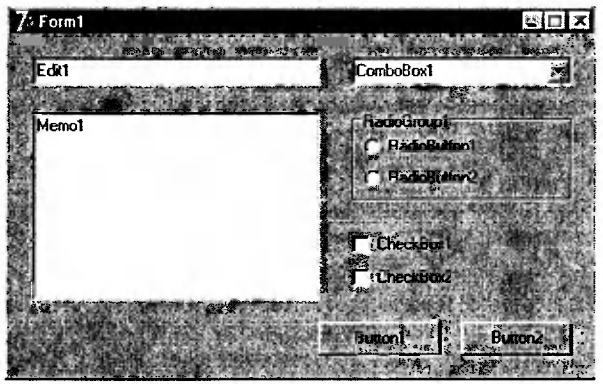


Рис. 4.5. Окно Конструктора Формы

Первоначально форма имеет заголовок Form1. Объекты, размещаемые на форме, должны быть выбраны (с помощью манипулятора «мышь») на Палитре Компонентов и перенесены в область формы (путем указания места расположения).

Редактировать размещение компонента можно с помощью группы команд меню Edit, а также с помощью контекстного меню.

Конструктор формы интуитивно понятен и прост в использовании, что в большой степени упрощает создание визуального интерфейса.

Окно Редактора Кода (рис. 4.6), первоначально скрытое окном формы, предназначено для редактирования исходного кода модуля программы, описывающего данную форму. Это обычный текстовый редактор, с помощью которого можно редактировать текст модуля и другие текстовые файлы приложения (а также и простые текстовые файлы в кодировке Windows). В Редакторе Кода можно открывать несколько файлов, каждый из которых размещается на отдельной странице. Между страницами можно переключаться, т. е. «листать» файлы по ярлычкам.

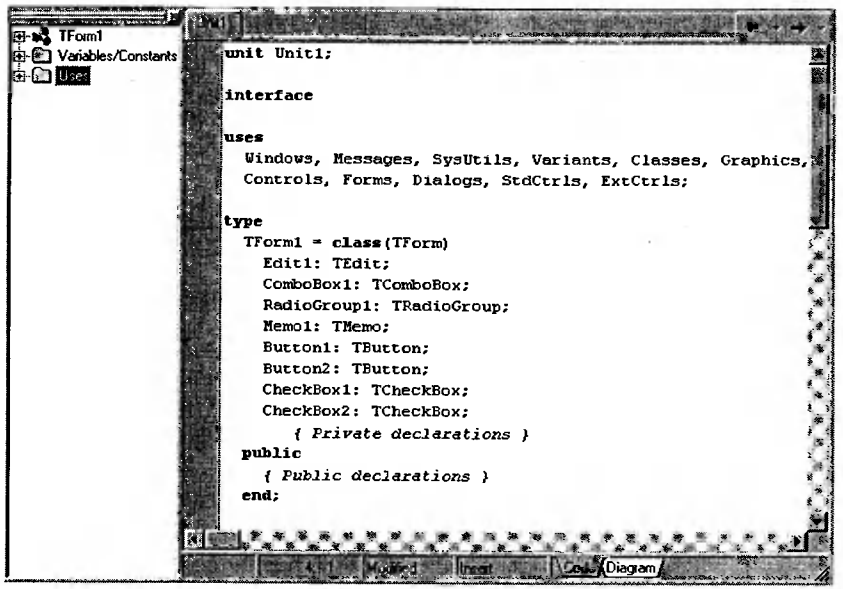


Рис. 4.6. Окно Редактора Кода

Окно Редактора Кода первоначально размещено «под» окном формы, поскольку первый этап разработки формы — размещение на ней элементов графического интерфейса — сопровождается автоматическим генерированием программного кода. Необходимость использования окна модуля для ввода и редактирования наступает, когда программируются обработчики событий для компонентов, размещенных на форме.

Переключение между редактором кода и формой поддерживается функциональной клавишей <F12>.

Окно Проводника Кода располагается в левой части окна редактора кода. В нем в виде дерева отображаются все объекты модуля формы, например переменные и процедуры. В окне Проводника кода удобно просматривать объекты приложения и обращаться к ним, что особенно важно при работе с большими модулями. В функции Проводника входит и автоматизированное создание новых классов.

При закрытии файла закрывается и Проводник кода. Проводник кода можно убирать и вызывать с помощью команды меню View\Code Explorer.

4.3. Характеристика проекта Delphi

Любой проект представляет собой совокупность не менее семи файлов:

- Главный Файл проекта — файл с расширением .dpr, представляет собой основной модуль программы;
- Файл Главной Формы (описания формы) — файл с расширением .dfm, используется для сохранения информации о внешнем виде главной формы;
- Первый Модуль Программы (модуль главной формы) — файл с расширением .pas, автоматически появляется в начале работы;
- Файл Ресурсов — файл с расширением .res. Содержит иконку для проекта, создается автоматически и имеет то же имя, что и главный файл проекта;
- Файл Параметров Проекта — файл с расширением .cfg, текстовый файл для сохранения конфигурации данного проекта. Имя файла совпадает с именем Главного Файла Проекта;
- Файл Параметров Среды (Delphi Options File) — файл с расширением .dof, текстовый файл, в котором хранятся текущие установки параметров проекта, таких, как параметры компиляции, рабочие директории, условные директивы, параметры командной строки. Имя файла совпадает с именем Главного Файла проекта;

- Файл Настроек Рабочей Области Среды (Desktop File) — файл с расширением .dsk, в котором сохраняется состояние среды Delphi для проекта. Имя файла совпадает с именем Главного Файла Проекта.

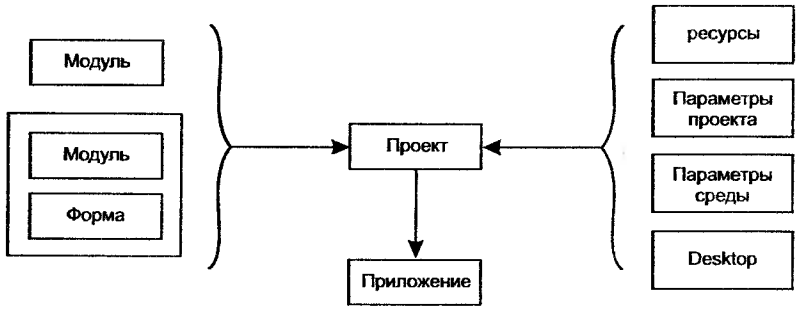


Рис. 4.7. Структурная схема приложения Delphi

Помимо перечисленных файлов в проект могут входить и дополнительные модули — файлы с расширением .pas (рис. 4.7).

Файлы проекта

Файлы проекта по умолчанию располагаются в одном каталоге. Если главный файл проекта сохраняется под другим именем, то это имя получают и файлы с расширениями .res, .dof, .cfg и .dsk.

При запуске Delphi автоматически создается новый проект с именем Project1.dpr, который имеет в своем составе форму Form1.dfm и соответствующий ей модуль Unit1.pas.

Главный файл проекта. При выполнении операций с проектом код файла проекта (программы) формируется средой Delphi автоматически:

```
Program Project1;          // Имя программы
Uses
          // Далее следует перечисление используемых модулей
  Forms,   // Имя подключаемого модуля
  Unit1 In 'Unit1.pas' {Form1};
                  // Перечисление модулей всех форм проекта
    {$R *.RES}    // Директива подключения
                  // к проекту файла ресурсов
```



```
Begin                          // Начало блока программы
   Application.Initialize;  // Инициализация приложения
   Application.CreateForm(TForm1,Form1);  // Создание формы
   Application.Run;                       // Запуск приложения
End.                                      // Конец блока программы
```

Служебное слово **Uses** сообщает компилятору, какие модули должны быть подключены при построении приложения. В приведенном примере подключается библиотечный модуль Forms и модуль с исходным кодом формы Unit1.pas. Имя формы (Form1) указано в виде комментария. Если проект содержит несколько форм, перечисляются модули всех форм проекта.

Просмотреть и отредактировать код файла в окне Редактора Кода можно с помощью команды Project/View Source (Проект/Просмотр исходного текста).

Подключаемый Файл Ресурсов имеет имя, совпадающее с именем файла проекта.

Файлы формы. Для каждой формы автоматически создаются Файл Описания, первоначально имеющий имя Unit1.dfm, и Файл Модуля Unit1.pas (файлы модуля формы и описания формы имеют всегда одинаковое, но отличающееся от имени файла проекта, имя).

Файл Описания формы (*.dfm) — текстовый файл, содержащий параметры формы и ее компонентов. При конструировании формы в Файл описания автоматически вносятся соответствующие изменения.

Чтобы отобразить содержание этого файла на экране, необходимо активизировать команду контекстного меню Окна Формы View As Text (или нажатием клавиш <Alt+F12>).

Окно Редактора Кода и его содержимое будет доступно для просмотра и редактирования описания формы (рис. 4.8).

Переключиться в режим формы можно с помощью команды контекстного меню View As Form (или клавишами <Alt+F12>).

Чтобы открыть окно любой формы проекта для конструирования, необходимо выбрать ее в диалоговом окне, появляющемся по команде Главного Меню View\Forms (рис. 4.9).

Файл модуля формы (*.pas) создается автоматически при добавлении новой формы и содержит описание класса формы (состав и поведение компонентов и функционирование обработчиков событий).

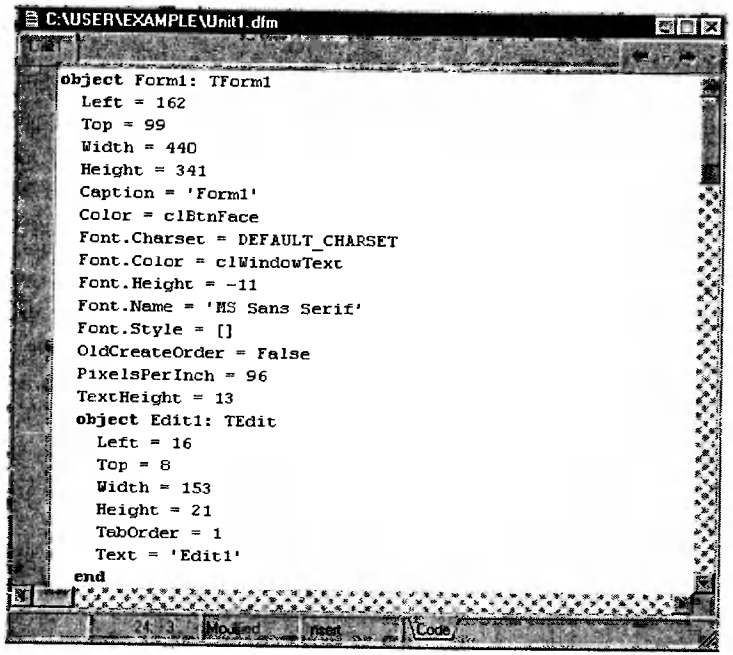


Рис. 4.8. Содержимое Файла Описания формы

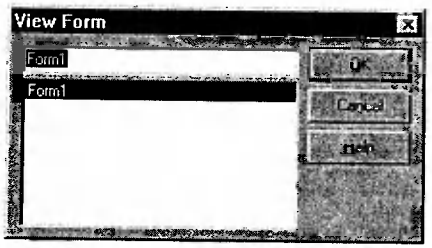


Рис. 4.9. Выбор формы для конструирования

В процессе конструирования формы (при размещении на форме компонентов) в модуль формы вносятся соответствующие изменения, причем изменения в описание класса вносятся автоматически, а процедуры обработки событий кодируются разработчиком.

Открыть модуль формы можно либо с помощью команды меню File\Open, либо в диалоговом окне команды View\Units (Просмотр\Модули), выбрав нужный модуль.

Тексты модулей форм отображаются в окне Редактора кода и редактируются с его помощью.

Файлы модулей. Помимо файлов, создаваемых средой Delphi, в проект могут включаться файлы, не связанные с формами. Например, константы, переменные, процедуры и функции, общие для нескольких модулей проекта, целесообразно оформить в виде отдельного модуля и подключать его по мере необходимости. Такие модули оформляются по правилам языка программирования Object Pascal и сохраняются в отдельных файлах с расширением .pas.

Для того чтобы модуль в дальнейшем мог быть использован другим модулем или проектом, его имя должно быть указано в разделе **Uses** этого модуля или проекта как имя подключаемого модуля.

Файл Ресурсов. При первом сохранении проекта автоматически создается Файл Ресурсов с именем, совпадающим с именем Файла Проекта, и расширением .res.

Файл Ресурсов имеет иерархическую структуру, в которой ресурсы разбиты на группы, а каждый ресурс имеет уникальное в пределах группы имя. Имя ресурса задается при его создании и в последующем используется в приложении для доступа к этому ресурсу.

В файле ресурсы разбиты на группы. Каждая группа имеет имя.

Файл содержит следующие ресурсы:

- пиктограммы;
- растровые изображения;
- курсоры.

Первоначально Файл ресурсов содержит пиктограмму проекта. Ее можно изменить, используя Редактор Изображений (Image Editor). Вызывается редактор командой Tools\Image Editor (Инструменты\Редактор Изображений).

Редактор Изображений позволяет обрабатывать файлы четырех видов. Три из них объединяют файлы, содержащие ресурсы, определенные выше:

- растровые изображения (*.bmp);
- пиктограммы приложений (*.ico);
- курсоры (*.cur).

К последнему (четвертому) виду относятся файлы, имеющие формат откомпилированных файлов-ресурсов (файлы с расши-

рением .res), которые могут в свою очередь содержать ресурсы предыдущих трех видов.

На рис. 4.10 показано окно Редактора, в которое загружен файл, и выполняется редактирование пиктограммы приложения.

Пиктограмма проекта находится в группе Icon и по умолчанию имеет имя mainicon.

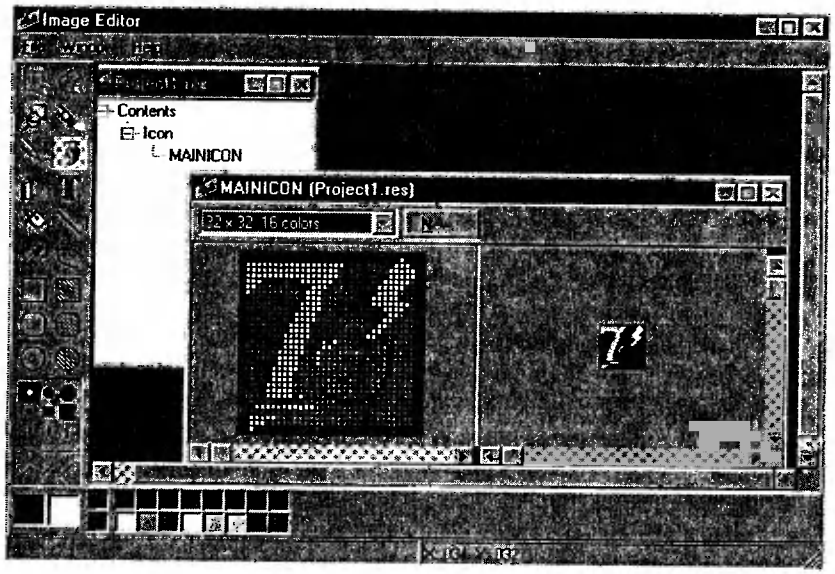


Рис. 4.10. Пиктограмма проекта в окне Редактора Изображений

Файл Параметров Проекта содержит конфигурационные установки, использующиеся при компиляции приложения, такие, как директории для поиска файлов проекта, текущие установки директив компиляции.

Например, если конфигурационный файл содержит следующие строки:

```
-$D+
-$I+
-U"C:\ DELPHI\UNITS",
```

это означает, что проект будет содержать коды отладчика ($D+), в проект будет включена автоматическая генерация результатов выполнения процедур ввода-вывода ($I+), и при построении

приложения для поиска модулей будет использоваться директория c:\delphi\units.

Для установки параметров проекта используется страницы Forms и Application окна Параметров Проекта (Project Options), которое открывается командой Главного Меню Project\Options (Проект\Параметры) (рис. 4.11).

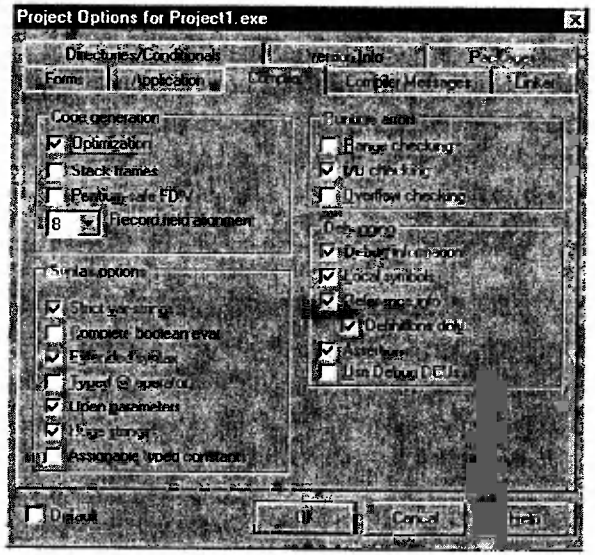


Рис. 4.11. Окно установки параметров проекта

Файл Параметров Среды представляет собой текстовый файл, который содержит текущие установки для параметров проекта, таких, как:

- настройки компилятора и компоновщика;
- имена служебных каталогов;
- директивы условий компиляции;
- параметры командной строки.

Для установки параметров проекта используется диалоговое окно, вызываемое с помощью команды меню Project\Options. Параметры разбиты на группы, каждая из которых располагается на соответствующей странице (рис. 4.11). После установки отдельных параметров Delphi автоматически вносит нужные изменения в файл параметров среды, представляя информацию в виде текстовых строк.

Например, к проекту на стадии разработки имеет смысл подключать отладочную информацию. Для этого необходимо установить опцию Debug Information на странице Compiler.

Файл Настроек Рабочей Области Среды содержит настройки рабочей области для текущего проекта, например информацию о том, какие окна открыты, где находится курсор и т. п. Такая информация позволяет восстановить состояние рабочей области при каждом новом открытии проекта в среде.

Чтобы обеспечить автоматическое создание и сохранение Файла Настроек, необходимо:

- с помощью команды меню Tools\Environment Options открыть диалоговое окно (рис. 4.12);
- на странице Preferences окна в разделе Autosave options установить параметр Project Desktop.

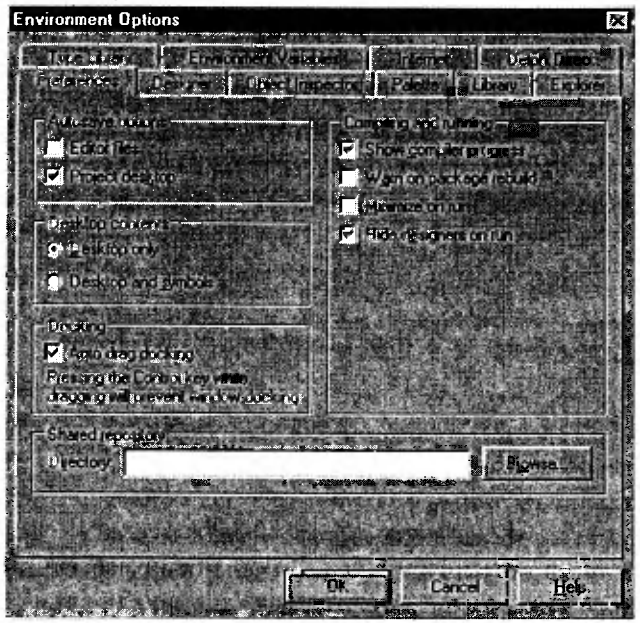


Рис. 4.12. Окно настроек среды

Среда Delphi обновляет Файл Настроек Рабочей Области всякий раз при закрытии проекта. Файл хранится в той же директории, что и Главный Файл Проекта, и имеет то же имя.

При новой загрузке проекта внешний вид среды восстанавливается, т. е. становится таким же, как при предыдущем закрытии проекта.

При создании нового проекта опция автоматического сохранения настроек проекта по умолчанию всегда включена.

Резервные файлы. Среда Delphi может создавать резервные копии главного файла проекта и файлов модулей и описаний форм. Резервные копии файлов содержат в расширении «тильду» (~) в качестве первого символа и создаются при повторном сохранении проекта для тех файлов, в исходном коде которых были сделаны изменения:

- *.~dp — резервная копия главного файла проекта;
- *.~pa — резервная копия модуля;
- *.~df — резервная копия файла описания формы.

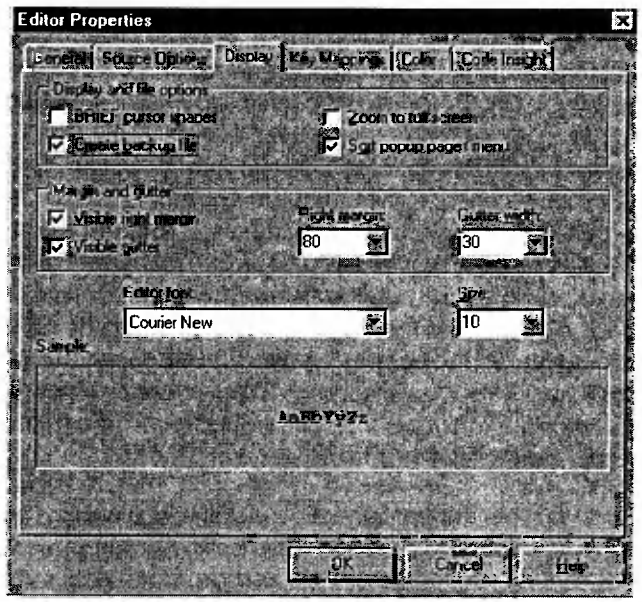


Рис. 4.13. Установка параметров редактирования

Задать режим сохранения резервных копий можно с помощью диалогового окна команды меню Tools\Editor Options, установив на странице Display опцию Create Backup File (рис. 4.13).

Компиляция и выполнение проекта

Компиляция проекта. Компиляция может быть выполнена на любой стадии разработки проекта и позволяет проверить внешний вид интерфейсных окон и правильность функционирования фрагментов создаваемого кода.

Запуск процесса компиляции выполняется по команде меню `Project\Compile <Project Name>`. В строке меню команды указано имя проекта, разработка которого выполняется в текущий момент. При сохранении проекта под другим именем, соответственно, должно измениться имя проекта в меню.

При компиляции происходит следующее:

- компиляция файлов всех модулей, содержимое которых изменилось после последней компиляции, и модулей, использующих их с помощью директивы `uses` (в результате создаются файлы с расширением `*.dcu`);
- создание исполняемого файла-приложения: в процессе компиляции проекта создается готовый к выполнению файл (`*.exe`) или динамически загружаемая библиотека (`*.dll`).

Само приложение является автономным и не требует при своей работе дополнительных файлов Delphi.

Имя приложения совпадает с именем файла проекта.

Ход процесса компилирования (рис. 4.14) будет отображаться на экране, если установить опцию `Show compiler progress` в меню `Tools\Environment Options\Preferences`.

Сборка проекта. Сборка проекта выполняется командой `Project\Build <Project Name>`. При сборке перекомпилируются все модули, входящие в проект, независимо от того, были в них внесены изменения или нет.

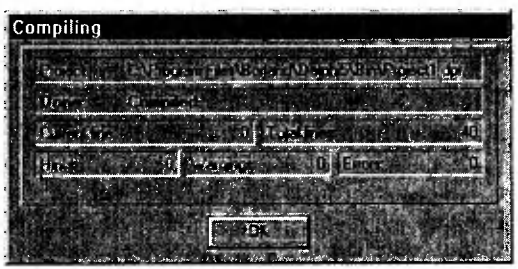


Рис. 4.14. Отображение процесса компиляции

Запуск проекта. Запускать проект можно из среды Delphi и из среды Windows.

Запуск проекта из среды Delphi выполняется командой `Run\Run`. Созданное приложение начинает свою работу. Если в модули программы были внесены изменения, то перед запуском проекта выполняется его компиляция.

При запуске проекта в среде Delphi необходимо учитывать следующие особенности:

- нельзя запускать вторую копию приложения;
- вносить изменения в модули (т. е. продолжить разработку проекта) можно только после завершения работы приложения;
- если приложение не может нормально завершить работу, необходимо выполнить завершение средствами Delphi с помощью команды меню `Run\Program Reset`.

Запуск проекта из среды Windows осуществляется так же, как и запуск любого другого приложения.

Отладка приложения. Для отладки приложений в среде Delphi можно использовать встроенную систему отладки. Проект в этом случае должен быть откомпилирован с отладочной информацией. Подключение отладчика происходит через установку опции `Debug Information` в окне параметров проекта (см. рис. 4.11).

Встроенный отладчик (Debugger) облегчает поиск и устранение ошибок в приложениях.

Средства отладчика доступны:

- с помощью команд пункта меню `Run` (`Выполнение`);
- через подменю `View\Debug Windows` (`Окна Просмотр\Отладка`).

Система отладки предусматривает следующие действия:

- выполнение до указанного оператора (строки кода);
- пошаговое выполнение приложения;
- включение и выключение точек останова;
- выполнение приложения до точки останова;
- просмотр значений данных в окнах просмотра;
- установку новых значений данных при выполнении приложения.

Установка параметров отладчика выполняется с помощью команды `Tools\Debugger Options` (`Параметры отладчика`) (рис. 4.16).

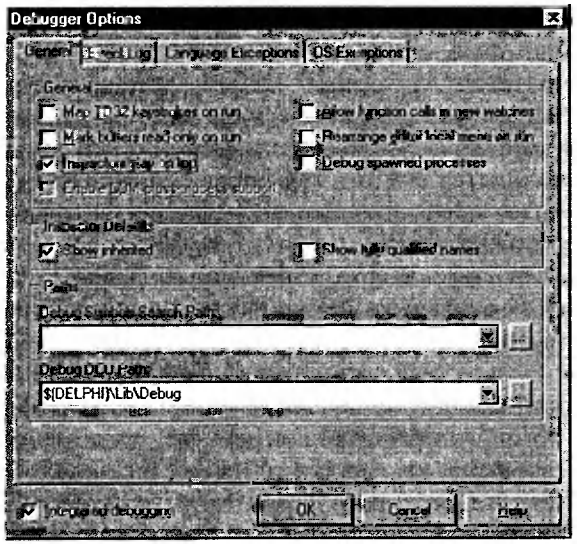


Рис. 4.15. Окно параметров отладчика

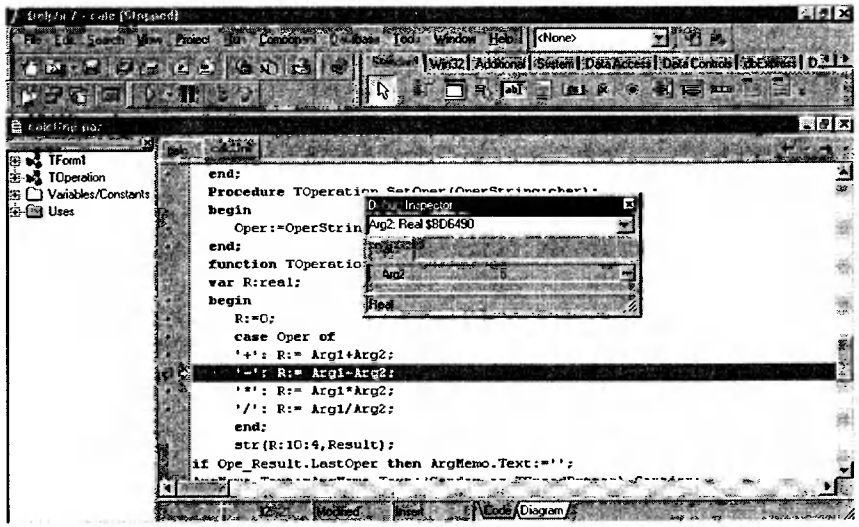


Рис. 4.16. Просмотр значения переменной в окне отладки

Если, например, в окне параметров отладчика установлена опция `Inspectors stay on top` (см. рис. 4.15), то окно Инспектора отладки будет видно всегда. На рис. 4.16 показано

окно инспектора отладки со значением переменной `Arg2` в момент временного прекращения работы приложения в точке останова (строке текста, выделенного на рисунке жирной точкой).

4.4. Средства управления параметрами проекта и среды разработки

Отображение и установка текущих параметров среды выполняются в диалоговом окне `Environment Options` (см. рис. 4.12), которое вызывается посредством команды меню `Tools\ Environment Options`. Параметры разбиты на группы, каждая группа размещена на отдельной странице окна.

Возможна настройка следующих групп параметров (страниц):

- `Preferences` — параметры конфигурации рабочего пространства среды;
- `Designer` — параметры Дизайнера Форм проекта;
- `Environment Variables` — просмотр и установка системных переменных, создание, редактирование и удаление пользовательских переопределений;
- `Object Inspector` — спецификация свойств Инспектора Объектов;
- `Library` — спецификация директорий размещения и параметров компиляции и компоновки библиотек проекта;
- `Palette` — параметры настройки содержимого страниц Палитры Компонентов;
- `Explorer` — параметры настройки Обозревателя Проекта (`Project Browser`) и Навигатора Проекта (`Code Explorer`);
- `Type Library` — параметры настройки редактора Библиотеки Типов (набора информации о типах объектов);
- `Internet` — типы файлов и параметры скриптов для приложений `WebSnap`;
- `Delphi Direct` — параметры доступа по сети к последним новостям Delphi на сайте `borland.com`.

Параметры среды для каждого проекта сохраняются в файле конфигурации с расширением `.cfg`.

Менеджер Проектов (Project Manager)

В состав интегрированной среды разработки приложений включен Менеджер Проектов, управляющий одновременно несколькими проектами. Окно Менеджера открывается по команде меню View\Project Manager.

Менеджер Проектов работает с группой проектов, в которой может быть один или несколько проектов. Новый проект может быть добавлен в группу с помощью контекстного меню, когда выделена строка ProjectGroup1.

Менеджер Проектов отображает иерархическую структуру группы проектов, самих проектов, а также все формы и модули, входящие в каждый проект. Для работы с проектами в группе можно использовать либо панель инструментов, либо контекстное меню, набор команд которого зависит от того, какой объект выделен в группе. На рис. 4.17 показано окно Менеджера Проектов с контекстным меню управления приложением, позволяющим:

- добавить модуль в проект;
- удалить модуль из проекта;
- сохранить проект;
- изменить параметры проекта;
- откомпилировать проект и построить приложение и т. п.

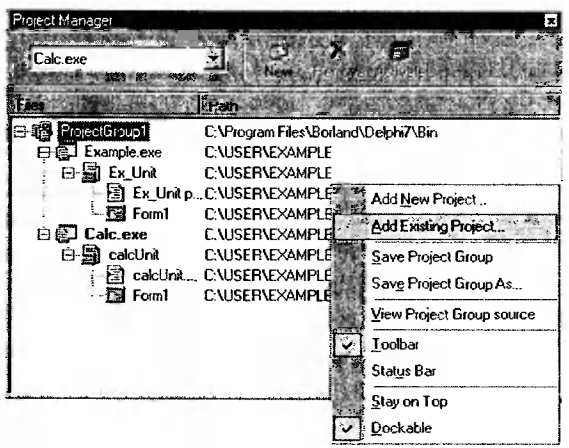


Рис. 4.17. Окно Менеджера Проектов

Менеджер позволяет объединять проекты, которые работают вместе, в одну проектную группу. Группы проектов сохраняются в файлах с расширением .bpg (Borland Project Group).

Обозреватель Проекта (Project Browser)

С помощью Обозревателя Проекта разработчик приложения может просматривать перечень модулей, классов, типов, свойств, методов, переменных и процедур, объявленных в проекте или используемых им (рис. 4.18). Окно Обозревателя Проекта открывается по команде меню View\Browser. Перед использованием Обозревателя проект должен быть сохранен и откомпилирован.

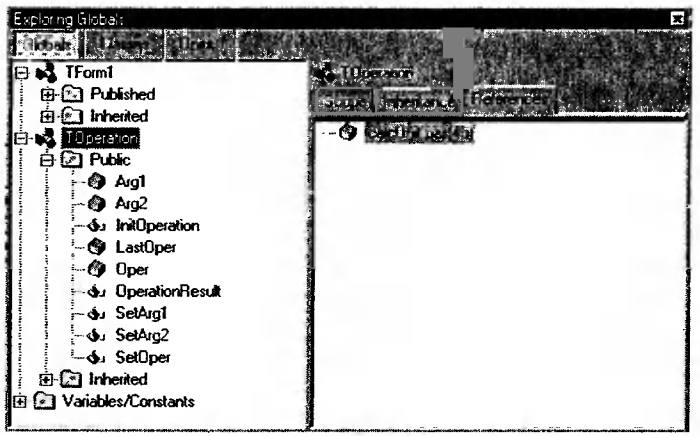


Рис. 4.18. Окно Обозревателя Проекта

Окно Обозревателя проекта разделено на две части. Слева в иерархическом виде отображаются доступные объекты, а справа для выбранного объекта более детально отображаются его характеристики.

Для просмотра в левой части окна доступны иерархии трех типов объектов, расположенные на различных страницах:

- Globals (Глобальные объявления);
- Classes (Классы объектов);
- Units (Модули).

В правой части окна доступны для просмотра следующие характеристики:

- Scope — список идентификаторов, объявленных в классе или модуле, выделенном в левой части окна;
- Inheritance — иерархическое дерево выделенного в левой части класса;
- References — имена файлов и номера строк программного кода текущего проекта, где есть ссылка на выделенный в левой части объект.

Для управления параметрами отображения объектов в Обозревателе используется команда меню Tools\Environment Options\Explorer.

С помощью Обозревателя Проекта можно перемещаться по списку используемых проектом модулей и просматривать символы в разделах Interface или Implementation; можно просматривать глобальные объявления и переходить к ссылкам на них в исходном коде.

Репозиторий Объектов

Репозиторий (хранилище) Объектов используется для создания новых элементов любого типа: форм, модулей данных, библиотек, компонентов и др. Объекты Репозитория рассматриваются в качестве шаблонов при разработке приложений.

Окно Репозитория открывается по команде меню File\New\Other... . Объекты-шаблоны сгруппированы. Каждая группа объектов размещена на отдельной странице:

- New — базовые объекты;
- ActiveX — объекты ActiveX и OLE;
- Multitier — объекты многопоточного приложения;
- Projects — проекты;
- Form — формы;
- Dialogs — диалоги;
- Data Modules — модули данных;
- Business — мастера форм.

Страница с названием Example (рис. 4.19) содержит форму текущего проекта. При добавлении или удалении формы проекта ее шаблон соответственно добавляется или исключается из Репозитория Объектов.

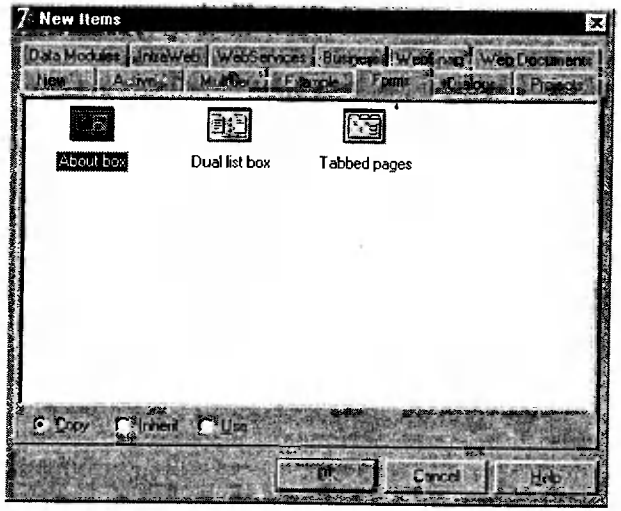


Рис. 4.19. Окно Репозитория Объектов

Разработанный и отлаженный проект тоже может быть помещен в Репозиторий. Для этого необходимо инициировать команду меню Project\Add to Repository и заполнить появившееся диалоговое окно. Точно так же, как и шаблоны проектов, можно добавлять в Репозиторий и шаблоны форм. Для этого необходимо выбрать форму, которую предстоит добавить в Репозиторий, и инициировать команду контекстного меню Add to Repository.

Добавить в текущий проект объект из Репозитория можно одним из трех способов:

- Copy — в проект добавляется копия выделенного объекта Репозитория. Все изменения в объекте являются локальными и не затрагивают оригинал;
- Inherit — в проект добавляется новый объект, который наследует свойства выделенного объекта Репозитория. При каждой новой компиляции проекта все изменения, внесенные в шаблон Репозитория, будут отражены в проекте;
- Use — объект Репозитория становится частью проекта. Изменения, вносимые в объект в рамках разработки проекта, на самом деле вносятся непосредственно в шаблон Репозитория.

Справочная система

В состав справочной системы Delphi входят:
- стандартная система справки;
- справочная помощь через Internet;
- контекстно-зависимая справочная помощь.

Окно стандартной системы справки (вызывается с помощью команд меню `Help\Delphi Help`, `Help\Delphi Tools`, `Help\Windows SDK`) состоит из трех страниц (рис. 4.20):
- информация на странице `Содержание` представлена иерархией тематических разделов и подразделов;
- `Предметный указатель` обеспечивает доступ к справочной информации через словарь ключевых терминов;
- страница `Поиск` предназначена для проведения полнотекстового поиска в справочном массиве и отображения всех разделов справочной системы, в которых встречается указанная фраза или слово.

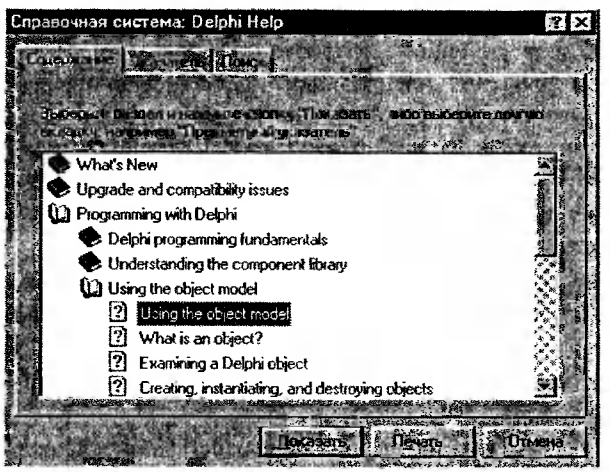


Рис. 4.20. Окно справки

Команды меню, поддерживающие справочную помощь через Internet (например, `Help\Borland Home Page`, `Help\Delphi Home Page`), обеспечивают доступ на соответствующий сайт средствами браузера Internet.

Контекстно-зависимая справочная помощь вызывается путем нажатия клавиши `<F1>` функциональной клавиатуры. Содержимое справочной информации зависит от текущего состояния среды и связано с активизированным в текущий момент объектом.

4.5. Разработка приложений

В соответствии с порядком разработки приложений с графическим интерфейсом необходимы последовательно два взаимосвязанных этапа: проектирование функционального интерфейса приложения и программирование процедур обработки событий, возникающих при работе пользователя с приложением.

Создание интерфейса приложения

В процессе этого (первого) этапа разработки приложения происходит последовательное размещение в `Окне Формы` компонентов, наилучшим образом соответствующих функциональному назначению приложения. Для установки параметров размещения компонента на форме используется окно `Инспектора Объектов`.

Чтобы поместить нужный компонент на форму, необходимо выбрать его из `Палитры Компонентов` и указать его местоположение на области формы. После размещения компонента на форме можно изменять с помощью мыши его положение и размеры.

По умолчанию компоненты выравниваются на форме по линиям сетки, которая при проектировании отображается на поверхности формы.

Выделение нескольких компонентов на форме выполняется с помощью мыши при нажатой клавише `<Shift>`.

Редактировать размещение компонентов можно с помощью контекстного меню или группы команд меню `Edit`, например:
- `Align` — выравнивание группы компонентов;
- `Bring to front` — перевод компонента на передний план;
- `Send to Back` — перевод компонента на задний план;
- `Size` — установка новых размеров компонента.

Следующий шаг — определение свойств компонентов при проектировании.

По типам хранящихся в них данных свойства делятся на следующие группы:

- *простые* — свойства, значения которых являются числами или строками (например, Caption, Name, Left, Top);
- *перечисляемые* — свойства, которые могут принимать значения из предложенного набора/списка (например, свойства Visible и Enabled могут принимать только значение true или false);
- *множества* — свойства, значения которых представляют собой комбинацию значений из предлагаемого множества (например, свойство формы BorderIcon);
- *объекты* — свойства, значения которых представляют собой объекты. В области значения свойства-объекта в скобках указывается тип объекта (например, свойство Font типа TFont).

Управлять свойствами компонента можно непосредственно в окне Конструктора Формы или с помощью Инспектора Объектов.

Для доступа к свойствам компонента через окно Инспектора Объектов необходимо сначала в раскрывающемся списке верхней части окна, где отображаются название компонента и его тип, выбрать нужный компонент. В *левой* части окна Инспектора Объектов приводятся названия всех свойств выбранного компонента, которые доступны на этапе разработки приложения. Для каждого свойства *справа* содержится значение этого свойства.

Чтобы изменить значения свойств, необходимо выбрать изменяемое свойство и, в зависимости от типа данных свойства, выполнить следующие действия:

- в случае простого свойства ввести в правой колонке Инспектора Объектов новое значение;
- в случае перечисляемого свойства открыть список, появившийся в соответствующей ячейке правой колонки, и выбрать нужное значение;
- в случае множества установить значение true на выбираемых элементах предлагаемого множества;
- в случае объекта заполнить отдельно значение каждого поля объекта или воспользоваться вспомогательными диалоговыми окнами задания значений.

Подтверждается изменение свойства нажатием клавиши <Enter>, отменяется — нажатием клавиши <Esc>. Если для

свойства введено неправильное значение, то выдается предупреждающее сообщение.

Результат изменения свойств компонента сразу отображается в окне Конструктора Формы.

На рис. 4.21 показано изменение простого свойства формы Caption (введено значение Новая форма), перечисляемого

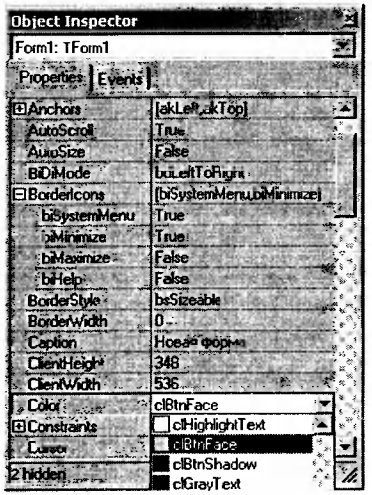


Рис. 4.21. Изменение свойств формы

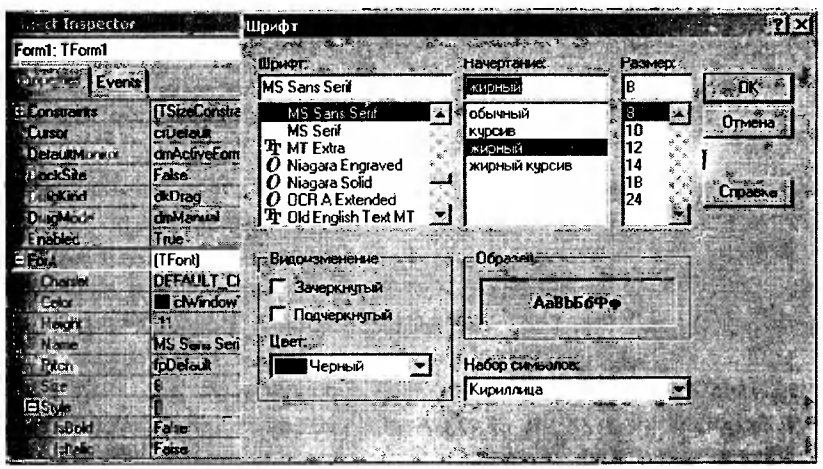


Рис. 4.22. Изменение свойства Font

свойства Color и свойства типа Множество — BorderIcon. На рис. 4.22 представлено окно заполнения свойства типа Объект — установка шрифта (свойство Font).

При выполнении приложения свойства компонентов можно изменять с помощью операторов присваивания. Например, изменить заголовок формы можно программно путем оператора присваивания:

```
Form1.Caption:= 'Новая форма';
```

Создание функциональной схемы работы приложения

На втором этапе разработки приложения для каждого компонента, размещенного на форме, разработчик должен определить нужную реакцию на те или иные действия пользователя (например, нажатие кнопки или выбор переключателя).

Функциональная схема работы приложения определяется процедурами, которые выполняются при возникновении определенных событий, происходящих при взаимодействии пользователя с управляющими элементами формы. Реакция на события присуща каждой форме и не зависит от назначения приложения и его особенностей.

Каждый компонент имеет свой набор событий, на которые он может реагировать. Вместе с тем, существуют две категории стандартных событий, определенных для всех визуальных компонентов:

- события, определенные для всех без исключения визуальных компонентов (табл. 4.1);
- события, характерные только для оконных визуальных компонентов (табл. 4.2).

Процедура, связанная с несколькими событиями для различных компонент, называется *общим обработчиком* и вызывается при возникновении любого из связанных с ней событий.

Для создания процедуры обработки события нужно:

- выделить на форме компонент;
- перейти на страницу событий Инспектора Объектов;
- выделить событие, для которого будет создаваться процедура-обработчик события;

Таблица 4.1. **События, общие для всех визуальных компонентов**

Событие	Момент возникновения и параметры процедур
OnClick	Возникает при нажатии левой клавиши мыши в области компонента или при нажатии клавиши <Enter>, если на компонент установлен фокус
OnDblClick	Возникает при двойном нажатии левой клавиши мыши в области компонента
OnDragDrop	Возникает в случае, когда пользователь «отпустил» перетаскиваемый объект. Параметр Source соответствующей процедуры-обработчика содержит указатель на объект, который был «отпущен», а параметр Sender — на объект, принявший «перетаскиваемый»
OnDragOver	Возникает в случае, когда пользователь «перетаскивает» объект для его размещения в рамках другого компонента
OnEndDrag	Возникает в конце процедуры «перетаскивания» объекта. Параметр Sender соответствующей процедуры-обработчика указывает на «перетаскиваемый» объект, а параметр Target — на его образ в рамках принимающего компонента
OnMouseDown	Возникает при нажатии любой клавиши мыши в области компонента. Параметр Button соответствующей процедуры-обработчика этого события позволяет определить, какая клавиша была нажата, параметр Shift — были ли нажаты клавиши <Shift>, <Ctrl> или <Alt> вместе с клавишей мыши, а параметры X и Y содержат координаты курсора мыши в момент нажатия клавиши
OnMouseMove	Возникает при «буксировке» манипулятора «мышь» (т. е. при его перемещении при нажатой левой клавише). Параметр Shift соответствующей процедуры-обработчика определяет, были ли нажаты клавиши <Shift>, <Ctrl> или <Alt> вместе с клавишей мыши, а параметры X и Y содержат координаты курсора мыши в момент нажатия клавиш.
OnMouseUp	Парное для OnMouseDown. Возникает, когда ранее нажатая клавиша мыши отпущена. Имеет те же параметры, что и OnMouseDown

- посредством двойного нажатия кнопки мыши в области значения события получить доступ в модуль формы, где Delphi автоматически создаст заготовку процедуры-обработчика;
- в месте, где будет установлен текстовый курсор, написать код, который должен выполняться при возникновении события.

Итак, для обеспечения выполнения функций приложения необходимо:

- задать в Инспекторе Объектов значения свойств и процедур обработки событий;

Таблица 4.2. **События оконных визуальных компонентов**

Событие	Момент возникновения и параметры процедуры
OnEnter	Возникает, когда компонент получает фокус ввода
OnExit	Возникает при потере фокуса компонентом
OnKeyDown	Возникает при нажатии любой клавиши на клавиатуре. Параметр Key соответствующей процедуры-обработчика имеет тип Word и может содержать коды виртуальных клавиш. Параметр Shift передает информацию о нажатии клавиш <Shift>, <Ctrl>, <Alt> и о нажатии клавиш мыши
OnKeyPress	Возникает при нажатии клавиши на клавиатуре. Параметр Key соответствующей процедуры-обработчика имеет тип Char и содержит ASCII-код нажатой клавиши. Для клавиш, которые не имеют ASCII-кодов (например, <Shift>, <Ctrl> или <Alt>), событие не возникает
OnKeyUp	Парное для OnKeyDown. Возникает, когда пользователь отпускает нажатую ранее клавишу. Имеет те же параметры, что и OnKeyDown

- написать программный код для заданных процедур обработки событий.

Создание простейшего приложения

Рассмотрим процесс создания простейшего приложения для иллюстрации последовательности действий при разработке.

Пусть форма нашего приложения содержит окно для редактирования строки текста и кнопку, нажав на которую пользователь должен получить в окне редактирования текст: Ура!!! Приложение работает!!!.

С помощью команды меню File\New Application создается главный файл проекта (имя по умолчанию Project1) и заготовка, содержащая главную форму приложения, для которой уже автоматически построено описание (Unit1.dfm) и модуль (Unit1.pas). Форма содержит основные элементы окна Windows: заголовок Form1, кнопки минимизации, максимизации и закрытия окна и кнопку вызова системного меню.

Построение интерфейса. Разместим на форме компонентов редактирования строки — Edit класса TEdit (рис. 4.23). Компонент находится на странице Стандартные (Standard) Палитры Компонентов. Имя компонента по умолчанию Edit1 (следующий компонент этого же класса получит имя Edit2 и т. д.). Оставим имя компонента без изменения, а значение свойства компонента Text обнулим. Для этого необходимо очистить содержимое правой части строки Инспектора Объектов для свойства Text (по умолчанию значение этого свойства совпадает со значением свойства Name — имя компонента).

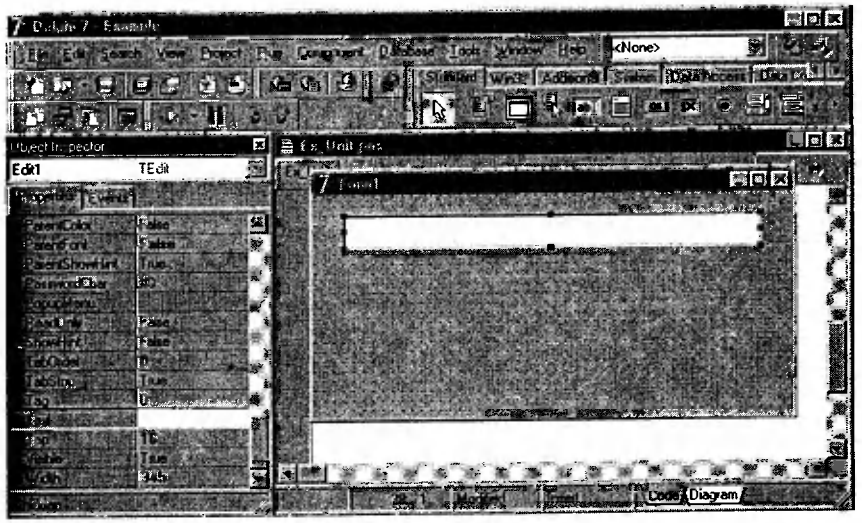


Рис. 4.23. Размещение на форме компонента Edit1

По умолчанию для текстовой строки, отображаемой в окне компонента Edit1, установлен размер шрифта, равный 8 пунктам. Чтобы сделать буквы крупнее, изменим свойство Font.Size (размер шрифта), задав новое значение, равное 12.

Следующий компонент приложения — кнопка. Разместим на форме кнопку класса TButton (страница «Стандартные» Палитры Компонентов). На форме компонент получит имя Button1. Изменим свойство кнопки Caption (Заголовок), установив значение Проверка работы приложения (рис. 4.24).

Построение интерфейса приложения завершено. Следующий шаг — определение функций приложения.

Функциональная схема приложения. Разрабатываемый пример должен продемонстрировать реакцию приложения на нажатие кнопки. Значит, необходимо разработать обработчик события OnClick для кнопки. Для этого переключимся в Инспекторе Объектов на страницу Events (События) для компоненты Button1. С помощью двойного нажатия левой клавиши мыши

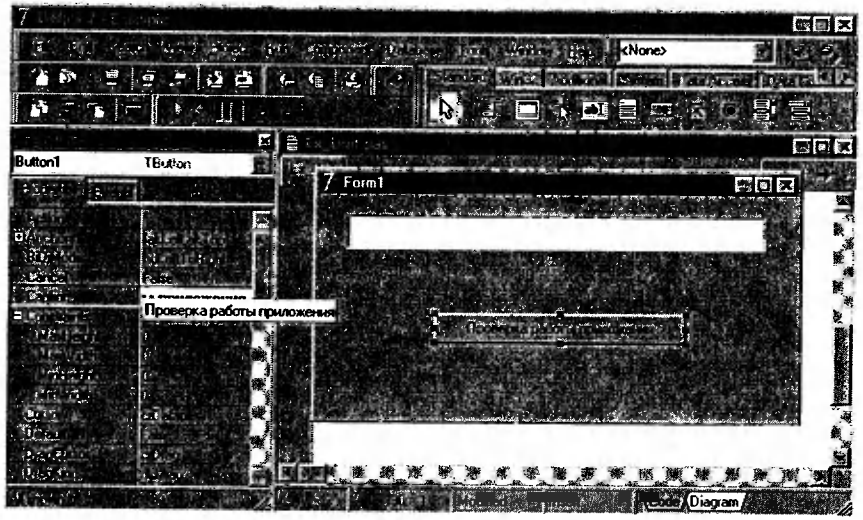


Рис. 4.24. Размещение на форме компонента Button1

на левой части строки события в Инспекторе объектов попадем в процедуру обработки события, заготовку для которой среда генерирует автоматически, и по положению текстового курсора введем оператор:

```
Edit1.Text := 'Ура!!! Приложение работает!!!';
```

Оператор присваивания в данном случае заполнит свойство Text компонента Edit1 изнутри приложения при возникновении события Нажатия на кнопку (рис. 4.25).

Разработка приложения завершена. С помощью команды меню File\Save Project As... сохраним результаты работы, присвоив Главному Файлу Проекта имя Example.dpr, а Модулю Формы — имя Ex_Unit.pas.

Код модуля формы с обработчиком события кнопки:

```
unit Ex_Unit;
interface
uses
     Windows, Messages, SysUtils, Classes, Graphics,
     Controls, Forms,Dialogs, StdCtrls;
type
     Form1 = class(TForm)
          Edit1: TEdit;
          Button1: TButton;
```

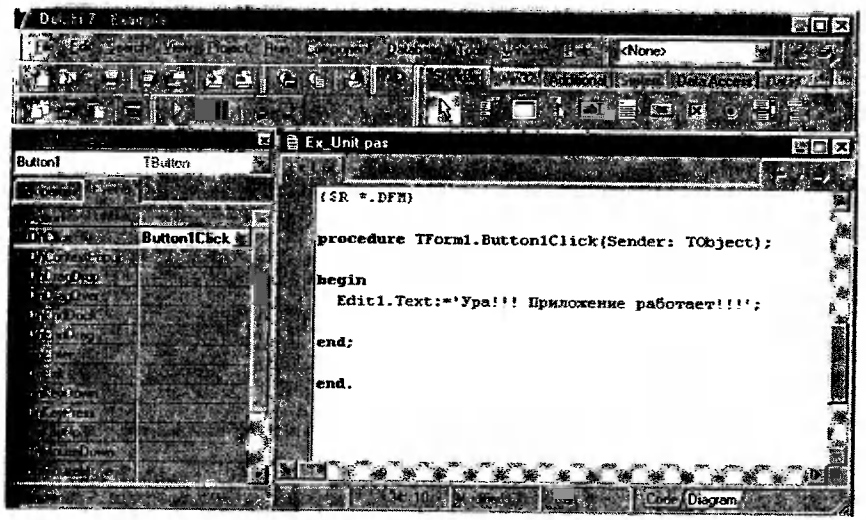



Рис. 4.25. Процедура обработки события OnClick

```
Procedure Button1Click(Sender: TObject);
          private
               { Private declarations }
          public
               { Public declarations }
     End;
     Var
          Form1: TForm1;
     Implementation
          {$R *.DFM}
Procedure TForm1.Button1Click(Sender: TObject);
          begin
               Edit1.Text:='Ура!!! Приложение работает!!!';
          end;
     end.
```

Для удаления процедуры-обработчика в дальнейшем достаточно удалить код, внесенный разработчиком самостоятельно. После этого при сохранении или компиляции модуля обработчик будет удален автоматически из файлов (dfm и pas) формы. При изменении в дальнейшем с помощью Инспектора Объектов имени компонента-кнопки или имени процедуры происходит автоматически переименование процедуры-обработчика в файлах формы.

Компиляция и запуск приложения. По команде меню Run\Run сначала выполняется компиляция проекта, а затем, в случае отсутствия ошибок компиляции, проект запускается на исполнение. На рис. 4.26 представлен результат работы приложения.

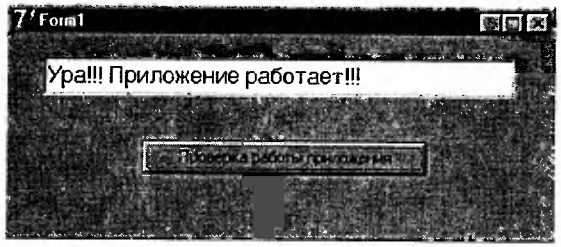


Рис. 4.26. Результат работы приложения Example

Pascal и визуальная среда разработки приложений Delphi

Среда разработки приложений Delphi и ее библиотеки базируются на языке программирования Object Pascal, разработанном фирмой Borland объектно-ориентированном расширении языка Pascal, основанном на введении понятия класса и объекта.

Рассмотрим в качестве примера задачу разработки приложения Калькулятор, реализующего четыре арифметических действия над вещественными числами.

Создание новых классов данных. Для решения задачи введем новый класс данных TOperation, описывающий выполнение арифметической операции над двумя аргументами.

Чтобы определить в Object Pascal новый класс данных, имеющий свои поля данных и свои методы, используется следующая синтаксическая конструкция:

```
TOperation = Class
     Arg1 : Real;
     Arg2 : Real;
     Oper : Char;
     LastOper : boolean;
     Procedure InitOperation;
     Procedure SetArg1(ArgString:string);
     Procedure SetArg2(ArgString:string);
     Procedure SetOper(InpOper:char);
     Function OperationResult:string;
End;
```


В приведенном описании поля Arg1 и Arg2 типа Real служат для хранения значений аргументов операции, поле Oper содержит символьное изображение операции ('+' — сложение, '-' — вычитание, '*' — умножение и '/' — деление), а логический флаг LastOper принимает значение true в момент задания операции.

Для класса определены следующие методы:

- InitOperation — инициализация объекта класса;
- SetArg1 — установка значения первого аргумента;
- SetArg2 — установка значения второго аргумента;
- SetOper — установка знака операции;
- OperationResult — выполнение операции.

Методы класса должны быть описаны в модуле программы. Чтобы подчеркнуть, что они являются частью класса TOperation, их названия предваряются именем класса (в качестве разделителя выступает точка):

```
Procedure TOperation.InitOperation;
Begin
     Arg1:=0;          // Обнуление первого аргумента
     Arg2:=0;          // Обнуление второго аргумента
     Oper:=#0;         // Очистка знака операции (# -
                       // признак символьного кода)
     LastOper:=false   // Флаг операции - false
End;
Procedure TOperation.SetArg1(ArgString:string);
     Var i:Integer;
     Begin
     val(ArgString,Arg1,i)  // Стандартная процедура
                            // преобразования типов:
                            // строка ArgString преобразуется
                            // в вещественное число Arg1.
                            // Параметр процедуры i - код
                            // ошибки преобразования
                            // (в целях упрощения алгоритма
                            // код ошибки не обрабатывается)
     End;
Procedure TOperation.SetArg2(ArgString:string);
     Var i:Integer;
     begin
          val(ArgString,Arg2,i)
     end;
Procedure TOperation.SetOper(InpOper:char);
     Begin
          Oper:= InpOper;   // Заполнение поля знака операции
     End;
```

```
Function TOperation.OperationResult:string;
     Var R:Real;
     Begin
          R:=0;
          // Оператор Case в зависимости от знака операции
          // обеспечивает выполнение арифметического действия.
          // Локальная переменная R служит для временного
          // хранения значения результата.
          Case Oper of
               '+': R:= Arg1+Arg2;
               '-': R:= Arg1-Arg2;
               '*': R:= Arg1*Arg2;
               '/': R:= Arg1/Arg2;
          End;
          str(R:10:4,Result);     // Стандартная процедура
                                  // преобразования типов.
          // Для примера используется назначение формата
          // представления вещественного числа с фиксированной
          // точкой: 10 - общее количество цифр числа;
          // 4 - количество цифр дробной части.
     End;
```

Создание объекта. После того как класс описан, можно создать объект и использовать его, например, таким образом:

```
Var
     Ope_Result:TOperation;
Begin
     // Создание объекта - общий для всех классов метод
     // Create:
     Ope_Result:=TOperation.Create;
     // Использование объекта:
     Ope_Result.SetArg1('500'); // аналогично действию:
     // Ope_Result.Arg1:= 500;
     Ope_Result.SetArg2('200'); // аналогично действию:
     //Ope_Result.Arg2:= 200;
     Ope_Result.SetOper('+');   // аналогично действию:
     //Ope_Result.Oper:= '+';
     // Вывод в виде сообщения результата выполнения операции
     ShowMessage('Результат выполнения операции =
     ' + Ope_Result.OperationResult);
     // Уничтожение объекта - общий для всех классов метод
     // Free:
     Ope_Result:=TOperation.Free;
End;
```

Object Pascal при объявлении переменной, имеющей тип класса, не создает автоматически объект, а использует так называемую ссылочную модель объекта. Основная идея этой

модели в том, что каждая переменная типа **Class** (как, например, Ope_Result) содержит не значение объекта, а лишь ссылку (указатель) на область памяти, в которой размещается объект. Поэтому, когда описывается переменная типа **Class**, то тем самым только резервируется место под ссылку на объект, а экземпляр объекта при этом должен создаваться вручную. Исключение составляют экземпляры компонентов графического интерфейса, которые помещаются на форму — они автоматически создаются средой Delphi.

Для создания экземпляра объекта необходимо вызвать его метод Create — конструктор класса TObject, свойства и методы которого наследуют все другие классы, как стандартные, так и создаваемые самостоятельно.

Если объект создан, то после использования он должен быть уничтожен. Эта операция выполняется с помощью метода Free, который также является методом наследуемого класса TObject.

Разработка приложения Калькулятор в среде Delphi

Вернемся к задаче разработки приложения Калькулятор. Теперь необходимо разработать форму, которая будет являться окном приложения в среде графического интерфейса ОС Windows.

Создание формы. Пусть на форме необходимо разместить следующие компоненты:

- окно для ввода значений аргументов и вывода результата;
- кнопки-цифры для формирования числа в окне при вводе значений аргументов;
- кнопки-знаки арифметических операций;
- кнопку «точка» для формирования вещественного числа с дробной частью;
- кнопку «равно» для получения конечного результата;
- кнопку удаления последнего введенного знака;
- кнопку очистки окна ввода значений аргументов и вывода результата.

Предлагаемый внешний вид формы представлен на рис. 4.27.

При создании формы окна приложения определяется тип класса новой формы (в нашем примере это класс TForm1), кото-

Рис. 4.27. Окно программы Калькулятор

рый наследует свойства и методы класса TForm и автоматически размещается объект Form1.

Исправим некоторые значения свойств в инспекторе объектов для формы Form1 (рис. 4.28):

- изменим свойство Caption — текст заголовка окна формы, задав значение Пример-калькулятор;
- исключим из системного меню окна возможности изменять внешний вид окна: для свойств biMinimize и

Рис. 4.28. Свойства формы в окне Инспектора Объектов

biMaximize установим значения false, запретив тем самым сворачивать окно до пиктограммы и разворачивать его на весь экран.

Окно для ввода значений аргументов и вывода результата представляет собой компонент класса Tmemo. Разместим компонент на области формы и изменим его свойства (рис. 4.29):

- установим в свойстве Name (имя объекта) значение ArgMemo;
- зададим свойству Align (размещение на форме) значение alTop, т. е. тем самым разместим компонент в верхней части формы и сделаем равным по ширине горизонтальному размеру формы;
- установим в свойстве Alignment значение taRightJustify, т. е. зададим принцип выравнивания введенного в окно значения по правой границе окна;
- установим в свойстве ReadOnly значение true, запретив ввод значений в окно с клавиатуры.

Рис. 4.29. Свойства окна ввода в Инспекторе Объектов

Следующий шаг разработки формы — размещение кнопок. Все кнопки представляются объектами, принадлежащими классу TSpeedButton. Для каждого объекта кнопки установим следующие свойства:

- Name (имя объекта): SB_0, SB_1,..., SB_9 — для кнопок-цифр; SB_point — для кнопки «точка»; SB_plus, SB_min, SB_dev, SB_mul, SB_eq — для знаков операций и «равно»; SB_del — для кнопки удаления знака; SB_clear — для кнопки очистки значения компоненты ArgMemo;

- Caption — текст на кнопке — символ цифры, точки, знака операции, 'C' и 'Del'.

Каждый из <Объектов-кнопок> будет служить источником событий, обработка которых и будет поддерживать процесс вычисления с помощью разрабатываемого калькулятора:

- при нажатии на <Кнопку «Цифра»> необходимо добавить к формируемому числу справа цифру, указанную на кнопке;
- при нажатии на <Кнопку «Точка»> необходимо добавить к формируемому числу справа точку, предварительно убедившись, что такого знака еще нет в формируемом числе;
- при нажатии на <Кнопку «Знак операции»> необходимо выполнить предыдущую операцию (в случае вычисления выражения типа A+B-C), отобразить ее результат и установить знак очередной операции;
- при нажатии на <Кнопку «Равно»> необходимо вычислить и отобразить результат;
- при нажатии на <Кнопку «Удаление знака»> (<'C'>) необходимо удалить из формируемого числа последний знак;
- при нажатии на <Кнопку «Очистка окна ввода»> (<'Del'>) необходимо очистить текстовое значение объекта ArgMemo.

Приведем описание процедур обработки событий.

Событие OnClick для <Кнопки «Цифра»> вызывает выполнение процедуры TForm1.SB_DigitClick:

```
Procedure TForm1.SB_DigitClick(Sender: TObject);
    Begin
        // Если последним был введен знак операции, то область
        // ArgMemo предварительно очищается для ввода нового
        // значения и происходит сброс флага ввода операции:
        If Ope_Result.LastOper Then
        Begin
            ArgMemo.Text:=''; Ope_Result.LastOper := false
        End;
        // Добавление цифры к числу справа
        ArgMemo.Text:=ArgMemo.Text+(Sender As
           TSpeedButton).Caption;
    End;
```

Процедура обработки события для <Кнопки «Точка»> (TForm1.SB_PointClick) отличается от процедуры SB_DigitClick тем, что, во-первых, обеспечивает ввод двух знаков ('0' и '.') в случае, если начинается ввод нового числа, и, во-вторых, запрещает ввод второй точки в изображение числа с десятичной дробью:


```
Procedure TForm1.SB_PointClick(Sender: TObject);
    Begin
        If Ope_Result.LastOper Then
        Begin
            ArgMemo.Text:=''; Ope_Result.LastOper := false
        End;
        If ArgMemo.Text = '' Then
            ArgMemo.Text:=ArgMemo.Text+'0.'
        Else
            If pos('.',ArgMemo.text)= 0 Then
            ArgMemo.Text:=ArgMemo.Text+'.';
    End;
```

Для обработки события OnClick <Кнопок «Знак операции»> предложим следующий алгоритм:

- полю Arg2 объекта Ope_Result присвоим значение, введенное в окне ArgMemo;
- проверим содержимое поля Oper. Если текущая операция — первая в выражении (т. е. Oper = #0), то перенесем содержимое Arg2 в Arg1. В противном случае выполним операцию, знак которой находится в поле Oper, и результат занесем в Arg1. Таким образом, после нажатия на <Кнопку «Знак операции»> операции в объекте Ope_Result устанавливается значение первого аргумента;
- занесем новое значение в поле Oper и установим флаг ввода операции.

Описанный алгоритм реализуется процедурой TForm1.SB_OperClick:

```
Procedure TForm1.SB_OperClick(Sender: TObject);
    Var s:shortstring;
    Begin
       With Ope_Result Do
       Begin SetArg2(ArgMemo.Text);
          If Oper <> #0 Then
          s:= OperationResult
          // выполнение операции и запись результата
          // в локальную переменную s
          Else s:=ArgMemo.Text;
          SetArg1(s); // Занесение значения первого аргумента
          SetArg2('0'); // Обнуление второго аргумента
          ArgMemo.Text:=s;// Вывод значения первого аргумента
          s:= (Sender As TSpeedButton).Caption;
```

```
            SetOper(s[1]); // Установка знака операции
                           // и флага ввода операции
            LastOper:=true;
        End;
    End;
```

Процедура обработки события **OnClick** для <Кнопки «Равно»> разрешает выполнение операции только в случае, если поле Oper не пусто, инициализирует объект Ope_Result и устанавливает флаг ввода операции:

```
Procedure TForm1.SB_EqClick(Sender: TObject);
    Begin
        With Ope_Result do
        Begin
            If Oper <> #0 then
            Begin SetArg2(ArgMemo.Text);
                ArgMemo.Text:=OperationResult;
                SetArg1(ArgMemo.Text);
            End;
            InitOperation;
            LastOper:=true;
        End;
    End;
```

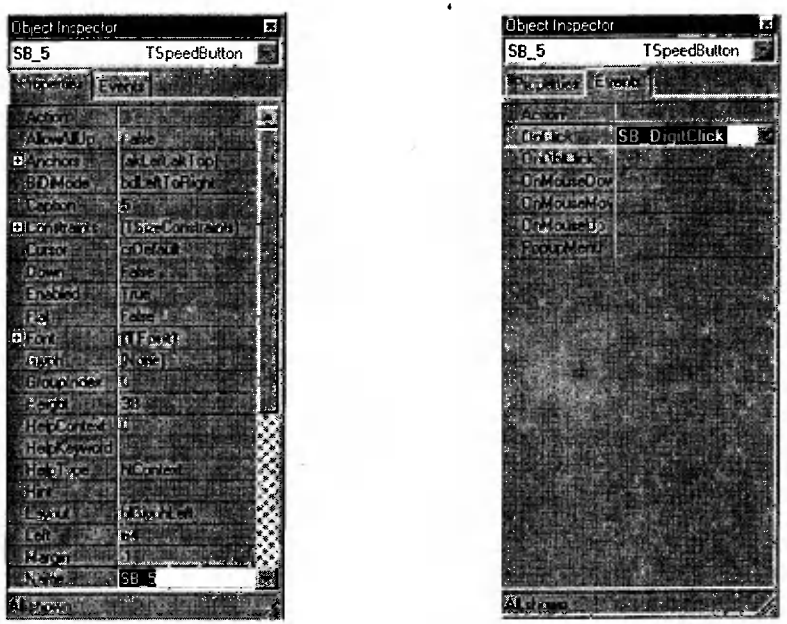
Для поддержки событий нажатия на кнопки <'C'> и <'Del'> опишем процедуры TForm1.SB_delClick:

```
Procedure TForm1.SB_delClick(Sender: TObject);
    Var s:String;
    Begin
      // Локальная переменная s используется для временного
      // хранения текстового значения окна ArgMemo
      s:= ArgMemo.text;
      SetLength(s,dec(Length(s));//Установка нового значения
                                 //длины строки (меньше на 1)
      ArgMemo.text:=s;
    End;
```

и TForm1.SB_ClearClick:

```
Procedure TForm1.SB_ClearClick(Sender: TObject);
    Begin
       ArgMemo.text:='';        // Очистка текстового значения
                                // окна ArgMemo
       Ope_Result.InitOperation; // Инициализация объекта
                                 // Ope_Result
    End;
```


На рис. 4.30 представлено окно Инспектора объектов для компоненты <Кнопка «5»>.



а *б*

Рис. 4.30. Свойства (*а*) и события (*б*) <Кнопки «5»> в Инспекторе Объектов

Заметим, что алгоритмы обработки событий строятся на использовании объекта Ope_Result спроектированного ранее класса TOperation, поэтому необходимо обеспечить размещение объекта в памяти и освобождение памяти при завершении работы приложения.

Для этих целей удобно использовать события формы OnCreate (создание формы) и OnClose (закрытие формы) (рис. 4.31). Опишем обработчики событий следующим образом;

```
Procedure TForm1.FormCreate(Sender: TObject);
    Begin
        Ope_Result:=TOperation.Create; // Создание объекта
                                       //Ope_Result
        Ope_Result.InitOperation;  // Инициализация объекта
    End;
```

```
Procedure TForm1.FormClose(Sender: TObject; var Action:
        TCloseAction);
    Begin
        Ope_Result.Free;  // Уничтожение объекта Ope_Result
    End;
```

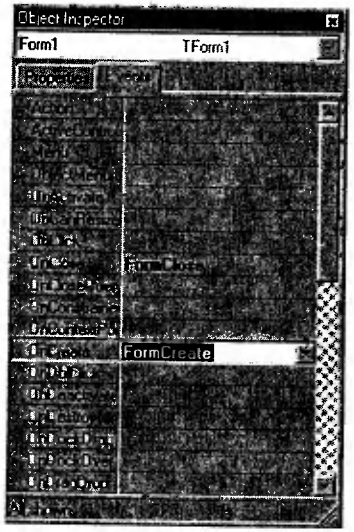


Рис. 4.31. События формы в Инспекторе Объектов

На этом разработку учебного приложения «Калькулятор» можно считать завершенной. Средой Delphi сгенерирован текст основной программы и построен модуль `calcUnit`, содержащий описание класса `TOperation` и интерфейсной формы `Form1`:

```
Program calc;
    Uses
        Forms,
        calcUnit in 'calcUnit.pas' {Form1};
        {$R *.res}
    Begin
        Application.Initialize;
        Application.CreateForm(TForm1, Form1);
        Application.Run;
    End.
    Unit calcUnit;
    Interface
    Uses
        Windows, Classes, Controls, Forms, Buttons,
        StdCtrls, ExtCtrls;
```



```
Type
    // Описание класса TForm1
TForm1 = class(TForm)
    SB_1: TSpeedButton;
    SB_2: TSpeedButton;
    SB_3: TSpeedButton;
    SB_4: TSpeedButton;
    SB_5: TSpeedButton;
    SB_6: TSpeedButton;
    SB_7: TSpeedButton;
    SB_8: TSpeedButton;
    SB_9: TSpeedButton;
    SB_0: TSpeedButton;
    ArgMemo: TMemo;
    SB_Point: TSpeedButton;
    SB_dev: TSpeedButton;
    SB_mul: TSpeedButton;
    SB_min: TSpeedButton;
    SB_plus: TSpeedButton;
    SB_del: TSpeedButton;
    SB_Eq: TSpeedButton;
    SB_Clear: TSpeedButton;
    Procedure SB_DigitClick(Sender: TObject);
    Procedure SB_PointClick(Sender: TObject);
    Procedure SB_OperClick(Sender: TObject);
    Procedure SB_EqClick(Sender: TObject);
    Procedure SB_delClick(Sender: TObject);
    Procedure SB_ClearClick(Sender: TObject);
    Procedure FormCreate(Sender: TObject);
    Procedure FormClose(Sender: TObject; var Action:
        TCloseAction);
    Private
        { Private declarations }
    Public
        { Public declarations }
End;
    // Описание класса TOperation
TOperation = Class
    Arg1:Real;
    Arg2:real;
    Oper:Char;
    LastOper:boolean;
    Procedure InitOperation;
    Procedure SetArg1(ArgString:string);
    Procedure SetArg2(ArgString:string);
    Procedure SetOper(OperString:char);
    Function OperationResult:string;
End;
```

```
Var
     Form1: TForm1;
     Ope_Result:TOperation;
implementation
     {$R *.dfm}
     // Методы класса TOperation
Procedure TOperation.InitOperation;
     Begin
          Arg1:=0;
          Arg2:=0;
          Oper:=#0;
          LastOper:=false
     End;
Procedure TOperation.SetArg1(ArgString:string);
     Var i:Integer;
     Begin
          val(ArgString,Arg1,i)
     End;
Procedure TOperation.SetArg2(ArgString:string);
     Var i:Integer;
     Begin
          val(ArgString,Arg2,i)
     End;
Procedure TOperation.SetOper(OperString:char);
     Begin
          Oper:=OperString;
     End;
Function TOperation.OperationResult:string;
     Var R:Real;
     Begin
          R:=0;
          case Oper of
          '+': R:= Arg1+Arg2;
          '-': R:= Arg1-Arg2;
          '*': R:= Arg1*Arg2;
          '/': R:= Arg1/Arg2;
     End;
     str(R:10:4,Result);

End;
     // Обработчики событий класса TForm1
Procedure TForm1.SB_DigitClick(Sender: TObject);
     Begin
     if Ope_Result.LastOper then ArgMemo.Text:='';
     ArgMemo.Text:=ArgMemo.Text+(Sender as
             TSpeedButton).Caption;
     Ope_Result.LastOper := false;
End;
```



```
Procedure TForm1.SB_PointClick(Sender: TObject);
   Begin
        If Ope_Result.LastOper Then
     Begin ArgMemo.Text:=''; Ope_Result.LastOper := false end;
     If ArgMemo.Text = '' Then
     ArgMemo.Text:=ArgMemo.Text+'0.'
     Else If pos('.',ArgMemo.text)= 0
     Then ArgMemo.Text:=ArgMemo.Text+'.';
   End;
Procedure TForm1.SB_OperClick(Sender: TObject);
     Var s:shortstring;
     Begin
          With Ope_Result do
          Begin
               SetArg2(ArgMemo.Text);
               If Oper <> #0 Then s:= OperationResult Else
               s:=ArgMemo.Text;
               SetArg1(s);
               ArgMemo.Text:=s;
               s:= (Sender as TSpeedButton).Caption;
               SetOper(s[1]);
               LastOper:=true;
          End;
     End;
Procedure TForm1.SB_EqClick(Sender: TObject);
     Begin
          With Ope_Result Do
          Begin
               If Oper <> #0 then
               Begin SetArg2(ArgMemo.Text);
                    ArgMemo.Text:=OperationResult;
                    SetArg1(ArgMemo.Text);
               End;
               InitOperation;
               LastOper:=true;
          End;
     End;
Procedure TForm1.SB_delClick(Sender: TObject);
     Var s:String;
     Begin
          s:= ArgMemo.text;
          SetLength(s,Length(s)-1);
          ArgMemo.text:=s;
     End;
Procedure TForm1.SB_ClearClick(Sender: TObject);
     Begin
          ArgMemo.text:='';
          Ope_Result.InitOperation;
     End;
```

```
Procedure TForm1.FormCreate(Sender: TObject);
     Begin
          Ope_Result:=TOperation.Create;
          Ope_Result.InitOperation;
     End;
Procedure TForm1.FormClose(Sender: TObject; var Action:
          TCloseAction);
     Begin
          Ope_Result.Free;
     End;
End.
```

Разработка приложения Калькулятор в среде Visual Basic

Рассмотрим далее реализацию задачи Калькулятор в среде Visual Basic.

Для решения задачи создадим класс MyClass, описывающий выполнение арифметической операции над двумя аргументами.

Чтобы определить в Visual Basic новый класс данных, имеющий свои поля данных и свои методы, необходимо через главное меню среды добавить модуль класса (рис. 4.32).

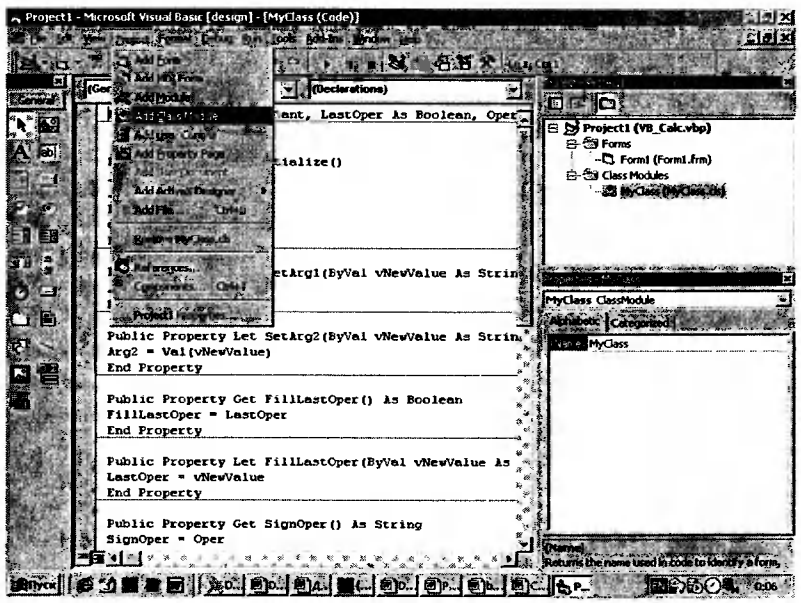


Рис. 4.32. Добавление модуля класса

Далее для класса необходимо определить данные, свойства и методы. Алгоритмы обработки аналогичны алгоритмам, примененным при решении подобной задачи в среде Delphi.

Текст модуля класса со всеми разработанными для него свойствами приведен ниже:

```
(General) (declarations)
Dim Arg1, Arg2 As Variant, LastOper As Boolean,
Oper As String
(Sub)
Private Sub Class_Initialize()
     Arg1 = 0
     Arg2 = 0
     LastOper = false
     Oper = ""
End Sub

Public Property Let SetArg1(ByVal vNewValue As String)
     Arg1 = Val(vNewValue)
End Property

Public Property Let SetArg2(ByVal vNewValue As String)
     Arg2 = Val(vNewValue)
End Property

Public Property Get FillLastOper() As Boolean
     FillLastOper = LastOper
End Property

Public Property Let FillLastOper(ByVal vNewValue As Boolean)
     LastOper = vNewValue
End Property

Public Property Get SignOper() As String
     SignOper = Oper
End Property

Public Property Let SignOper(ByVal vNewValue As String)
     Oper = vNewValue
End Property

Public Function OperationResult()
     R = 0
     ' Оператор Select Case в зависимости от знака операции
     ' обеспечивает выполнение арифметического действия.
     ' Локальная переменная R служит для временного
     ' хранения значения результата
     Select Case Oper
          Case "+"
          R = Arg1 + Arg2
```

```
        Case "-"
        R = Arg1 - Arg2
        Case "*"
        R = Arg1 * Arg2
        Case "/"
        R = Arg1 / Arg2
        Case Else
        R = 0
    End Select
    OperationResult = Str(R)
End Function
```

Предполагаемый внешний вид формы приложения представлен на рис. 4.33, *1*.

Исправим некоторые значения свойств в окне свойств формы `Form1` (рис. 4.33, *2*): изменим свойство `Caption` — текст заголовка окна формы, задав значение `Пример-калькулятор`;

Окно для ввода значений аргументов и вывода результата представляет собой визуальный компонент `TextBox`. Кнопки реализуются с помощью компонентов `CommandButton`. При разработке формы использовалась возможность создания массивов

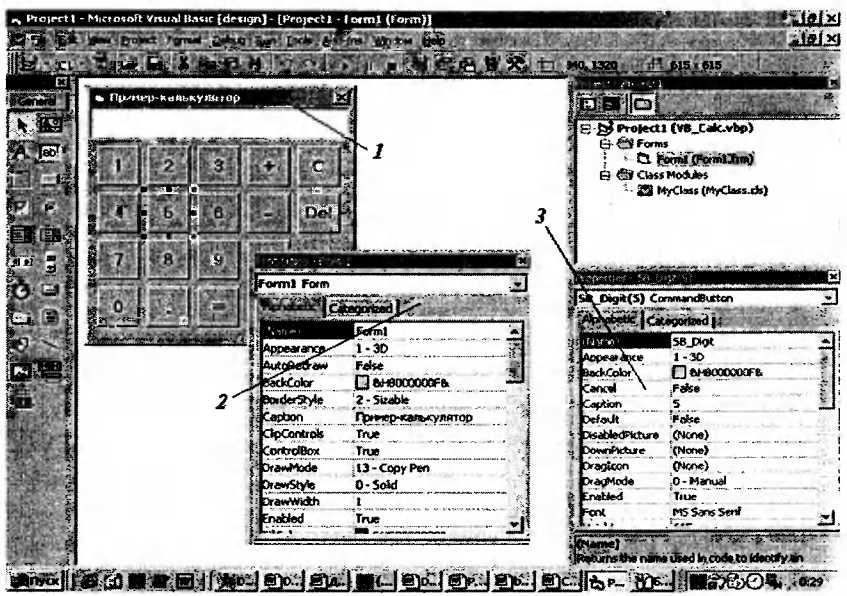


Рис. 4.33. Окно программы `Калькулятор` (*1*); изменение свойств формы в окне `Инспектора Объектов` (*2*); свойства кнопки-элемента массива (*3*)

однотипных кнопок с одной процедурой обработки события `Click`.

Приведем описание процедур обработки событий.

Событие `Click` для <Кнопки «Цифра»> вызывает выполнение процедуры `SB_Digit_Click`:

```
Private Sub SB_Digit_Click(Index As Integer)
    If Ope_Result.FillLastOper Then
        ArgMemo.Text = ""
        Ope_Result.FillLastOper = false
    End If          ' Добавление цифры к числу справа
        ArgMemo.Text = ArgMemo.Text +
        SB_Digit.Item(Index).Caption
End Sub
```

Процедура обработки события для <Кнопки «Точка»> (`SB_Point_Click`):

```
Private Sub SB_Point_Click()
    If Ope_Result.FillLastOper Then
        ArgMemo.Text = ""
        Ope_Result.FillLastOper = false
    End If
    If ArgMemo.Text = "" Then
        ArgMemo.Text = ArgMemo.Text + "0."
        ElseIf InStr(ArgMemo.Text, ".") = 0
        Then ArgMemo.Text = ArgMemo.Text + "."
    End If
End Sub
```

Обработка события `Click` <Кнопок «Знак операции»> `SB_Oper_Click`:

```
Private Sub SB_Oper_Click(Index As Integer)
    With Ope_Result
        SetArg2 = ArgMemo.Text
        If .SignOper <> "" Then
            s = .OperationResult
            Else: s = ArgMemo.Text
        End If
        .SetArg1 = s 'Занесение значения первого аргумента
        .SetArg2 = "0" 'Обнуление второго аргумента
        ArgMemo.Text = s 'Вывод значения первого аргумента
        .SignOper = SB_Oper.Item(Index).Caption
        ' Установка знака операции и флага ввода операции
        FillLastOper = true
    End With
End Sub
```

Процедура обработки события Click для <Кнопки «Равно»>

```
Private Sub SB_Eq_Click()
     With Ope_Result
          If .SignOper <> "" Then
               .SetArg2 = ArgMemo.Text
               ArgMemo.Text = .OperationResult
               .SetArg1 = ArgMemo.Text
          End If
     .SetArg1 = 0
     .SetArg2 = 0
     .SignOper = ""
     .FillLastOper = true
     End With
End Sub
```

Для поддержки событий нажатия на кнопки (<'C'>) и <'Del'> опишем процедуры SB_Clear_Click и SB_del_Click:

```
Private Sub SB_Clear_Click()
     ArgMemo.Text = ""
     ' Очистка текстового значения окна ArgMemo
     With Ope_Result
          .SetArg1 = 0
          .SetArg2 = 0
          .SignOper = ""
          .FillLastOper = false
     End With
End Sub

Private Sub SB_Del_Click()
     s = ArgMemo.Text
     'Установка нового значения длины строки (меньше на 1)
     ArgMemo.Text = Left(s, Len(s) - 1)
End Sub
```

Алгоритмы обработки событий строятся на использовании объекта Ope_Result, поэтому необходимо обеспечить размещение объекта в памяти и освобождение памяти при завершении работы приложения.

```
Dim Ope_Result As MyClass

Private Sub Form_Load()
     Set Ope_Result = New MyClass
End Sub

Private Sub Form_Unload(Cancel As Integer)
     Set Ope_Result = Nothing
End Sub
```


Средой Visual Basic сгенерирован текст основной программы и построен модуль MyClass, содержащий описание класса Toperation.

Основной модуль:

```
Dim Ope_Result As MyClass

Private Sub Form_Load()
     Set Ope_Result = New MyClass
End Sub

Private Sub Form_Unload(Cancel As Integer)
     Set Ope_Result = Nothing
End Sub

Private Sub SB_Digit_Click(Index As Integer)
     If Ope_Result.FillLastOper Then
          ArgMemo.Text = ""
          Ope_Result.FillLastOper = false
     End If
     ArgMemo.Text = ArgMemo.Text +
     SB_Digit.Item(Index).Caption
End Sub

Private Sub SB_Oper_Click(Index As Integer)
     With Ope_Result
          .SetArg2 = ArgMemo.Text
          If .SignOper <> "" Then
               s = .OperationResult
               Else: s = ArgMemo.Text
          End If
          .SetArg1 = s
          .SetArg2 = "0"
          ArgMemo.Text = s
          .SignOper = SB_Oper.Item(Index).Caption
          .FillLastOper = true
     End With
End Sub

Private Sub SB_Point_Click()
     If Ope_Result.FillLastOper Then
          ArgMemo.Text = ""
          Ope_Result.FillLastOper = false
     End If
     If ArgMemo.Text = "" Then
          ArgMemo.Text = ArgMemo.Text + "0."
          ElseIf InStr(ArgMemo.Text, ".") = 0
          Then ArgMemo.Text = ArgMemo.Text + "."
     End If
End Sub
```

```
Private Sub SB_Eq_Click()
     With Ope_Result
          If .SignOper <> "" Then
               .SetArg2 = ArgMemo.Text
               ArgMemo.Text = .OperationResult
               .SetArg1 = ArgMemo.Text
     End If
     .SetArg1 = 0
     .SetArg2 = 0
     .SignOper = ""
     .FillLastOper = true
     End With
End Sub

Private Sub SB_Clear_Click()
     ArgMemo.Text = ""
     With Ope_Result
          .SetArg1 = 0
          .SetArg2 = 0
          .SignOper = ""
          .FillLastOper = false
     End With
End Sub

Private Sub SB_Del_Click()
     s = ArgMemo.Text
     ArgMemo.Text = Left(s, Len(s) - 1)
End Sub
```

Контрольные упражнения

1. Модифицировать приложение Калькулятор (как для Basic, так и для Delphi), добавив операцию (кнопку) возведения в целую степень.
2. Модифицировать приложение Калькулятор, добавив операцию вычисления *N*!.
3. Модифицировать приложение Калькулятор, добавив операцию «возведение в квадрат».
4. Модифицировать приложение Калькулятор, добавив операцию «извлечение квадратного корня».

Глава 5
FOXPRO — СИСТЕМА ПРОГРАММИРОВАНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

Система программирования с развитыми элементами системы управления базами данных (СУБД) FoxPro [6, 18] предназначена для разработки приложений, являющихся открытыми или замкнутыми документальными или фактографическими информационными системами, и впервые предложена в 1988 г. Предшествующими аналогичными системами являлись — dBase/I/II/III/IV и Clipper (1984—1987). Аналогичность здесь подразумевает преемственность, основанную на том, что различие команд, форматов и функций не превосходит 10—15 % и ранее разработанные приложения не требуют существенных переработок; навыки программирования, полученные на Clipper, имеют силу для FoxPro и т. п.

Здесь рассматриваются возможности и свойства подобных систем на примере Visual FoxPro 9.0 (VFP9.0) — системы, использующей принципы объектно-ориентированного программирования и визуального проектирования приложений. Авторы заранее признаются читателям в том, что здесь будет использовано и проиллюстрировано не более чем 15—20 % возможностей этого мощного программного продукта. Далее, некоторые из приводимых ниже в примерах команд, функций и типов файлов не являются присущими именно VFP, а поддерживаются системой для обеспечения преемственности и совместимости с более ранними версиями — FoxPro for Windows и FoxPro/DOS.

Приведем текст простой диалоговой программы, содержащий наиболее характерные особенности ЯП FoxPro (команды ввода-вывода на экран и управления программой).

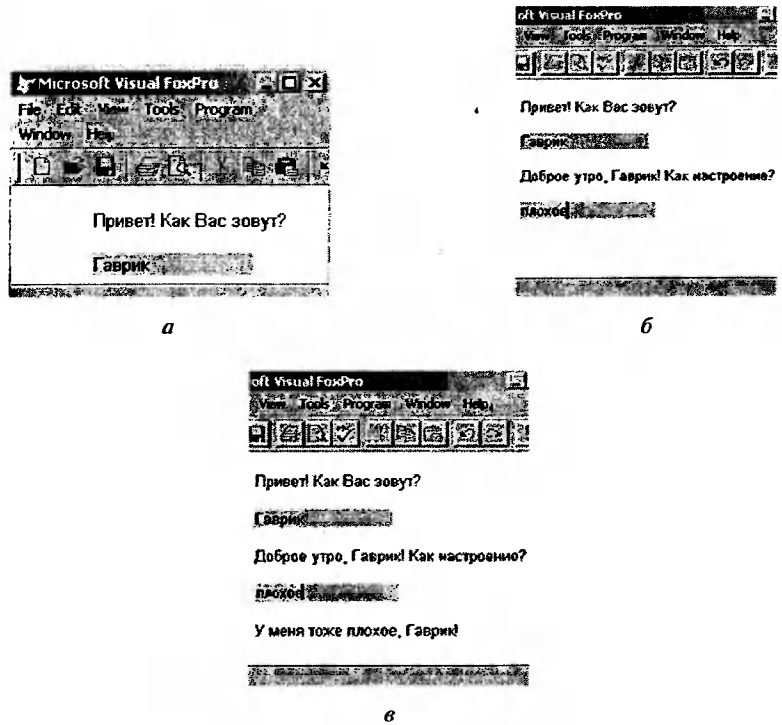
Возможный диалог с такой программой может выглядеть следующим образом (см. также рис. 5.1):

```
Привет! Как Вас зовут?
Гаврик
Доброе утро, Гаврик! Как настроение?
Плохое
У меня тоже плохое, Гаврик!
```

Рассмотрим текст такой программы.

```
* Простая диалоговая программа
Set Echo off
Set Talk off
On Escape quit      &&условие выхода из программы
Clear
Do While .t.                  && внешний цикл
     name= [ ]                && инициализация переменной
     @ 1,10 Say [Привет! Как Вас зовут?]
     @ 3, 10 Get name
     Read                                          (рис. 5.1, а)
     If name =[ ]             &&требование ввода непустой строки
          Loop
     Endif
     If substr(name,1,1)=[*]     && выход из программы
          Exit
     Endif
     Do while .t.                && внутренний цикл
          nastr=[ ]              && инициализация переменной
          @ 5,10 Say [Доброе утро, ]+rtrim(name)+
          [! Как настроение?]
          @ 7,10 Get nastr
          Read                                     (рис. 5.1, б)
          If substr(nastr,1,1)=[ ]
               Loop
          Endif
          If substr(nastr,1,1)=[*]
                              &&условие выхода из внутр. цикла
               Exit
          Endif
          @ 9,10 Say [У меня тоже ]+rtrim(nastr)+
          [, ]+rtrim(name)+[!]                     (рис. 5.1, в)
     Enddo
Enddo
* Конец программы
```

В начале программы находятся команды из множества **Set**, настраивающие режим работы интерпретатора или компилятора на обработку исходного текста в определенном режиме. В дан



а *б*

в

Рис. 5.1. Иллюстрации к простой диалоговой программе:
а — начало внешнего цикла; *б* — начало внутреннего цикла; *в* — окончание внутреннего цикла и выход во внешний

ном случае речь идет о запрещении дублирования вводимых команд на экране. В этом случае на экране отображается только та информация, которая находится в операторах **Say**, **Get**.

On Escape — задание действий при нажатии клавиши <Esc> (в данном случае — прекращение работы программы и выход из среды FoxPro). Инициализация переменных `name`, `nastr` одновременно задает их тип как строковый и определяет длину.

Программа состоит из двух вложенных циклов **Do ... Enddo**, в заголовках которых стоит условие бесконечного повторения (`.t.`), поэтому должен быть предусмотрен принудительный выход из каждого цикла. Этот выход происходит при условии ввода в строке `name` или `nastr` символа «*» («звездочка»). С помощью команды **Exit** осуществляется выход во

внешний цикл из внутреннего и из внешнего циклов на окончание работы.

При попытке ввести пустую строку в ответ на запрос программа возвращается в заголовок, соответственно, внутреннего или внешнего цикла и повторяет запрос на ввод строки (команда возврата в начало цикла **Loop**). Управление условными переходами осуществляется комнадой **If ... Endif**.

Команды **Say** и **Get** осуществляют вывод на экран строки текста и открытие окна для ввода в соответствующих позициях экрана (строка и столбец). Для ограничения строки (литерала) в программе может использоваться как двойная кавычка ("), так и квадратные скобки ([]).

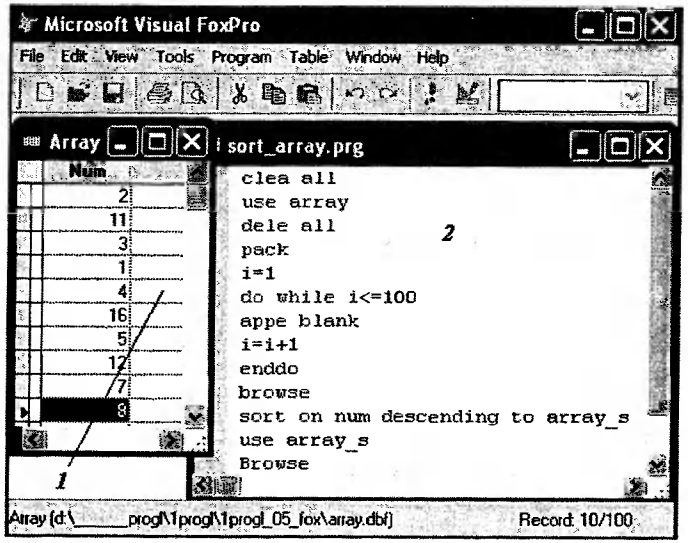
Команда **Read** останавливает программу для ожидания ввода данных в окно на экране и нажатия подтверждения <Enter>. Команда **Clear** осуществляет очистку экрана монитора.

Конкатенация (сцепление) строк осуществляется оператором «+», а функция `rtrim()` используется для подавления «хвостовых» пробелов во введенной строке с целью повышения удобочитаемости текста. Функция `substr()`, обычная и для других ЯП, позволяет выделить из строки-операнда подстроку-результат (с указанием длины подстроки и смещения от начала исходной строки).

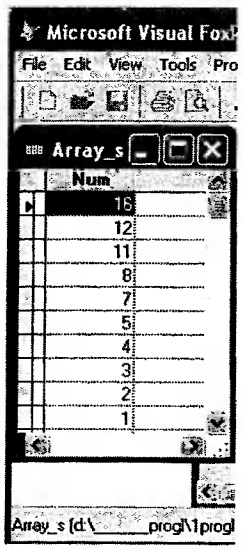
Из текста программы следует также, что допустимы два типа комментариев — полная строка (начинается с символа *) и частичная строка (начинается с &&).

Программа сортировки по убыванию массива чисел занимает несколько строк, 70 % которых — инициализация (очистка) файла данных:

```
Clea All              && Закрытие всех файлов данных
Use array             && Открытие файла данных Array
Dele All              && Логическое удаление всех записей
Pack                  && Физическое удаление
i=1
Do While i<=100
     Appe Blank
     i=i+1
Enddo           && Цикл создания 100 «пустых» записей файла
Browse          && Ввод числовых значений          (рис. 5.2, а)
Sort On num Descending To array_s
     * Сортировка по убыванию с записью в массив Array_s
Use array_s     & Переход к Array_s
Browse          && Вывод числовых значений         (рис. 5.2, б)
```

а



б

Рис. 5.2. Иллюстрации к программе сортировки массива (файла): *а* — ввод данных (в файл `Array.dbf`); *б* — вывод результата (из файла `Array_s.dbf`); *1* — окно команды **Browse**; *2* — окно редактора программ

В начале программы осуществляется закрытие всех активных файлов (таблиц БД) и открытие ранее созданного файла данных `Array`, записи которого содержат единственное поле `Num` (рис. 5.2). Затем осуществляется удаление всех записей файла (сохраненных результатов ранее выполненных прогонов программы). Цикл **`Do While...Enddo`** создает 100 незаполненных записей файла данных, после этого командой **`Edit`** можно заполнить и просмотреть файл данных (рис. 5.2, *а*). Затем командой **`Sort On ... Descending To`** осуществляется сортировка записей по убыванию значения поля `Num` и результат записывается в файл `Array_s`, после этого записи файла просматриваются (рис. 5.2, *б*) второй командой **`Edit`**.

Из текста программ также очевидно, что для ввода большинства команд достаточно первых четырех символов наименований — **`Clea`** вместо **`Clear`**, **`Dele`** вместо **`Delete`** и пр., а также, что заглавные и строчные символы не различаются интерпретатором.

5.1. Типы и структуры данных, файлы, базы данных

Начнем рассмотрение, как и ранее, с данных. В ЯП FoxPro данным соответствуют:

- типы;
- контейнеры;
- операторы.

Типы, контейнеры данных, операторы над данными

Типы данных. В FoxPro, как и в других ЯП, фигурируют различные типы данных; в табл. 5.1 приведены примеры основных из них. Более подробное описание типов (с учетом контейнеров) содержится в табл. 5.4.

Таблица 5.1. **Примеры основных типов данных**

Тип	Примеры
Числовой	`123 3.1415 - 7`
Символьный	`"Test String" "123" [01/01/98]`
Логический	`.t. .f.`
Дата	`{^1998-01-01}`
Дата-время	`{^1998-01-01 12:30:00 p}`

Контейнеры данных — формы, в которых те или иные типы данных появляются для обработки в программах, участвуя в выражениях и связываясь между собой операторами (табл. 5.2).

Таблица 5.2. **Основные контейнеры данных ЯП FoxPro**

Тип	Описание
Переменные	Элементарные данные, хранящиеся в оперативной памяти компьютера
Записи файла данных	Множественные записи файла (или строки таблицы), содержащие разнотипные данные. Таблицы (файлы) хранятся на НЖМД
Массивы	Множественные элементы данных, хранящиеся в оперативной памяти

Операторы. Наконец, примеры операторов, связывающие данные в выражениях, приводятся в табл. 5.3.

Таблица 5.3. **Примеры некоторых операторов**

Оператор	Действие	Тип данных	Пример	Результат
=	Сравнение, присваивание	Любой	`? n = 7`	Печать `.t.` если переменная памяти `n` равна 7, `.f.` в противном случае
+	Сложение, конкатенация	Числовой, символьный, дата, дата-время	`? "Fox" + "Pro"` `? 5+7`	Печать «FoxPro» Печать 12
! или `.not.`	Отрицание	Логический	`? !.t.`	Печать `.f.`
*	Умножение	Числовой	`? 5 * 5`	Печать 25
/	Деление	Числовой	`? 25 / 5`	Печать 5

Примечание. Знак вопроса (?) — команда вывода на экран или печать строки данных.

Заметим, что если при описании ЯП Basic и Pascal мы заканчивали материал главы файлами и операциями над ними, здесь мы с них начнем, поскольку, как уже отмечалось, это — система программирования с элементами СУБД, в которой, в силу особенностей исторического развития, файлы (объекты операционной системы и файловой системы — физический уровень рассмотрения) по своему содержанию представляют собой таблицы (или отношения) — компоненты реляционной или табличной базы данных (объекты реляционной СУБД — ло-

гический уровень рассмотрения), и поэтому их именуют файлами данных.

Данные (и их типы), которые ранее определялись в описаниях ЯП, здесь становятся переменными памяти (memory variables), которые, в общем-то, играют второстепенную, вспомогательную роль. На первый план здесь выходят поля базы данных (точнее, файлов данных, или таблиц — field variables).

Необходимо отметить также и другие отличия от привычной терминологии, используемой в FoxPro (в системах помощи и документации FoxPro), которой мы здесь вынуждены придерживаться. Операторы и ключевые слова, известные из состава других ЯП (например, `If...Endif`) здесь обычно именуются командами, операторами же являются элементы, известные в других ЯП как операции («<», «=», «>» и пр., см. табл. 5.6).

Кроме команд (некоторые из которых приведены в табл. 5.8) и функций (см. табл. 5.7), VFP поддерживает объекты (object), методы (method), свойства (property) и события (event), описания которых, как и системных переменных (system variable), мы вынуждены опустить.

Типы данных

Типы данных FoxPro приведены в табл. 5.4, состоящей из двух частей — в первой описаны типы, характерные как для данных в памяти, так и для полей БД, во второй — только для таблиц БД.

Таблица 5.4. **Типы данных в ЯП/СП Visual FoxPro**

Тип	Описание	Размер	Пределы изменения
Данные переменных памяти (data types)			
Большой бинарный объект (Blob)	Бинарные данные неопределенной длины. Могут размещаться в мемо-полях (файлах .fpt). Перекодирование символов недоступно	4 байта в таблице	Ограничивается доступной памятью и/или пределом в 2 Гбайта
Символьный (Character)	Алфавитно-цифровой текст	От 1 до 254 байт	Любые символы

Продолжение табл. 5.4

Тип	Описание	Размер	Пределы изменения
Валюта (Currency)	Денежные единицы	8 байт	От –$922337203685477,5807 до $922337203685477,5807
Дата (Date)	Хронологическая дата, состоящая из года, месяца и дня	8 байт	При использовании формата ГГГГ:ММ:ДД (год, месяц, день) — от 0001-01-01 (1 января 1 г. н. э.) до 9999-12-31 (31 декабря 9999 г. н. э.)
Дата-время (DateTime)	Включает, кроме даты, также часы, минуты и секунды	8 байт	От 0001-01-01 (плюс 00:00:00 a.m.) до 9999-12-31 (плюс 11:59:59 p.m.)
Логический (Logical)	Булевское значение «истина" или «ложь"	1 байт	True (.t.) или False (.f.)
Числовой (Numeric)	Целое или десятичное число	8 байт в памяти, от 1 до 20 байт в таблице	От –0,9999999999E+19 до 0,9999999999E+20
Бинарный переменной длины (Varbinary)	Битовая строка. Аналогичен Varchar, но не включает заполнение нулевыми (0) байтами. Длина строки хранится в ней. Перекодирование символов недоступно	1 байт на шестнадцатеричное число до 255 байт	Любые шестнадцатеричные величины
Неопределенный (Variant)	Переменная типа Variant может содержать любые допустимые типы данных VFP. Обозначаются префиксом «e» в программе	См. другие типы данных	См. другие типы данных
Поля файлов данных (таблиц БД) — field types			
Символьно-двоичный Character (Binary)	Любые символьные данные, которые не должны перекодироваться	1 байт на символ до 254	Любые символы

Окончание табл. 5.4

Тип	Описание	Размер	Пределы изменения
Двойной точности (Double)	Число с плавающей запятой двойной точности	8 байт	От ±4,94065645841247E-324 до ±8,9884656743115E307
Число с плавающей запятой (Float)	То же, что и числовой (Numeric)	8 байт в памяти; 1—20 байт в таблице	От –0,9999999999E+19 до 0,9999999999E+20
Общий (General)	Ссылка на OLE-объект, например, книгу Microsoft Excel	4 байта в таблице	Ограничивается доступной памятью
Целый (Integer)	То же, что числовой (Numeric), но без дробной части	4 байта	От –2147483647 до 2147483647
Примечание (мемо-поле) (Memo)	Алфавитно-цифровой текст неопределенной длины или ссылка на блок данных	4 байта в таблице	Ограничено доступной памятью
Бинарное мемо-поле Memo (Binary)	То же, что Memo, однако не подлежит перекодировке*	4 байта в таблице	Ограничено доступной памятью
Символьный переменной длины (Varchar)	Алфавитно-цифровой текст. То же, что и Character, но не содержит «хвостовые» пробелы. Длина содержится в строке	1 байт на символ и до 254 байт	Любые символы
Бинарный Varchar (Binary)	То же, что и Varchar, но не подлежит перекодировке*	1 байт на символ и до 254 байт	Любые символы

* Строка может содержать «непечатные» битовые комбинации, не описанные ни в одной из кодовых таблиц

Табличные базы данных

Рассмотрим пример БД, реализованной в рамках системы программирования FoxPro и состоящей из трех файлов данных (таблиц), описывающих некоторых граждан, их автомобили и связанные с ними финансовые учреждения (рис. 5.3).

а

#	prsn#	fio	year	sex	adress	profession
1	**576**	**Распутин**	**1941**	**м**	**Москва**	**Программист**
2	231	Петрова	1937	ж	Гомель	Водитель
3	256	Иванов	1945	м	Самара	Преподаватель
4	578	Сидорова	1987	ж	Тамбов	Продавец
5	132	Грачев	1978	м	Москва	Секретарь

б

#	drv#	make	body	year	color	horses
1	256	OPEL	СЕДАН	1991	Синий	78
2	578	ЖИГУЛИ	УНИВЕР	1994	Черный	56
3	**576**	**BMW**	**СЕДАН**	**1987**	**Белый**	**125**

в

#	ovn#	bank	account#	current
1	**576**	**Автобанк**	**2345./34**	**23,345.00**
2	132	Сбербанк	25058-б	1,000.00
3	578	Интербанк	5476-34	765,243.00

г

Рис. 5.3. Типичная структура простейшей табличной БД:
а — общая логическая структура БД; *б* — структура файла (таблицы) person.dbf; *в* — структура auto.dbf; *г* — структура finances.dbf

Подобные БД называются табличными, или реляционными (от relation — отношение), и их теория рассматрива-

ется во многих источниках (см., например, [7]). Здесь мы ограничимся лишь базовыми понятиями:

- файл (file) соответствует совокупности однородных объектов и содержит их более или менее подробные описания в зависимости от приложений. В реляционных БД это — таблица или отношение. Файл имеет имя (например, `finance.dbf` и пр.);
  - элементом файла является запись (record) или агрегат разнотипных данных, описывающих объект (точнее, экземпляр объекта). В реляционных БД это — строка таблицы или экземпляр отношения. Записи имен не имеют, но им соответствуют физические номера в файле (колонка # на рис. 5.3, *а*);
  - элементом записи (здесь — неделимым) является поле — данное, описывающее какой-либо аспект (или атрибут, реквизит) объекта. Поля имеют имена (`prsn#`, `sex` и пр.). В реляционных БД это — столбец таблицы или атрибут отношения (рис. 5.3, *а* и пр.). Разные файлы могут иметь поля с одинаковыми именами, но лучше этого избегать. Иногда вводится понятие домена, или совокупности допустимых значений атрибута (например, поле `sex` может иметь только два значения — «м», «ж», поле `year` — только четырехразрядные числа, начинающиеся с 19, если в БД речь идет о родившихся в XX в. и пр.);
- открытый (opened) файл — файл, доступный в данный момент данному приложению. Открытие файла создает в памяти буфер, в который с внешнего накопителя считываются записи. В разные моменты времени могут быть открыты различные множества файлов, количество открытых файлов обычно стараются ограничить, чтобы не расходовать оперативную память;
- активный или текущий (current, active) — тот из открытых файлов, который обрабатывается в данный момент времени. Все операции над файлами (добавление записи; удаление записи; редактирование записи) адресуются именно к активному файлу;
- активная или текущая запись — запись открытого файла (рис. 5.4, *а*), доступная для обработки в данный момент времени (редактирование, ввод полей, корректировка, удаление). Указатель текущей записи есть физический номер доступной записи. Текущая запись находится в оперативной памяти. При переходе к другой записи данного файла указатель записи изменяется, и содержание оперативной памяти замещается содержимым новой текущей записи. Подразумевается, что если в командах или программах (аргументы функций и выражений) фигурируют имена некоторых полей, то их значения соответствуют содержанию текущей записи текущего файла;
- каждый файл и каждая запись могут в широких пределах обрабатываться независимо друг от друга (за исключением ситуаций проверки соответствия записей друг другу или *целостности БД*);
- навигация в БД — последовательность действий приложения (программы или пользователя в процессе диалога), при которой осуществляются изменения состояния файлов и записей (открытых, текущих файлов, активных записей). Изменение содержимого файлов при навигации необязательно. В процессе навигации просматривается или редактируется содержимое БД.

Вид представления записей на экране может быть не только табличным (отчет, запись в строке), но и картотечным (форматированный экран, запись на экране).

В последнем случае каждая запись выводится в виде определенной экранной формы. Структура формы одинакова для всех записей, причем название полей соответствует названиям столбцов табличной формы представления базы данных, а их расположение задается пользователем (см. рис. 5.22).

Так как строки в таблице не упорядочены, невозможно выбрать строку по ее позиции — среди них не существует «первой», «второй», «последней». Любая таблица имеет один или несколько столбцов, значения в которых однозначно идентифицируют каждую ее строку. Такой столбец (или комбинация столбцов) называется первичным ключом (primary key). В таблице `person` первичный ключ — это столбец `prsn#`, в таблице `auto` — столбец `drv#`. В этом столбце значения не могут дублироваться — в таблице `person` не должно быть строк, имеющих одно и то же значение в столбце `prsn#`.

Взаимосвязь таблиц является важнейшим элементом реляционной модели данных. Она поддерживается внешними ключами (external). Из рис. 5.3 видно, что **Распутин** (таблица `person`)

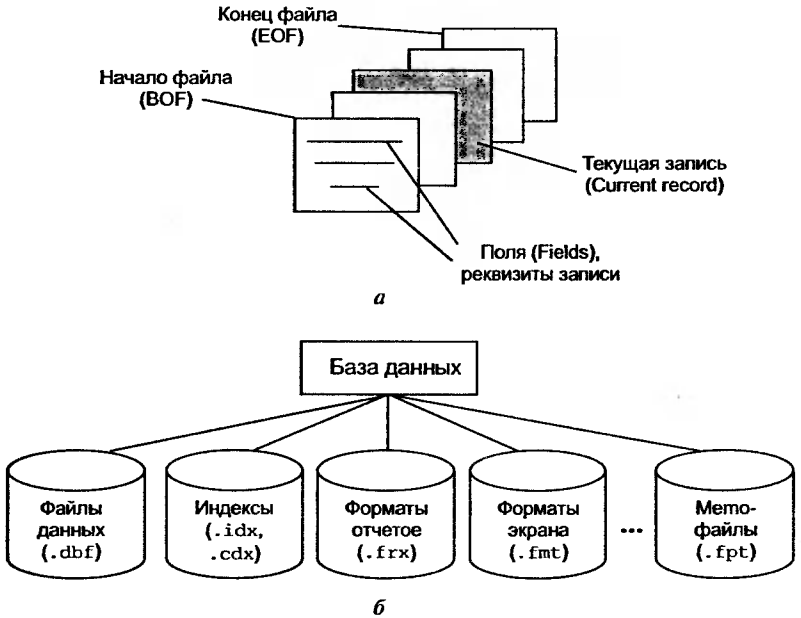


Рис. 5.4. Табличная БД в СП FoxPro:
а — навигация в таблице; *б* — физическая структура БД

разъезжает на автомобиле **BMW** (таблица `auto`) и держит деньги в **Автобанке** (таблица `finances`). Это следует из совпадения значений уникальных ключей (личный номер, номер водителя, номер владельца счета — `prsn# = drv# = ovn# = 576`).

Физическая структура БД

Всякая БД в FoxPro должна состоять хотя бы из одного файла данных, однако, кроме них, в ней может содержаться множество служебных файлов различных типов.

Основные типы файлов и расширения. В БД FoxPro предусмотрены следующие типы файлов:

- файл данных, файл базы данных (`database`, `dbf`) содержит упорядоченный набор определенных данных;
- индексный файл (`index`) управляет порядком доступа к записям в конкретной базе данных и их обработки. Индексные файлы позволяют изменить порядок, в котором записи

файла данных будут появляться на экране, порядок, в каком они будут напечатаны и т. д., однако фактический порядок данных в базе данных при этом не изменится (см. рис. 5.4). Расширения — `.idx` (компактный индекс), `cdx` (составной индекс). Одному файлу БД может соответствовать несколько существующих и/или активных индексных файлов. Текущий/активный индекс — открытый индексный файл, выбранный для управления текущим файлом БД; выборка данных из БД осуществляется по возрастанию ключа (индексного выражения, вычисляемого по полям текущего файла), соответствующего текущему индексу;

- файл связанных данных, мемо-поля (`memo`, `fpt`) содержит данные, сохраняемые в мемо-полях, являющихся частью базы данных. Информация из мемо-полей не сохраняется в файле данных (расширение `.dbf`), вместо этого она сохраняется в файле с тем же именем и расширением `.fpt`;
- формат экрана (`screen`, `frt`) содержит описание экрана пользователя (описание порядка выдачи данных на экран монитора или чтения данных с экрана), которое определяет форматы, используемые для ввода, редактирования и просмотра данных;
- формат отчета (`report`, `frx`) содержит описание отчета. Это описание определяет какую информацию содержит отчет, длину строки, ширину страницы, имена выдаваемых полей, заглавие отчета, имена колонок, печать итогов при прерывании и т. д. Это описание отчета используется для вывода отчета на экран или другое заданное устройство вывода информации;
- описание метки (`label`, `lbx`) содержит описание метки. Это описание определяет данные и расположение для вывода метки, включая ширину поля, ширину и высоту метки, сквозную нумерацию меток, а также расстояние и число строк между метками;
- командный (пакетный, программный, `program`, `prg`) содержит в каждой строке одну из команд языка FoxPro. Эти файлы могут быть созданы и отредактированы с использованием текстового редактора FoxPro (**`Modify File`**, **`Modify Command`**) или любого другого;
- откомпилированная программа (`compiled program`, `fxp`) содержит файл программы, которая откомпилирована

FoxPro в сжатую форму для более быстрого выполнения. FoxPro создает эти файлы для уменьшения времени решения этой задачи;

- текстовый файл (text, txt) содержит текстовые данные в коде ASCII, созданные с помощью команды **Copy To ..... Delimited**. Расширение .txt добавляется автоматически, когда создается файл текстовых данных. Это расширение также могло быть использовано с текстовыми файлами, созданными с помощью текстового редактора FoxPro, но расширение .txt должно быть определено, когда файл создан с помощью другого редактора. Например, следующая команда создает файл без расширения

```
Modify File notes
```

Если необходимо создать файл с расширением .txt, следует выдать команду, аналогичную следующей:

```
Modify File notes.txt
```

- резервная копия файла (file backup, bak) содержит предыдущую версию текста, программы или файла данных;
- файл сохранения переменных оперативной памяти (memory variable save, mem) позволяет запомнить информацию о переменных, определенных во время сеанса FoxPro (команда **Save To** ...);
- файл описания окна (window file, win) содержит описание окна, которое было создано с помощью команды

```
Save Window
```

Когда файл создан в FoxPro, расширение назначается автоматически (исключение для текстового файла). Если открывается файл, не надо указывать его расширение. Например, с помощью следующей команды открывается файл данных с именем datafile.dbf:

```
Use datafile
```

Может быть указано другое расширение файла, когда создается или открывается файл. Например, с помощью следующей команды создается файл данных с расширением new:

```
Create datafile.new
```


При открытии файла, имеющего расширение, отличное от расширения по умолчанию, пользователь должен указать его расширение.

Рабочие области и их псевдонимы. FoxPro предоставляет возможность одновременно открыть и работать с 10 файлами данных в 10 рабочих областях, которые идентифицируются буквами A–J или номерами 1—10. Перед открытием файла данных необходимо выбрать рабочую область. Когда файл открыт, ему присваивается псевдоним, с которым он позднее может быть идентифицирован. Псевдоним является именем файла данных (исключение для расширения .dbf).

Псевдоним по умолчанию. Если вы открыли базу данных, названную customer.dbf, в рабочей области A с помощью команд:

```
Select a
Use customer,
```

то файлу данных будет автоматически присвоен псевдоним customer.

В дальнейшем псевдоним может быть использован для определения того, в какой рабочей области происходит работа:

```
Select customer
Delete All For amt_due = 0
Pack
```

Назначение псевдонима. Существует возможность назначить псевдоним для рабочей области при открытии файла данных. Если открыть файл данных с именем customer.dbf (в рабочей области A) и назначить ему псевдоним people с помощью команды:

```
Select a
Use customer
Alias people,
```

то псевдоним people затем может быть использован для ссылки к рабочей области, в которой находится файл данных customer.

Обращение к рабочей области. Вы можете обратиться к рабочей области перед открытием файла данных, используя букву или номер рабочей области:

```
Select a  или
Select 1
```

Две вышеприведенные команды эквивалентны.

Если файл базы уже открыт, к рабочей области можно обратиться, используя букву, номер или псевдоним файла данных.

Например, если файл данных `customer.dbf` открыт в рабочей области `C (3)` и ему назначен псевдоним по умолчанию `customer`, можно обратиться к этой рабочей области из другой рабочей области с помощью одной из следующих команд:

```
Select c
Select 3
Select customer
```

При обращении к полю из другой рабочей области имя поля должно начинаться с псевдонима или буквы рабочей области файла данных, в которой находится необходимое поле, и разделителей «->» или «.» (точки).

Если текущей является рабочая область `C` и необходимо иметь доступ к полю с именем `lastname` из файла базы `customer`, открытого в рабочей области `A`, можно использовать следующие виды записи:

```
customer->lastname
a->lastname
customer.lastname
a.lastname
```

Если же файл данных `customer` в рабочей области `A` имеет псевдоним `people`, то можно воспользоваться обращениями:

```
people->lastname
a->lastname
people.lastname
a.lastname
```

Псевдоним переменной памяти. Некорректно назначать имена переменных идентичными именам полей файла данных (хотя иногда это полезно). Если имя переменной идентично имени поля файла данных, в FoxPro приоритет всегда имеет имя поля файла данных относительно переменной (имя понимается как ссылка на поле файла данных). Для устранения недоразумений доступ к переменной, имеющей такое же имя, как и поле, осуществляется с помощью префикса `m->` или `m.` (`m` с точкой):

```
m->lastname
m.lastname
```


Каждый раз, создавая поле файла данных (**Create**, **Create Structure**, **Create Database**), необходимо определить для него один из перечисленных выше типов. Назначенный тип можно изменить, модифицируя структуру файла данных (**Modify Database**, **Modify Structure**).

Поддержка пути. FoxPro позволяет определить множество каталогов (отличных от текущих рабочих каталогов), в которых осуществляется поиск файлов. Это делается с помощью команд **Set Default** и **Set Path**:

- **Set Default** изменяет имя накопителя по умолчанию на имя, отличное от имени накопителя по умолчанию операционной системы. Это важно помнить, так как хотя все операции FoxPro выполняются на накопителе, определенном командой **Set Default**, накопитель по умолчанию операционной системы остается тем же самым;
- **Set Path** определяет множество каталогов, используемых для поиска файла, если файл не найден в текущем каталоге. Местоположение каталога может быть указано или относительно, или с помощью полностью определенных имен путей, как показано ниже.

Путь считается полностью определенным, если он начинается с точки или обратной косой черты, а именно:

```
Set Path To \system\data
Set Path To ..\,
```

или если он начинается с имени накопителя, например:

```
Set Path To c:\system\data
```

Путь считается относительным для рабочего каталога, если он начинается с имени каталоги, например:

```
Set Path To data
```

Он обрабатывается FoxPro подобно пути, указанному полностью `set path to \data`.

Когда FoxPro пытается определить местоположение файла, имя которого определено не полностью, система сначала ищет его в рабочем каталоге на накопителе по умолчанию, определенном с помощью команды **Set Default**. Если поиск неудачен, то затем имена путей используются в порядке их появления в ко-

манде **Set Path**. Имена путей просто присоединяются к началу имени файла для осуществления поиска. Если теперь полностью определенное имя файла не содержит имени накопителя, то предполагается имя накопителя по умолчанию.

Правила поддержки пути, описанные выше, применяются для поиска существующих файлов с одним исключением. Команда **Dir** ищет в рабочем каталоге на накопителе по умолчанию, если имя пути полностью не определено.

Когда FoxPro создает файл, она также размещает его в рабочем каталоге на накопителе по умолчанию, если имя файла не содержит имени пути.

Массивы

FoxPro поддерживает переменные в памяти, представляющие собой одно- и двумерные массивы. К каждому элементу массива можно обращаться по индексам его строки и столбца. Так как массивы хранятся в памяти, к ним можно обращаться и работать с ними с высокой скоростью. Элементы массива могут содержать любой тип данных (символьный, числовой, тип даты, логический тип и пр.). При создании массива его элементы инициализируются логическим значением `false` (`.f.`).

Создание массивов. Для создания массивов используются команды **Dimension** и **Declare**. Данные команды идентичны по синтаксису и выполняемой ими функции и их можно свободно чередовать. В командах **Dimension** и **Declare** следует задавать имя массива и его размерность. Некоторые команды и функции FoxPro могут сохранять результат в массиве. Если заданный массив не существует, то следующие команды и функции могут автоматически создать массив — `average`, `acopy( )`, **Append From Array**, `adir( )`, **Calculate**, `afields( )`, **Copy To Array**, **Scatter**, **Select (SQL)**, **Sum**.

Имена массивов могут иметь в длину до 10 символов и включать в себя алфавитные символы, символы подчеркивания и цифры. Имя массива не может начинаться с цифры или содержать встроенные пробелы. Для создания одномерного массива укажите один индекс, который задает число строк в массиве. Для создания двумерного массива укажите пару индексов. Первый индекс обозначает число строк в массиве, второй — число столбцов. Индексы массива всегда начинаются с 1. В следующем примере создается одномерный массив с именем `deptnumber` и двумерный массив с именем `taxrates` с десятью строками и пятью столбцами:


```
Dimension deptnumber(10)
Dimension taxrates(10,5)
```

С помощью одной команды **Dimension** или **Declare** можно создать несколько массивов. Массивы приведенного выше примера может создать команда:

```
Dimension deptnumber(10), taxrates(10,5)
```

Функции FoxPro для работы с массивами. Для работы с массивами FoxPro поддерживает ряд функций. Перечень этих функций и их использование приведены в табл. 5.5.

Таблица 5.5. **Функции для работы с массивами**

Функция	Описание
`Adel()`	Удаляет из массива элемент, строку или столбец
`Adir()`	Помещает в массив информацию о соответствующих файлах
`Aelement()`	По индексу строки и столбца возвращает номер элемента
`Afields()`	Помещает в массив информацию о файле данных
`Ains()`	Включает в массив элемент, строку или столбец
`Alen()`	Возвращает число элементов, строк или столбцов в массиве
`Ascan()`	Выполняет поиск в массиве в памяти выражения
`Asort()`	Сортирует массив-переменную в памяти по возрастанию или в обратном порядке
`Asubscript()`	Возвращает индексы элемента (строку и столбец) по его номеру

Ссылки на элементы массива. К элементу массива можно обращаться по индексу его строки и столбца или по номеру эле-

мента. Индексы массива представляют собой числа или числовые выражения, которые задают место расположения элемента массива. Первый индекс задает место расположения элемента в строке, второй — место расположения элемента в столбце. Например, индексы 1,1 задают элемент массива, находящийся в первом столбце и первой строке. Индексы 2,5 задают элемент во второй строке и пятом столбце массива. В одномерном массиве номер элемента идентичен его индексу строки. Номер элемента в двумерном массиве определяется подсчетом строк. Предположим, например, что вы создали следующий массив 3 × 3:

a b c
d e f
g h i

Номерами элементов для *a*, *b* и *c* будут 1, 2 и 3. Номерами элементов для *d*, *e* и *f* будут 4, 5 и 6 и т. д. Имеются две полезные функции для работы с массивами — `aelement()` и `asubscript()`:

- функция `aelements()` возвращает номер элемента по заданным индексам столбца и строки;
- функция `asubscript()` возвращает индекс строки и столбца по заданному номеру элемента массива.

Присваивание значений элементам массива. Для присваивания значений элементам массива используются команда **Store** и оператор присваивания «=». Для присваивания значения отдельному элементу массива используются индекс элемента (одномерный массив) или его индексы (двумерный массив). В приведенных ниже примерах буквы A—F присваиваются элементам массивов 2 × 3 с именами `alpha` и `beta`. Для присваивания элементам массива значений A—F используется команда **Store** и оператор =.

```
Dimension alpha(2,3), beta(2,3)
Store 'a' To alpha(1,1)
Store 'b' To alpha(1,2)
Store 'c' To alpha(1,3)
Store 'd' To alpha(2,1)
Store 'e' To alpha(2,2)
Store 'f' To alpha(2,3)
```



```
beta(1,1) = 'a'
beta(1,2) = 'b'
beta(1,3) = 'c'
beta(2,1) = 'd'
beta(2,2) = 'e'
beta(2,3) = 'f'
Display Memory Like alpha
Display Memory Like beta
```

Можно также присвоить элементам значения, используя номера элементов:

```
Dimension gamma(2,3)
Store 'a' To gamma(1)
Store 'b' To gamma(2)
Store 'c' To gamma(3)
Store 'd' To gamma(4)
Store 'e' To gamma(5)
Store 'f' To gamma(6)
Display Memory Like gamma
```

Изменение размерности массивов. Объем или размерности массива можно изменить, снова используя команды **Dimension** или **Declare**. Можно увеличить или уменьшить объем массива, одномерный массив преобразовать в двумерный, а двумерный свести к одномерному. Если вы увеличите число элементов массива, то содержимое всех элементов исходного массива копируется с соответствии с порядком элементов в массив с измененной размерностью. Дополнительные элементы массива инициализируются логическим значением `false` (`.f.`).

Если вы уменьшаете число элементов, то массив усекается в соответствии с порядковым номером элемента. Конкретные строки и столбцы можно удалить из массива с помощью функции `adel( )`.

Передача данных между массивами и файлами данных. В FoxPro имеется несколько команд, которые облегчают передачу данных из файла данных в массив и обратно. Команда **Scatter** пересылает данные в массив из отдельной записи файла данных. Команда **Copy To Array** передает в массив данные из последовательности записей, **Select (SQL)** может передавать в массив результат запроса. Данные из массива можно переслать в отдельную запись файла данных с помощью команды **Gather**. Команда **Append From Array** добавляет к файлу данных новые

записи и заполняет записи данными из массива. Команда **Insert (SQL)** присоединяет к базе данных одну новую запись и заполняет ее данными из массива. Команды **Scatter** и **Copy To Array** пересылают данные из файла данных в массив.

5.2. Выражения, операторы, функции

Выражения

Как и в других ЯП, выражения строятся из констант, переменных и функций. В качестве переменных выступают переменные памяти, поля файлов данных и массивы. Наиболее часто используемые константы относятся к числовому, логическому, символьному и типу даты:

- числовые константы представляют собой десятичные числа, записанные в обычной или экспоненциальной форме (см. примеры в табл. 5.1, 5.4);
- логические константы — это значения «истина» (`.t.`), или «ложь» (`.f.`);
- литералы — константы символьные или типа даты используют следующие ограничители:
  — `" "`, `' '`, `[ ]` — для представления строковых констант;
  — `[ ]` – для представления констант даты.

Рассмотрим пример выполнения в диалоговом режиме последовательности действий над переменными оперативной памяти и выражениями (рис. 5.5):

- A и B определяются как символьные переменные, поскольку в них вводятся строки (*1*);
- C также становится символьной, так как в нее записывается конкатенация строк и литералов (*2*);
- затем переменные распечатываются (*3*), а также выводится иллюстрация к функции `substr` (*4*);
- в A и B записываются соответственно истинное (5 > 3) и ложное (5 < 3) логические выражения (*5*), поэтому они переопределяются как логические переменные, затем результат операции над ними записывается в C, которая также принимает логический тип (*6*). Далее A, B, C распечатываются (*6*);

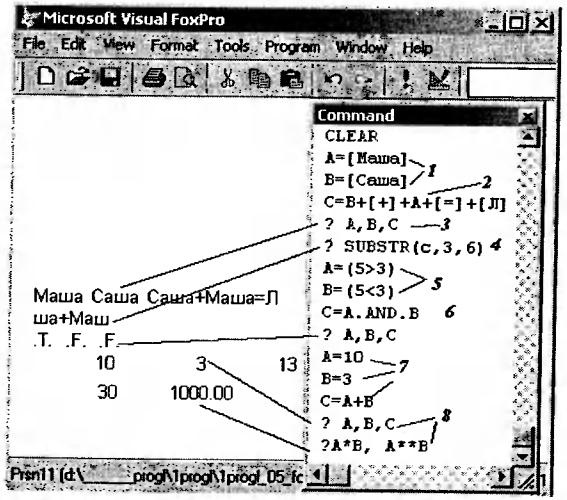


Рис. 5.5. Некоторые операции и выражения над переменными памяти:
1 — ввод символьных переменных; *2* — конкатенация переменных и литералов; *3* — распечатка символьных значений; *4* — функция `substr`; *5* — создание логических значений из выражений; *6* — операция над логическими переменными и их вывод; *7* — ввод числовых переменных и операция над ними; 8 — распечатка числовых переменных и результатов операций

- A и B переопределяются как числовые переменные и C, как их сумма, также принимает числовой тип (*7*). Значения A, B и C распечатываются (*8*);
- осуществляются операции умножения и возведения в степень над переменными A и B и распечатываются их результаты (*8*).

Операторы

Операторы, используемые в FoxPro, разделяются на четыре основные категории — логические, арифметические, операторы отношения и строковые (табл. 5.6):

- логические операторы работают с логическим типом данных и возвращают логические значения;
- арифметические (числовые) операторы работают с числовыми величинами и функциями. Результатом является числовая величина;

- операторы отношения могут сравнивать переменные одного типа для различных типов данных и возвращают логическое значение (.t./.f.)
- строковые (символьные) операторы соединяют несколько строк в одну.

Таблица 5.6. **Операторы ЯП FoxPro**

Тип операторов	Знак	Использование
Логические операторы (в порядке старшинства)	()	Круглые скобки, используются для группирования выражений
	.not., !	Логическое отрицание НЕ
	.and.	Логическое И
	.or.	Логическое ИЛИ
	#	Знак может использоваться для «Исключающего ИЛИ» (XOR)
Арифметические операторы (в порядке старшинства)	()	Круглые скобки, используются для группирования выражений
	**, ^	Возведение в степень
	*, /	Умножение и деление
	%	Взятие модуля (остаток от деления)
	+, -	Сложение и вычитание
Дата и время	+	Сложение величин
	-	Вычитание величин
Операторы отношения	<	Меньше чем
	>	Больше чем
	=	Равно
	#, <>	Не равно
	<=	Меньше или равно
	>=	Больше или равно
	$	Вхождение подстроки в строку или сравнение символьных строк
	==	Сравнение на полное совпадение строк. Установка set exact игнорируется
Строковые операторы	+	Сцепление строк (строки соединяются в одну) — рис. **5.5, *2***
	-	Сцепление строк (с удалением «хвостовых» пробелов)

Функции

Visual FoxPro предусматривает целый ряд функций, примеры которых приводятся в табл. 5.7 (более полное описание можно найти, например, в [7]). Основные типы функций следующие:

- числовые (арифметические) обрабатывают и возвращают числовые значения;
- символьные (строковые) работают с символьными данными;
- функции преобразования типов данных из одних в другие;
- логические функции проверяют те или иные свойства или условия;
- функции даты/времени создают и обрабатывают хронологические данные;
- COM-функции предназначены для доступа к COM-объектам и серверам;
- XML-функции обеспечивают обмен данными через формат XML.

Данная классификация, конечно, не идеальна — например, between(), будучи логической по значению, на входе имеет два символьных или числовых выражения; asc() — числовая по значению, также имеет на входе символьное выражение и т. п.

В табл. 5.7 используются следующие сокращения: <Симв_в> означает символьное выражение, <Числ_в> — числовое выражение, <Лог_в> — логическое выражение, <Дата_в> — выражение типа даты. В случаях, когда тип данных не имеет значения, используется термин <Выражение>.

Таблица 5.7. **Некоторые функции СП FoxPro**

Функция	Синтаксис	Действие
Числовые (арифметические)		
Ceiling	ceiling (<Числ_в>)	Возвращает ближайшее целое, большее или равное <Числ_в>. Положительные числа с дробными значениями округляются до ближайшего целого числа, а отрицательные числа с дробными значениями — до следующего ближайшего к нулю числа
Date	date()	Возвращает текущую системную дату
Day	day(<Дата_в>)	Возвращает номер дня в месяце, который соответствует выражению в дате

Продолжение табл. 5.7

Функция	Синтаксис	Действие
Floor	floor(<Числ_в>)	Возвращает ближайшее целое значение, меньшее или равное числовому выражению. Все положительные числа с дробными значениями округляются до ближайшего меньшего целого, а все отрицательные числа с дробными значениями — вниз до ближайшего числа, большего нуля
Inkey	inkey([<Числ_в>])	Возвращает целое значение от 0 до 255, соответствующее десятичному ASCII-коду нажатой клавиши. Если никакая клавиша не нажималась, возвращается нулевое значение
Int	int(<Числ_в>)	Возвращает целую часть <Числ_в>, округления не происходит
Mod	mod(<Числ_в-1>, <Числ_в-2>)	Возвращает остаток от деления <Числ_в-1> на <Числ_в-2>
Max	max(<Выражение-1>, <Выражение-1>[, <Выражение-3>...])	Возвращает максимальное значение из списка. Выражения должны относиться к одному типу данных
Min	min(<Выражение-1>, <Выражение-2> [,<Выражение-3>...])	Возвращает выражение с минимальным значением из списка. Выражения должны относиться к одному типу данных
Rand	rand([<Числ_в>])	Возвращает случайное число между 0 и 1. Необязательное <Числ_в> используется как «зерно" (стартовое значение), отличное от принятого по умолчанию, для генерации случайных чисел
Round	round(<Числ_в-1>, <Числ_в-2>)	Округляет число, указанное в <Числ_в-1>. <Числ_в-2> указывает, до какого десятичного знака должно идти округление
Rtod	rtod(<Числ_в>)	Преобразует радианы (<Числ_в>) в градусы
Sign	sign(<Числ_в>)	Возвращает числовое значение, представляющее знак числового выражения. Если <Числ_в> положительно — значение «1", если отрицательно, то значение «–1".

Продолжение табл. 5.7

Функция	Синтаксис	Действие
Time	time([<Числ_в>])	Возвращает текущее системное время в формате чч:мм:сс или чч:мм:сс am|pm
Строковые (символьные) функции		
Asc()	asc(<Симв_в>)	Возвращает десятичный ASCII-код самого левого символа в <Симв_в>
At()	at(<Симв_в-1>, <Симв_в-2>, [<Числ_в>])	Ищет вхождение <Симв_в-1> в <Симв_в-2> и возвращает целое значение — начальную позицию <Симв_в-1>. Если <Симв_в-1> не найдено, возвращает ноль
Chr()	chr (<Числ_в>)	Возвращает символ, десятичный ASCII-код которого соответствует <Числ_в>
Len()	len(<Симв_в>)	Возвращает длину <Симв_в>, которое может быть мемо-полем, в этом случае возвращается длина хранимого в мемо-поле текста
Lower()	lower(<Симв_в>)	Преобразует все заглавные буквы, находящиеся в <Симв_в>, в строчные. Функция не влияет на неалфавитные символы
Ltrim()	ltrim(<Симв_в>)	Удаляет ведущие пробелы из <Симв_в>
Occurs()	occurs(<Симв_в-1>, <Симв_в-2>)	Возвращает целое число, которое указывает, сколько раз <Симв_в-1> встречается в <Симв_в-2>
Right()	right(<Симв_в> | <Переменная>, <Числ_в>)	Возвращает самую правую часть символьной строки <Симв_в> или переменной памяти <Переменная>. Для указания числа возвращаемых символов используется <Числ_в>
Rtrim()	rtrim(<Симв_в>)	Удаляет из строки символов те пробелы, которые находятся в ее конце
Str()	str(<Числ_в-1> [, <Числ_в-2> [, <Числ_в-3>]])	Преобразует числовое выражение в символьное, где <Числ_в-1> означает числовое выражение, которое надо преобразовать в символьную строку
Substr()	substr(<Симв_в>, <Числ_в-1>[, <Числ_в-2>])	Выделяет подстроку из <Симв_в>; <Числ_в-1> — начальная позиция в выражении, <Числ_в-2> — число символов, которое надо выделить из выражения

Продолжение табл. 5.7

Функция	Синтаксис	Действие
Type()	type(<Симв_в>)	Возвращает одиночный символ, указывающий тип <Симв_в>: «C» — символьный; «L» – логический; «N» — числовой, «D» — дата, «M» — мемо, «U» — неопределенный тип
Upper()	upper(<Симв_в>)	Преобразует в заглавные буквы все алфавитные символы <Симв_в>
Функции преобразования данных (из одного типа в другой)		
Chr()	chr(<Числ_в>)	Возвращает символ, десятичный ASCII-код которого соответствует <Числ_в>
Ctod()	ctod(<Симв_в>)	Возвращает значение типа Дата, соответствующее <Симв_в> в принятом по умолчанию формате даты (обычно мм/дд/гг)
Dtoc()	dtoc(<Дата_в>,[1])	Возвращает строку символов, содержащую дату, которая соответствует выражению даты. Необязательный аргумент «1» заставляет функцию вывести строку в формате ггггммдд
Val()	val(<Симв_в>)	Преобразует <Симв_в>, содержащее цифры, в числовое значение
Логические		
Between	between (<Выражение-1>, <Выражение-2>, <Выражение-3>)	Возвращает логическую истину (.t.), если <Выражение-1> больше или равно <Выражению-2> и меньше или равно <Выражению-3>; в противном случае функция возвращает логическую ложь (.f.). Выражения должны быть одного типа
Iif	iif(<Лог_в>, <Выражение-1>, <Выражение-2>)	Возвращает значение <Выражения-1>, если <Лог_в> истинно, или <Выражение-2>, если ложно. <Выражение-1> и <Выражение-2> должны относиться к одному типу данных
Isalpha	isalpha(<Симв_в>)	Возвращает .t., если первый символ <Симв_в> лежит в диапазоне a-z или A-Z. Значение .f. возвращается в том случае, когда <Симв_в> начинается с неалфавитного символа
Isdigit	isdigit(<Симв_в>)	Возвращает .t., если первый символ в <Симв_в> является цифрой (0–9)

Окончание табл. 5.7

Функция	Синтаксис	Действие
Islower	islower(<Симв_в>)	Возвращает .t., если первый символ в <Симв_в> является строчным алфавитным символом, и .f., если первый символ является каким-либо другим символом
Like	like(<Симв_в-1>, <Симв_в-2>)	Сравнивает два символьных выражения и возвращает .t., если в <Симв_в-1> является подстрокой <Симв_в-2>. Можно включать маски «*» и «?»
Дата/время		
Cdow()	cdow (<Дата_в>)	Возвращает название дня недели, соответствующее указанной дате
Cmonth()	cmonth(<Дата_в>)	Возвращает название месяца, соответствующее указанной дате
Date()	date()	Возвращает текущую системную дату
Day()	day(<Дата_в>)	Возвращает номер дня месяца, который соответствует выражению в дате
Time()	time([<Числ_в>])	Возвращает текущее системное время в формате чч:мм:сс (если set hours установлено на 24) или в формате чч:мм:сс am|pm (если set hours установлено на 12)

5.3. Команды языка FoxPro

В табл. 5.8 содержится список типов команд, с краткой характеристикой их некоторых представителей.

Таблица 5.8. **Обзор некоторых команд FoxPro для их различных типов**

Тип команд	Примеры команд	Результат выполнения
Режим выполнения команд		
Диалоговые	**Resume**	Заставляет программу продолжить выполнение со строки, следующей за той, на которой выпопнение программы было приостановлено (командой **Suspend**)
	Run	Выпопняет из среды FoxPro файл с расширением .com, .exe или .bat
	Continue	Продолжает поиск, начатый командой **Locate**

Продолжение табл. 5.8

Тип команд	Примеры команд	Результат выполнения
Пакетные (программные)	`Do Case... Endcase`	Выбирает один вариант действия из имеющегося перечня
	`Do While... Enddo`	Непрерывно выполняет команды между `Do While` и `Enddo` до тех пор, пока условие остается истинным
	`Exit`	Позволяет выйти из цикла `Do While`, `For`
Диалоговые и пакетные	`Clear`	Очищает экран
	`Browse`	Может вывести из файла данных на дисплей до 20 записей **(см. рис. 5.13)**
	`Edit`	Загружает полноэкранный редактор записей **(см. рис. 5.7, *а*)**.
Процедурность команд		
Процедурные	`If...Endif`	Если условие является истинным, то выполняются все команды, расположенные между `If` и `Endif`
	`Loop`	Переход на начало цикла `Do While ... Enddo`
Непроцедурные	`Find`	Устанавливает указатель записи на первую запись, содержащую индексный ключ, соответствующий выражению в команде
	`Sum`	Дает общую сумму полей, включающих числовые поля
	`Sort`	Создает и сортирует копию файла данных
Назначение команд		
Установление режимов и опций	`Set Exact`	Определяет с какой точностью будут сравниваться две строки символов.
	`Set Format`	Открывает файл формата ввода-редактирования текущего файла данных
	`Set Color`	Определяет атрибуты цветов экрана
Управление вычислительным процессом	`Do, Return`	Вызывает командный файл и осуществляет возврат в вызывающую программу
	`Do`	См. выше `Do While <Условие> ... Enddo`, `Loop`, `Exit`, `Do Case ... Case ... Endcase`, `if ... Else ... Endif`
Создание БД	`Create`	Создает новый файл данных и определяет его структуру
	`Create Report`	Создает или изменяет файл с форматом отчета
	`Index`	Создает индексный файл, основанный на выражении (которое обычно является именем поля или комбинацией полей)

Продолжение табл. 5.8

Тип команд	Примеры команд	Результат выполнения
Загрузка/ выгрузка БД	`Append From`	Загружает (импортирует) файл БД из файла ОС
	`Copy To`	Выгружает (экспортирует) данные из файла БД в файл ОС
Ввод данных	`Get`	Вводит отдельное поле, строку, или пользовательскую переменную
	`Append`	Создает пустую запись в конце текущего файла и позволяет заполнение с применением пользовательского формата экрана (команды `Browse`, `Edit`)
Вывод данных	`Say`	Выдает на экран или принтер поле, данное или строку
	`?`	Выводит на экран значение выражения FoxPro **(см. рис. 5.5)**
	`Report`	Выдает поколонный отчет на экран или принтер с использованием файла формата отчета **(см. рис. 5.26)**
Навигация в БД	`Use, Select`	Позволяет осуществить выбор среди нескольких открытых файлов БД текущего
	`Go, Skip`	Изменяет положение указателя текущей записи путем прямого перехода к записи или путем пропуска некоторого числа записей
	`Set Relation To`	Устанавливает связь двух файлов БД так, что при перемещении указателя записи в основном файле синхронно перемещается указатель в зависимом
Работа с массивами	`Gather From`	Используется для переноса данных из массива в файл данных. Элементы массива переносятся, начиная с первого элемента массива в соответствующие записи файла данных
	`Scatter To`	Используется для перенесения данных из текущей записи файла данных или из переменной памяти в массив. Поля текущей записи переносятся, начиная с первого поля записи в соответствующие элементы массива

Рассмотрим далее некоторые из этих команд, в соответствии с их назначением.

Команды работы с файлами данных (файловые команды)

Здесь могут быть выделены (табл. 5.9):

- непроцедурные команды, которые действуют на файл в целом, причем теоретически одной командой могут быть об-

Таблица 5.9. **Основные файловые команды**

Команда	Результат выполнения
	Непроцедурные команды
Append From	Копирует записи из <Файла> и добавляет их в текущий файл данных. **For|While** определяют условие, которое должно выполниться для копирования записи
Average	Вычисляет среднее значение числовых полей, указанных в <Списке полей файла> **(см. рис. 5.9)**
Browse	Выводит из файла данных на дисплей до 20 записей. Если в записи содержится так много полей, что они не могут поместиться на экране, то **Browse** выводит только те поля, которые помещаются. Остальные поля можно увидеть, прокручивая экран вправо и влево с помощью мыши или клавиши <Tab>. В режиме **Browse** можно редактировать содержимое полей **(см. рис. 5.11)**
Calculate	Вычисляет значения, используя стандартные финансовые и статистические функции, например, Avg (<Числовое выражение >) — среднее значения выражения
Copy	Копирует файл данных или его часть в <Файл>
Copy To Array	Копирует в массив данные из полей файла данных. Для каждой записи файла данных первое поле сохраняется в первом элементе массива, второе поле — во втором и т. д.
Count	Считает число записей активного файла данных, которые соответствуют определенному условию **(см. рис. 5.9)**
Delete	Помечает указанные записи на удаление. Если **Delete** используется без номера записи, то на уничтожение помечается текущая запись
Edit	Загружает экранный редактор записей FoxPro **(см. рис. 5.5)**
List	Выдает содержимое файла данных **(см. рис. 5.5)**
Locate	Команда находит первую запись, соответствующую условию
Replace	Помещает в указанное поле новое значение. Можно одновременно заменить значения в нескольких полях, указав список полей и нужных значений (<Поле> **With** <Выражение>)
Report Form	Команда **Report Form** использует файл с формой отчета (созданный ранее командой **Create Report** — **см. рис. 5.24)** для создания отчета. Предполагается, что имя файла формата имеет расширение .frx, если не указано другое
	Процедурные команды
Go или **Goto**	Команды устанавливают указатель записи на нужную запись. **Go Top** переместит указатель в начало файла данных, а **Go Bottom** — в конец **(см. рис. 5.6)**
Skip	Перемещает указатель активной записи текущего файла. Если отсутствуют какие-либо параметры, то перемещение указателя осуществляется на одну запись вперед

работаны вообще все записи файла (если все они удовлетворяют указанным в команде условиям). Примерами являются команды **Edit**, **Replace**, **Report**, **Copy** и др.;

- процедурные, которые выполняют элементарное действие над файлом (команды **Skip**, **Go**).

Общий синтаксис непроцедурных файловых команд:

<Команда> [**FROM** { [**File**] <Файл> | [**Array**] <Массив> }]

[**Fields**<Список полей файла>] [<Интервал>]

[**While** <Логическое выражение-1>]

[**For**<Логическое выражение-2>]

[**To** { [**File**] <Файл> | [**Array**] <Массив> | <Переменная> | <Список переменных> | **Printer** }].

Здесь:

- <Интервал> (Scope — охват или диапазон) позволяет указать область действия **<Команды>** для работы с определенными записями в текущем файле данных и может быть определен следующим образом:
  - **All** — команда выполняется для всех записей в файле данных;
  - **Next <n>** — команда выполняется для области записей, которая начинается с текущей записи и завершается записью с номером <n>;
  - **Next 1** — команда работает с текущей записью;
  - **Record <n>** — работает с записью файла данных, имеющей физический номер <n>;
  - **Rest** — работает с областью записей, которая начинается с текущей записи и завершается последней записью в файле;
- **While** позволяет **<Команде>** работать с записями файла данных, пока <Логическое выражение-1> оценивается как истинное. Выполнение **<Команды>** прекращается, если встречена запись, не соответствующая <Логическому вы-

ражению-1> и все другие оставшиеся записи файла данных не обрабатываются;

- **For — <Команда>** распространяется только на записи, удовлетворяющие <Логическому выражению-2>. Каждая запись в файле данных проверяется с помощью выражения **For**. Оператор интервала и выражения **For**, **While** могут быть использованы в одной и той же команде FoxPro. Если заданы оба выражения **For** и **While**, то **While** имеет более высокий приоритет;
- **Fields** <Список полей файла>; указывает перечень полей файла данных, на которые распространено действие данной команды (см. рис. 5.7, *а*, команда **List**);
- **From** — источник входной информации для команды, если необходимо — файл или массив (например, команда **Append From**);
- **To** — контейнер записи результата, если необходимо — файл, массив, переменные и пр. (например, команда **Copy To**).

Примеры команд

```
Use personal
    Edit For sex = [M] .and. year > 1985
```

позволяет выбрать для редактирования записи с мужчинами, родившимися после 1985 г.

```
Use personal
    Edit While year = 1985
```

должна будет выбрать для редактирования записи, пока год рождения в них равен 1985 г.

Как только появится 1987 или 1986, работа команды завершится. Очевидно, все это имеет смысл только в том случае, если записи отсортированы по году рождения или включен соответствующий индекс. Кроме того, указатель текущей записи должен быть установлен на первую запись с годом рождения 1985 (например, командой **Seek 1985**).

Просмотр данных с использованием диалоговых команд. Команда **?** может применяться для вывода информации из файла данных и оперативной памяти. Рассмотрим несколько примеров «навигационных команд» и их результатов (рис. 5.6).

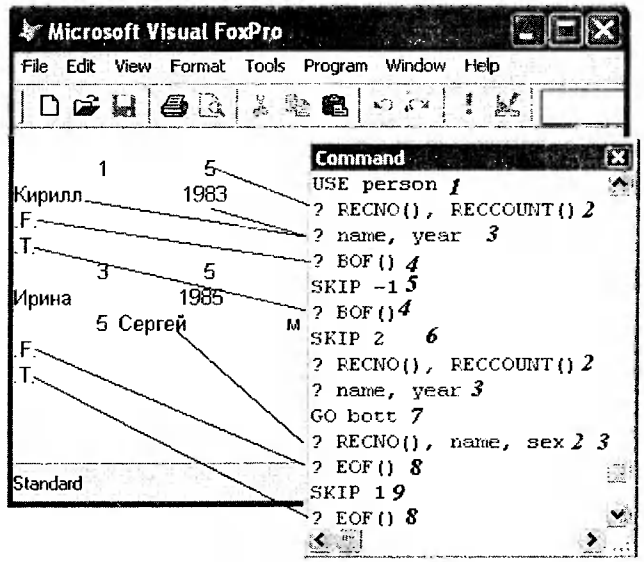


Рис. 5.6. Некоторые операции с файлом данных и наблюдение за ними с помощью диалоговых команд:
1 — открытие файла Person; *2* — вывод значений функций; *3* — вывод значений полей; *4*— значение функции Bof() (начало файла); *5* — шаг «назад на одну запись»; 6 — шаг «вперед на две записи»; *7* — переход в конец файла; *8* — значение функции Eof() (конец файла); *9* — шаг «вперед на одну запись»

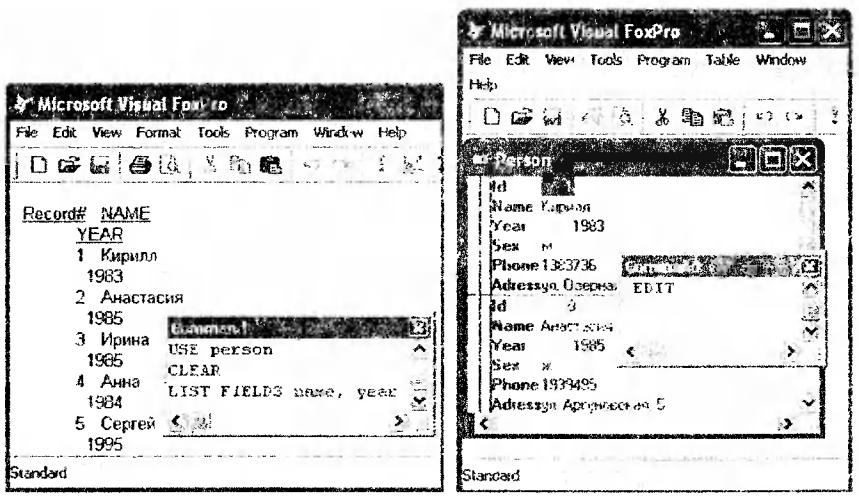
Здесь изображено рабочее поле экрана (слева) и окно командных строк (справа). Соответствие команд и отображаемой информации задается стрелками:

- команда **Use** открывает файл данных Person (*1*);
- командой **?** просматриваются значения функций (*2*), определяющих номер текущей записи Recno() и общее число записей в файле Reccount();
- просматриваются значения полей текущей записи (*3*);
- функция Bof() имеет значение .f. (false), это говорит о том, что начало файла (**Begin Of File**) не достигнуто (*4*);
- командой **Skip -1** делается попытка «шагнуть за начало файла» (*5*), что приводит к установке Bof() в значение .t. (true);
- командой **Skip 2** осуществляется переход к 3-й записи (*6*);
- командой **?** просматриваются значения функций (*2*) и значения полей записи (*3*);

- командой **Go Bottom** задается переход к последней записи файла (*7*);
- командой **?** просматриваются значение функции `Recno()`, а также значения полей текущей записи;
- функция `Eof()` имеет значение `.f.` (`false`), это означает, что конец файла (**End Of File**) не достигнут (*8*);
- командой **Skip 1** делается попытка «шагнуть за конец файла» (*9*), что приводит к установке `Eof()` в значение `.t.` (`true`).

Просмотр данных. Просмотреть загруженные данные можно с использованием команды **List** (рис. 5.7, *а*):

- **List** — просмотр всего открытого файла;
- **List** <Имя поля-1>, <Имя поля-K> — просмотр 1- и *K*-го полей;
- **List Record 3** — просмотр отдельной записи.



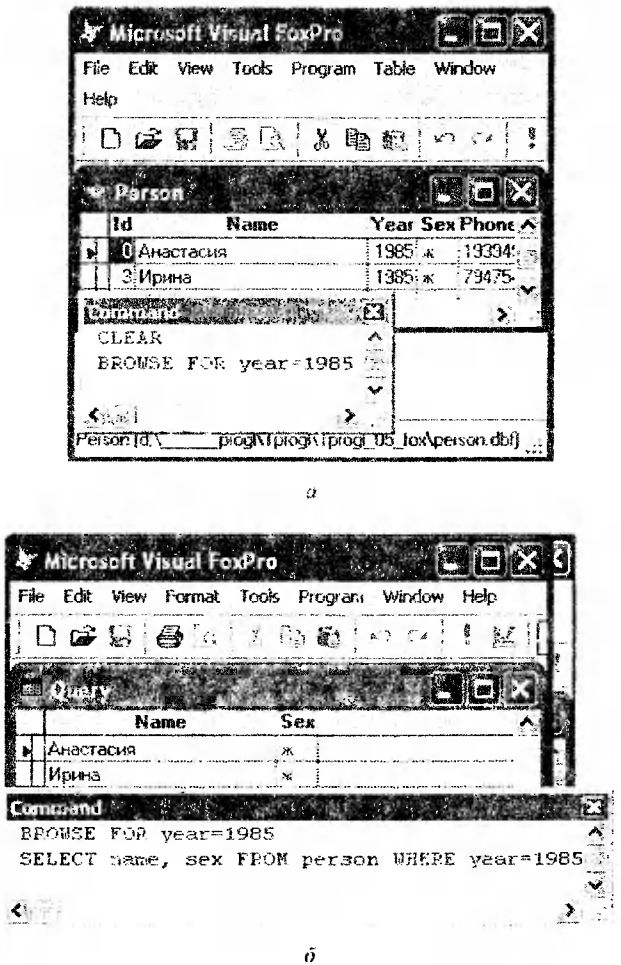
а *б*

Рис. 5.7. Выборочный просмотр полей записей файла командой **List** (*а*); режим просмотра и редактирования файла по записям командой **Edit** (*б*)

Конструкция **List Off** позволяет исключить номера записей из листинга.

Команды **Edit** и **Browse** позволяют осуществлять просмотр и редактирование записей файла данных.

Browse отображает файл БД в форме таблицы, в которой строки соответствуют записям, а столбцы — полям данных (рис. 5.8, *а*). Окно просмотра обычно недостаточно велико, чтобы дать возможность увидеть всю таблицу сразу. Для того чтобы увидеть различные части таблицы, нужно прокрутить окно просмотра по горизонтали и по вертикали. Текущая (активная) запись помечается символом ▸.



а

б

Рис. 5.8. Просмотр с использованием логических выражений: *а* — команда **Browse**; *б* — команда **Select (SQL)**

Команда **Edit** позволяет увидеть все поля в одном окне. В этом режиме поля каждой записи располагаются одно под другим (см. рис. 5.7, *б*).

Просмотр данных с учетом логических выражений. С этой целью используются команды, содержащие логические выражения: **List For** <Имя поля> = [<Значение>], например,

```
List For sex = [M]
```

Кроме операторов сравнения на равенство (=) можно работать с другими логическими операторами: <, >, <=, >=. Значения символьных полей заключаются в кавычки.

FoxPro позволяет просматривать и копировать данные, используя сложные логические выражения с операторами (связками) .and., .or., .not.:, **List For** year >= 1990 **.and.** sex = [f].

Логические связки имеют три уровня приоритета: .not.; .and.; .or. .

На рис. 5.8 приводятся примеры выборочного просмотра записей файла командой **Browse** (рис 5.8, *а*) и командой SQL **Select** (рис. 5.8, *б*). Отметим, что команды языка SQL [4, 7] поддерживаются системой FoxPro в качестве альтернативы аналогичным командам и операторам внутреннего языка.

Некоторые другие операции, относящиеся к файлам данных

Просмотр списка файлов. Чтобы вывести на экран дисплея список всех имеющихся файлов, используют команду **Dir**.

На экране появляется список файлов с расширением. Расширение, добавляемое системой, служит для группировки файлов по типам:

.idx — индексный файл;
.prg — файл команд;
.dbf — файл данных;
.frt — форматы отчетов;
.bak — резервная копия файла.

Копирование файла. Чтобы скопировать файл, нужно выполнить две операции.

Поместить файл, который нужно скопировать, в рабочую область, используя команду **Use**:

```
Use <Имя файла>,
```

затем скопировать файл командой

```
Copy To <Имя нового файла>
```

Удаление файла. Удалить файл, который является текущим рабочим файлом, невозможно. Чтобы добиться цели, следует закрыть текущий файл. Все файлы данных можно закрыть по команде **Use**. Имя файла не указано, закрывается активный (текущий) открытый в настоящее время файл. Просмотреть оставшиеся файлы можно по команде **Dir**:

```
Use
Delete File <Имя файла с расширением>
Dir
```

Копирование структуры файла. Для копирования структуры файла следует набрать команды:

```
Use <Имя файла>                          открыть исходный файл
Copy Structure To <Имя нового файла>     скопировать структуру
Use <Имя нового файла>                   перейти к новому файлу
Display structure                        просмотр структуры
```

Добавление записей из другого файла. Вызвать исходный файл. Добавить записи к исходному файлу из второго:

```
Append From <Имя второго файла>
```

Удаление записей из файла. Удаление записи осуществляется в два этапа:

- логическое удаление — запись, предназначенная для удаления, маркируется (при просмотре командами **List**, **Browse** она помечается символом «*»). Если обнаружена ошибка, запись можно восстановить (командой **Recall**);
- физическое удаление — файл «упаковывается» (команда **Pack**) и после этого восстановить записи (средствами FoxPro) нельзя.

```
Use <Имя файла>
Edit — просмотр файла.
Delete Record <Номер удаляемой записи>.
```

Маркируются все записи, которые нужно удалить. Для восстановления ошибочно помеченной на удаление записи можно использовать команду:

```
Recall Record <Номер восстанавливаемой записи>
Edit — просмотр файла
Pack.
```

В командах **Delete** и **Recall** для маркировки записей можно использовать диапазоны и логические выражения:

```
Go Top
Delete All
Delete For <Логическое выражение>
Recall For <Логическое выражение>
Recall All
```

Подсчет записей и данных. В FoxPro включены команды, с помощью которых можно получить количественную информацию о данных, содержащихся в файлах:

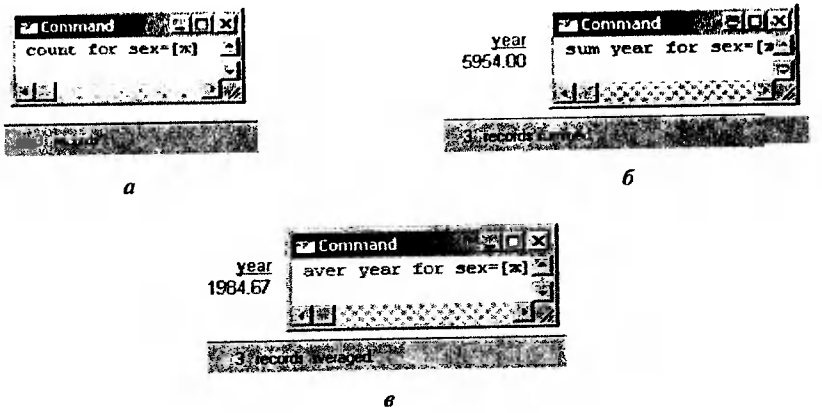
- если нужно найти число записей, удовлетворяющих заданному критерию, то используют команду **Count**:

```
Use <Имя файла>
Count For <Логическое выражение>
```

- команда **Sum** предназначена для суммирования содержимого конкретного поля. Она может применяться в сочетании с логическими выражениями для получения суммы определенных записей файла:

```
Use <Имя файла>
Sum <Имя поля> For <Логическое выражение>
```

Вернемся к базе данных Person. На рис. 5.9 приведены результаты этих команд. Здесь команда размещена в командном окне, количество записей — на статусной строке, результат — в рабочей зоне. Конечно, подсчет суммы годов рождения особей женского пола (рис. 5.9, *а*) не более чем шутка (хотя результат может использоваться, например, для определения среднего возраста женщин, представленных в БД). Но это и лишнее напоминание читателю, что машине все равно, что суммировать (поле — числовое) — она не думает. Думать должен программист (или хотя бы пользователь).



а *б* *в*

Рис. 5.9. Результаты команд **Count** (*а*), **Sum** (*б*) и **Aver** (*в*)

Между прочим, вычисление среднего значения числового поля тоже предусмотрено в FoxPro (рис. 5.9, *в*).

Команды структурного программирования

Эти команды (табл. 5.10) играют роль операторов языка структурного программирования и реализуют циклы и ветвления в программах

Команда Do While ... Enddo выполняет последовательность команд в теле цикла. Синтаксис команды:

```
Do While <Логическое выражение>
    <Команды>
    [Loop]
    [Exit]
Enddo
```

Параметры:

- <Логическое выражение> — до тех пор, пока оно имеет значение «истина» (.t.), выполнение <Команд> будет повторяться;
- <Команды> — образуют тело цикла;
- **Loop** — осуществляет возврат управления в заголовок цикла;

Таблица 5.10. **Процедурные команды FoxPro**

Команда	Действие
Do while ... Enddo	Выполняет набор команд внутри цикла «Пока»
For Each ... Endfor	Выполняет последовательность команд для каждого элемента массива или коллекции
For ... Endfor	Выполняет последовательность команд определенное количество раз
Scan ... Endscan	Исполняет блок команд для каждой записи текущего файла данных при выполнении определенных условий
Loop	Возвращает управление в заголовок цикла **Do While**, **For** или **Scan**
Exit	Осуществляет выход из цикла **Do While**, **For** или **Scan**
If ... Endif	Выполняет последовательности команд по условию
Do Case ... Endcase	Выполняет первую из альтернативных последовательностей команд, для которой условие выполнения равно «истине» (**.t.**)

- **Exit** — задает выход из цикла и передачу управления первой команде, расположенной после **Do While ... Enddo**. Обе эти команды могут располагаться где угодно между **Do While** и **Enddo**.

Рассмотрим примеры на вычисление $y = N!$ ($y = 1 \cdot 2 \cdot 3 \cdot \ldots \times \times N$) для $N = 100$ с использованием команд **Loop** и **Exit** и без них.

1. Без использования этих команд:

```
y=1
N=100
i=1
Do While i<=N
     y = y*i
     i=i+1
Enddo
```

2. Используя команду **Exit**:

```
y=1
N=100
i=1
Do While .t.
     y = y*i
     i=i+1
     If i>N
          Exit
     Endif
Enddo
```


3. Используя команды **Exit** и **Loop**:

```
y=1
N=100
i=1
Do While .t.
     y = y*i
     i=i+1
     If i<=N
          Loop
     Endif
     Exit
Enddo
```

Команда For Each исполняет некоторую последовательность команд для каждого элемента массива FoxPro или коллекции (множества). Синтаксис (сокращенный):

```
For Each <Переменная> In <Группа>
     <Команды>
     [Exit]
     [Loop]
Endfor | Next [<Переменная>]
```

Параметры:

- <Переменная> — переменная или элемент массива;
- <Группа> — массив или коллекция VFP (или OLE);
- <Команды> — последовательность команд, которая должна быть выполнена для всех элементов <Группы>;
- **Exit** и **Loop** — команды нарушения естественного порядка цикла.

Пример. Создается массив cMyArray из пяти элементов. Затем содержимое 3-й записи файла Person помещается в этот массив (команда **Gather**, см. табл. 5.8). После этого команда **For Each ... Endfor** осуществляет вывод элементов массива (см. рис. 5.10), содержащих поля этой записи.

```
Dimension cMyArray(5)
Clear All
Use Person
Go 3
Gather To cMyArray
```

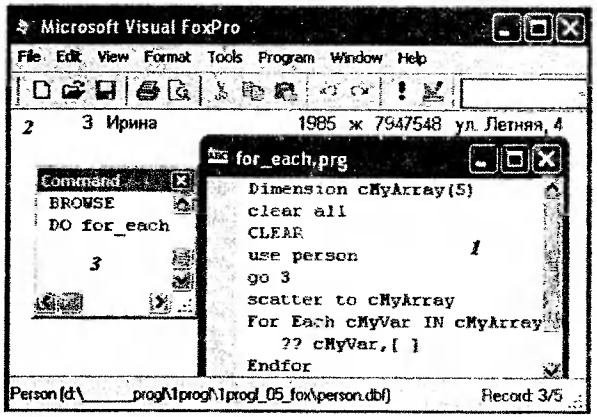


Рис. 5.10. Команда `For Each`:
1 — текст программы в окне редактора; *2* — вывод результата в основном окне; *3* — командное окно

```
For Each cMyVar In cMyArray
     ?? cMyVar+[ ]
Endfor
```

Команда For ... Endfor выполняет некоторую последовательность команд определенное число раз. Синтаксис (сокращенный):

```
For <Переменная> = <Значение-1> To <Значение-2>
[Step <Приращение>]
     <Команды>
     [Exit]
     [Loop]
Endfor
```

Параметры:

- <Переменная> — величина, играющая роль счетчика и изменяющаяся на каждом шаге. Не обязательно должна быть определена до начала цикла;
- <Значение-1>, <Значение-2> — задают начальное и конечное значения <Переменной>. Считываются только при входе в цикл и их дальнейшие изменения не влияют на выполнение цикла;
- **Step** <Приращение> — определяет величину, на которую изменяется <Переменная> на каждом шаге цикла.
- <Команды> — образуют тело цикла.

Пример. Команды **For ... Endfor** образуют цикл, в котором распечатываются числа от 1 до 10 с использованием команды ?:

```
For gnCount = 1 To 10
     ? gnCount
Endfor
```

Команда Scan ... Endscan проверяет (сканирует) записи файла данных и выполняет группу команд, если запись удовлетворяет необходимым условиям:

```
Scan [<Интервал>] [For <Логическое выражение-1>]
     [While <Логическое выражение-2>]
     [<Команды>]
     [Loop]
     [Exit]
Endscan
```

Параметры:

- <Интервал> — описывает охват записей, которые будут просматриваться (варианты: `All`, `Next` <Число записей>, `Record` <Номер записи> и `Rest`);
- **For** <Логическое выражение-1> — команда выполняется только для тех записей, для которых <Логическое выражение-1> принимает значение (`.t.`).
- **While** <Логическое выражение-2> — команда выполняется до тех пор, пока <Логическое выражение-2> имеет значение (`.t.`).
- <Команды> — блок команд тела цикла, выполняемых над записью файла данных;
- **Loop**, **Exit** — команды, нарушающие естественную последовательность команд тела цикла выходом из него или переходом к заголовку;
- **Endscan** — конец процедуры **Scan**.

Пример. Просмотреть файл данных `Person` и напечатать часть полей записей, где поле `sex` содержит литерал [ж] (рис. 5.11).

```
Clear All
Use Person
     Scan For sex=[ж]
          ? name, phone, sex, adress
     Endscan
```

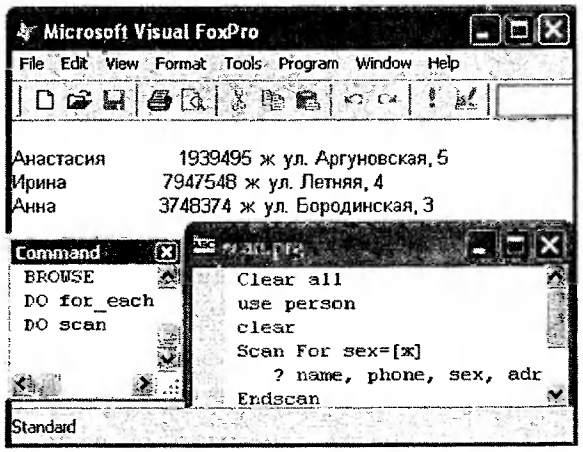


Рис. 5.11. Команда **Scan**

Команда If ... Endif осуществляет ветвления процесса вычислений на основе вычисленного значения логического выражения. Синтаксис команды:

```
If <Логическое выражение> [Then]
     <Команды-1>
[Else
     <Команды-2>]
Endif
```

Если <Логическое выражение> истинно (.t.), то выполняются <Команды-1>, если (.f.), то <Команды-2>.

Пример. Написать последовательность действий для решения следующей задачи, в которой значение y зависит от x следующим образом:

$$y = \begin{cases} \sin x, & x < -1; \\ 1 - \sin x, & -1 \le x < 0; \\ 1 - \cos x, & 0 \le x \le 1; \\ \cos x, & x > 1. \end{cases}$$

Условный оператор для реализации алгоритма может быть организован так:

```
If x < -1 Then
     y = sin(x)
Else
     If (x >= -1) .and. (x < 0) Then
          y = 1 - sin(x)
     Else
          if (x >= 0) .and. (x <= 1)
               y = 1 - cos(x)
          Endif
     y = cos(x)
endif
```


Команда Do Case ... Endcase аналогична по действию оператору **Case** в языке Pascal. Выполняет первый блок команд, для которого условие исполнения принимает значение .t. Синтаксис **Do Case** следующий:

```
Do Case
     Case <Логическое выражение-1>
     <Команды-1>
     Case <Логическое выражение-2>
     <Команды-2>
     ...
Case <Логическое выражение-N>
     <Команды-N>
     [Otherwise
     <Команды>]
Endcase
```

Параметры:

- **Case** <Логическое выражение-i> <Команды-i>. Как только встречается первое истинное (.t.) Case-выражение, выполняется последовательность команд, оканчивающаяся следующим **Case** или **Endcase**. Управление затем передается на команду, следующую за Endcase. Если Case-выражение ложно (.f.), набор команд до следующего **Case** пропускается;
- **Otherwise** <Команды>. Если все CASE-выражения принимают значения «ложь» (.f.), выполняются команды, расположенные между **Otherwise** и **Endcase**.

Приведенный ранее пример с использованием команды **Do Case** будет выглядеть следующим образом:

```
Do Case
     Case x < -1
          y = sin(x)
     Case (x >= -1) .and. (x < 0)
          y = 1 - sin(x)
```

```
     Case (x >= 0) .and. (x < 1)
          y = 1 - cos(x)
     Otherwise
          y = cos(x)
Endcase
```

Рассмотрим также применение **Do Case** для следующей задачи: присвоить с использованием функции mod (см. табл. 5.7) строке S значение дня недели для заданного числа D при условии, что в месяце 31 день и 1-е число — понедельник.

```
Do case
     Case mod(D, 7)=1
          S = [понедельник]
     Case mod(D, 7)=2
          S = [вторник]
     Case mod(D, 7)=3
          S = [среда]
     Case mod(D, 7)=4
          S = [четверг]
     Case mod(D, 7)=5
          S = [пятница]
     Case mod(D, 7)=6
          S = [суббота]
     case mod(D, 7)=0
          S = [воскресенье]
Endcase
```

Процедуры и функции

Рассмотрим основные команды, позволяющие строить сложные программные проекты, — **Do** и **Procedure**.

Команда Procedure описывает процедуру, определенную пользователем в командном файле (.prg). Если процедура размещена в другом командном файле, необходимо выполнить команду **Set Procedure to** <Имя файла>. Синтаксис:

```
Procedure <Имя процедуры>
     [Parameters <Параметр-1> [,  <Параметр-2> ] ,...]
     <Команды>
     [Return[<Выражение>]]
[Endproc]
```

<Команды> — образуют тело процедуры.

Return — возвращает управление вызывающей программе. Если специфицировано <Выражение>, то оно описывает возвращаемое значение.

Endproc — является необязательным оператором.

Команда Do вызывает и исполняет программу или процедуру Visual FoxPro. Синтаксис команды:

```
Do <Программа-1> | <Процедура> [In <Программа-2>]
     [With <Список параметров>]
```

Параметры:

<Программа-1> — имя вызываемой программы. Если не указано расширение имени файла, Visual FoxPro будет отыскивать исполняемые файлы в следующем порядке:

- программный код (.exe);
- приложение (.app);
- откомпилированная программа на ЯП FoxPro (.fxp);
- программный файл (.prg);

<Процедура> — специфицирует имя вызываемой процедуры. Прежде всего, осуществляется поиск процедуры в файле вызывающей программы. Если процедура не найдена, поиск производится в файле, указанном в команде **Set Procedure**. Если указать **In** <Программа-2>, то поиск процедуры сначала будет осуществляться в файле <Программы-2>;

With <Список параметров> — специфицирует список параметров, передаваемых программе или процедуре.

Рассмотрим пример. Пусть необходимо вычислить факториал некоторого числа с использованием процедур. Вариант текста программы приводится ниже, а экран с выводом результата — на рис. 5.12.

```
* Программа с процедурой Factor
N=5                         && Присвоение аргументу значения
M=0
Do Factor With N, M         && Вызов процедуры
? [N= ], N, [N!= ], M       && Вывод результата (рис. 5.12, 3)
Procedure Factor            && Заголовок процедуры
Parameters a, b             && Список параметров
Do Case
     Case a=0               && Если аргумент равен нулю,
          b=1               && факториал равен 1
          Return
```

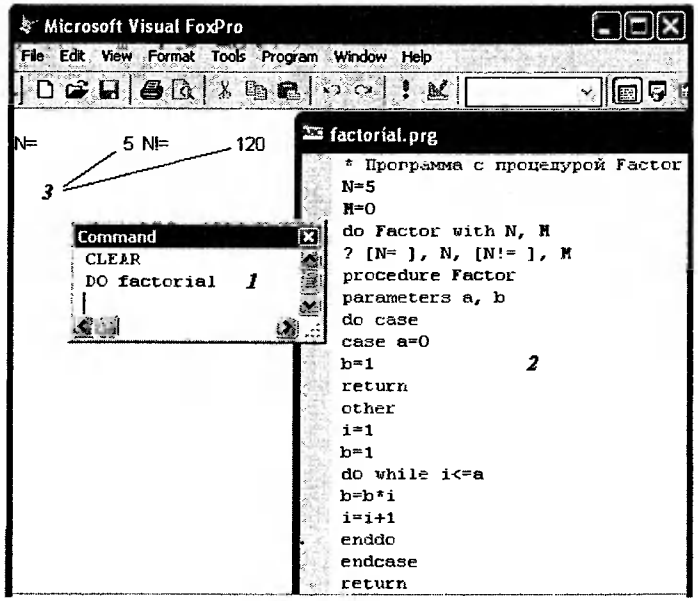


Рис. 5.12. Вызов и выполнение процедуры `Factor`:
1 — окно команд; *2* — окно редактирования программ; *3* — окно результатов

```
    Othercase
        i=1
        b=1
        Do While i<=a    && Начало цикла
            b=b*i
            i=i+1
        Enddo            && Конец цикла
Endcase
Return                   && Возврат в основную программу
```

Ошибки, возникающие при выполнении

Об ошибках, содержащихся в программе FoxPro сообщается, когда интерпретатор пытается выполнить ошибочную строку. Обработка пауз и звуковых сигналов осуществляется следующими диалоговыми командами:

- **Cancel** — немедленное завершение выполнения программы и возврат к режиму команды диалога;

- **Suspend** — приостановка выполнения программы и возврат к режиму команды диалога. Этот вариант полезен, когда осуществляется отладка программы;
- **Resume** — продолжение выполнения программы с точки, где она была приостановлена;
- **Ignore** — пропуск строки, в которой встретилась ошибка и выполнение следующей команды программы, или же, если была нажата клавиша <Esc> во время выполнения программы, игнорирование <Esc> и продолжение выполнения остальных команд программы.

5.4. Создание и модификация базы данных в программной среде FoxPro

Рассмотрим далее вкратце некоторые операции над базами данных, в процессе которых изменяется физическая структура БД — создаются или модифицируются новые файлы: данных, индексные, форматов экранов, отчетов и пр.

Краткие сведения об интерфейсе FoxPro

Ранее по тексту нам уже встречались компоненты интерфейса FoxPro. Рассмотрим основной экран интерфейса (рис. 5.13).

Строка заголовка — самая верхняя строка окна работающей программы (рис. 5.13, *1*). Она содержит имя активной прикладной программы или полное название (спецификацию) обрабатываемого в данный момент документа (файла).

Строка меню находится под строкой заголовка и включает следующие элементы:

- File (Файл);
- Edit (Правка);
- View (Вид);
- Tools (Инструментарий);
- Program (Программа);
- Window (Окно);
- Help (?).

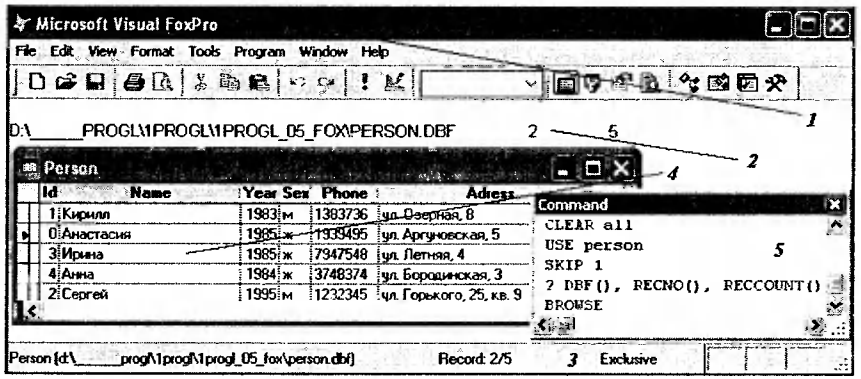


Рис. 5.13. Основной экран интерфейса:
1 — строка заголовка и основное меню; *2* — рабочее пространство (результаты команды `? Dbf(), Recno(), Reccount()`); *3* — статусная строка; *4* — окно пользователя (команда `Browse`); *5* — командное окно

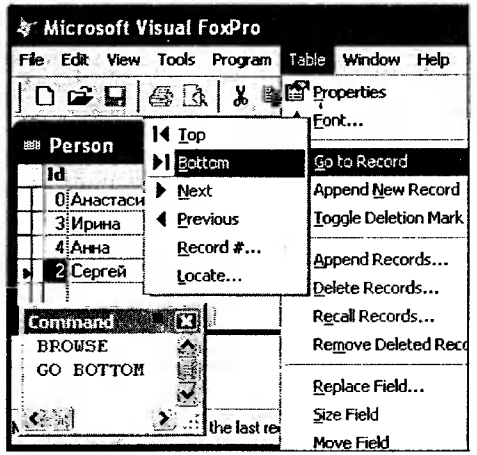
Каждому заголовку меню соответствуют свои меню директив и опций.

Из строки меню пользователь выбирает нужные ему операции для работы с БД. Операции разделены на группы по функциональной близости. Меню вызывается щелчком мыши на его имени в строке меню, а директивы из вызываемого меню — щелчком мыши на имени директивы в меню (рис. 5.14). Директивы, за названием которых следует многоточие, выполняются не сразу. Сначала открывается диалоговое окно (диалог), в котором пользователь задает некоторые параметры (опции) для директивы или выбирает разновидность директивы. Если директива изображена на экране не черным цветом, а бледно-серым, то это означает, что директива в данный момент недоступна пользователю.

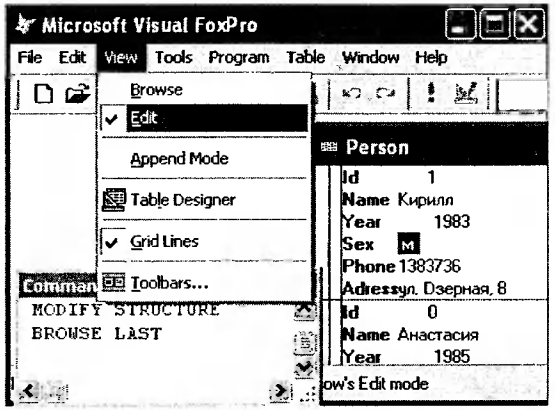
Рабочее пространство. После запуска FoxPro на мониторе появляется окно FoxPro, в котором присутствуют заставка и ряд служебных элементов, составляющих рабочее пространство программы: строка заголовка, строка меню, линейки прокрутки, статусная строка, рабочая зона, окно директив.

В центре экрана содержится рабочая зона пользователя (см. рис. 5.13, *2*), где он может создавать, редактировать и/или просматривать файлы, документы, программы и т. д.

Линейки прокрутки справа экрана или окна директив служат для перемещения содержимого открытого окна.



а



б

Рис. 5.14. Рубрики главного меню:
а — работа с таблицами (выбрана команда перехода к последней записи — `go bottom`); *б* — выбор режима просмотра (переключение на режим `edit` — полноэкранный вывод из режима `browse` — построчный вывод)

Статусная строка, расположенная внизу экрана, содержит информацию об открытом файле (рис. 5.13, *3*).

Окна. Каждое окно состоит из строки заголовка (с кнопкой вызова управляющего меню, кнопками распахивания/восстановления окна), рамки окна, линеек прокрутки. Не все окна

имеют полный набор перечисленных элементов. Линейка прокрутки показывает, какой фрагмент файла изображен на экране. С помощью мыши можно захватить бегунок прокрутки и отбуксировать его в требуемую позицию. Концевые манипуляторы линеек прокрутки — кнопки листания (это стрелки) в соответствующем направлении. Окна можно увеличивать или уменьшать с помощью директивы `Size` (размер). Захватив строку заголовка мышью, можно отбуксировать (Drag & Drop) окно в нужную позицию.

Часто встречаются ситуации, когда работа выполняется с несколькими файлами, расположенными в соответствующих окнах. Если нет необходимости видеть на экране все окна, их можно уменьшить до пиктограммы. Двойной щелчок мышью разворачивает окно до нормального состояния.

Диалоговые окна. Когда функцию невозможно выразить простой директивой, FoxPro вступает в диалог с пользователем. Например, надо сохранить файл. Программа должна знать, на каком накопителе, в каком каталоге, под каким именем его сохранить.

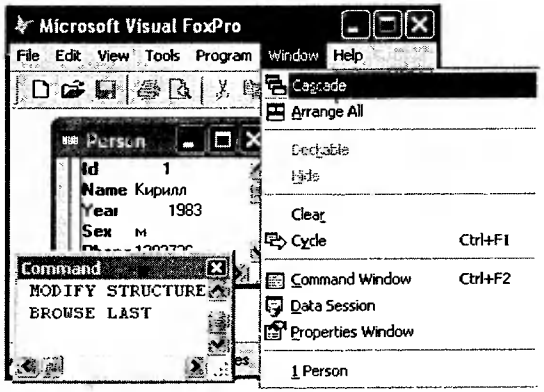
Диалог (диалоговое окно) — разновидность окна, для которого не предусмотрены средства манипулирования его размерами (они и не нужны). Диалоговое окно можно перемещать по экрану командой `Move` (перенести) или же, «захватив» заголовок диалогового окна мышью, отбуксировать его при нажатой кнопке мыши (рис. 5.15, *б*).

Внутри диалогового окна находятся различные интерфейсные элементы — командные кнопки, списки, селекторные кнопки, текстовые поля и т. д., которые расположены по тематическим группам, имеющим рамку или заголовок.

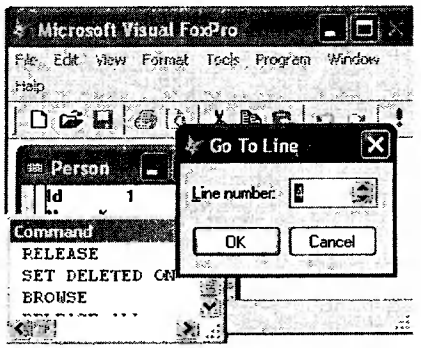
Во многих диалоговых окнах имеются также другие кнопки, которые открывают дополнительные (подчиненные) диалоговые кнопки. В наименованиях таких кнопок присутствует многоточие. Если командные кнопки не содержат многоточия (`New` или `Open`), то они выполняют конкретное действие, которое обозначено в названии кнопок.

Селекторные кнопки (radio button) используются в том случае, когда некоторый параметр может иметь одно значение из нескольких возможных.

Поля ввода (текстовые поля) используются тогда, когда пользователь должен сообщить системе обычный текст. Например, программа должна знать, куда отправить созданный файл. Это имя набирается с клавиатуры. Текстовые поля можно редактировать.



а



б

Рис. 5.15. Рубрики главного меню:
а — меню `Windows` (открыть — скрыть окна); *б* — диалоговое окно меню `Edit` (задается переход к 3-й строке диалогового сеанса с целью редактирования и повторного запуска команды)

Окно команд (`Command`) автоматически открывается при включении FoxPro (см. рис. 5.13, *5*). В этом окне пользователь может прямо вводить команды, т. е. пользоваться командным интерфейсом. Щелчком мыши можно выключить окно директив (левый верхний угол). С помощью меню Window (рис. 5.15, *а*) также можно выключать директиву `Hide` (подавить).

Создание и редактирование БД

Определение структуры файла. Для задания структуры файла данных используется команда **Create** — создать. При вводе команды система запрашивает имя создаваемого файла, которое будет использоваться для обращения к содержимому файла, и должно иметь длину не более десяти символов.

После задания имени файла на экране появляется диалоговое окно для определения структуры записей файла (рис. 5.16).

Определение структуры файла осуществляется заполнением приведенных полей. Каждому полю создаваемого файла соответствует одна строка структуры на экране. Структура файла набирается латинским шрифтом. Тип поля задается листанием допустимых в FoxPro полей. Ими могут быть: Character, Numeric, Date, Memo, Logical и др.

Имя поля в FoxPro набирается на латинице и должно начинаться с буквы, а остальные символы могут быть буквами, цифрами или знаками подчеркивания.

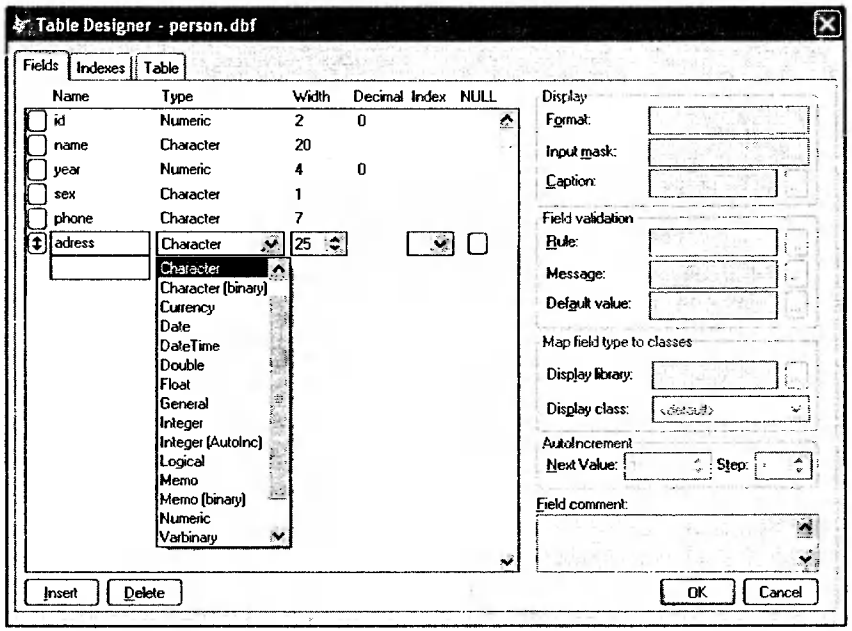


Рис. 5.16. Экран создания нового файла данных

После заполнения одной строки (создания одного поля файла) нажатием клавиши <Enter> переходят к заданию второго поля файла (второй строки экранной таблицы).

Для контроля соответствия созданного файла проекту пользователя может быть использована команда **Display Structure** (рис. 5.17). Здесь на рабочем поле экрана помещается отчет о структуре текущего файла данных.

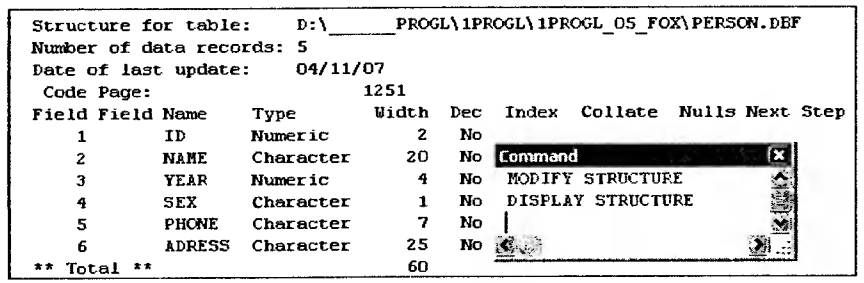


Рис. 5.17. Просмотр структуры файла командой **Display Structure**

Вставка нового поля. Для того чтобы вставить новое поле файла, следует нажать в области диалогового окна задания структуры файла (см. рис. 5.16) командную кнопку <Insert> (вставить).

Удаление поля. Чтобы удалить поле из файла, следует пометить поле и нажать командную кнопку <Delete> (удалить) того же диалогового окна (см. рис. 5.16).

Структура файла подготовлена. Переходим к заполнению файла, для этого следует нажать дважды <Enter> и подтвердить, что вы собираетесь далее заполнять файл <Y>. (Если вы не хотите вводить данные сразу, то нажмите <N>. Ввод информации в созданную структуру файла может быть осуществлен позже командой **Append**).

Создание и просмотр полей типа Memo. Предположим, необходимо ввести в структуру файла новое имя поля PRIM с типом поля Memo. Необходимо подвести курсор на поле Memo и нажать <Ctrl+PgDn>. В результате появляется экран текстового редактора. Ввести текст и после нажатия <Ctrl+PgUp> текст будет записан.

Просмотреть поля типа Memo:

```
List fio, prim
```

Изменение структуры файла

```
Use <Имя файла>
Modify Structure
```

Вставить новое поле между любыми имеющимися, воспользовавшись редактором.

Переименование файла

```
Rename <Старое имя> to <Новое имя>
```

Сортировка (**Sort**) — результат команды помещается не в исходный файл, а в результирующий, который при этом создается. Сортировать можно по убыванию или по возрастанию. При сортировке записи упорядочиваются в соответствии со значениями конкретного поля, а затем копируются в этой измененной последовательности. Итак, в процессе сортировки создается новый файл со своим именем (см. также рис. 5.2):

```
Sort On <Имя поля> To <Имя нового файла>
```

Индексирование (**Index**) служит той же цели, что и сортировка, т. е. упорядочению файла данных. При индексировании тоже создается новый (индексный) файл, но он не содержит всей информации отсортированного файла. Индексный файл включает только перечень номеров записей в той последовательности, какую они имели бы, будучи отсортированными.

Индексируемый файл по ФИО:

```
001 Иванов
002 Сидоров
003 Петров
```

Результат:

```
001 Иванов
003 Петров
002 Сидоров
```

Индексирование позволяет для одного файла данных хранить несколько индексных файлов.

```
Index On <Имя поля> To <Имя индексного файла>
```

В то время как команда **Sort** осуществляет физическую сортировку записей, команда **Index** управляет логической

сортировкой (оставаясь в исходной физической последовательности, записи файла будут обрабатываться в той логической последовательности упорядочения, которую задает открытый индексный файл).

Удаление дублирующихся записей. В файле могут быть дублирующиеся записи. Для их удаления можно использовать команду **Set Unique on**, выполняемую перед индексированием соответствующего файла:

```
Set Unique on
Index On <Имя ключевого атрибута>
To <Имя индексного файла>
Copy To <Новый файл>
```

При этом в <Новый файл> будут скопированы только записи, содержащие уникальные значения <Ключевого атрибута>.

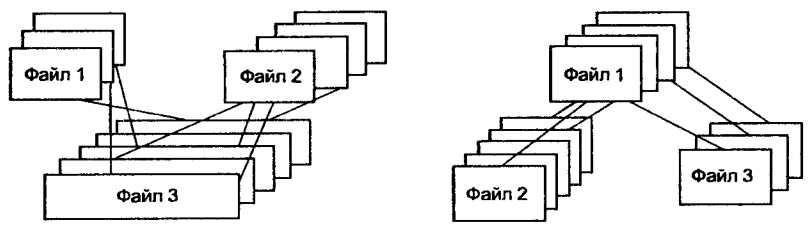
Переход к другому открытому файлу (рабочей области)

```
Select {<Номер рабочей области> | <Псевдоним файла>}
```

Команда **Select** обеспечивает выбор одной из десяти доступных рабочих областей для открытия файла данных или выбор рабочей области, в которой уже открыт файл БД. Если в рабочей области уже открыт файл, то выбрать рабочую область можно и по ее псевдониму К, где $К \le 10$:

```
Select К
Use <Имя файла>
```

Соединение двух таблиц БД — команда **Join**. Команда создает новый файл данных из двух существующих файлов (рис. 5.18, *а*). Команда **Join** устанавливает указатель на первую запись активного файла БД и выполняет последовательный просмотр второго файла.



а *б*

Рис. 5.18. Связывание таблиц:
а — команда **Join**; *б* — команда **Set Relation**

При нахождении записи, удовлетворяющей условию, ее содержимое присоединяется к текущей записи активного файла и помещается в выходной файл. Команда **Join** требует предварительной сортировки или индексирования обоих файлов. Синтаксис:

```
Join With <Псевдоним файла> To <Файл> For <Условие>
[Fields <Список полей>][For <Условие>]
```

Команда **Join** создает новый файл данных путем указанных записей и полей из текущего файла и файла данных, обозначенного как <Псевдоним файла>. Комбинированная таблица сохраняется в <Файл>. Вы можете ограничить выбор записей из активного файла данных, определив условие **For**. Если вы не включите <Список полей>, то будут скопированы все поля из обоих файлов. Определить поля из неактивного файла данных можно, используя конструкции <Файл>-><Имя поля> или <Файл>.<Имя поля>:

```
Select 1 Use file_1
Select 2 Use file_2
Select 1
Join With 2 Into file_3 For file1.field2=file2.field2
```

В результирующую таблицу попадут такие записи file_1 и file_2, которые имеют одинаковые значения поля field2.

Установление связи файлов данных. В то время как команда **Join** устанавливает физическую связь данных (создавая новый файл), команда **Set Relation** устанавливает логическую связь данных, не изменяя физического содержания таблиц, а синхронно извлекая из файлов данных связанные записи. При перемещении указателя текущей записи по основному файлу автоматически перемещаются указатели на активные записи в зависимых (их может быть несколько) файлах (рис. 5.18, *б*). Связывание файлов является необходимой операцией при работе с большими БД, включающими более двух файлов, поскольку оно обеспечивает работу SQL, реализующего большинство операций манипулирования данными. Синтаксис команды:

```
Set Relation To [<Выражение-1> Into
<Номер рабочей области-1> |
<Псевдоним файла-1> [,<Выражение-2> Into
<Номер рабочей области-2> |
<Псевдоним файла-2>...][In <Номер рабочей области> |
<Псевдоним файла>][Additive]]
```


Параметры:

- <Выражение-1> — задает выражение, которое устанавливает отношение между основной и зависимой таблицами. В качестве него обычно используется выражение управляющего индекса зависимой таблицы;
- **Into** <Номер рабочей области-1> | <Псевдоним файла-1> — задает для зависимой таблицы номер рабочей области (<Номер рабочей области-1>) или псевдоним (<Псевдоним файла-1>);
- <Выражение-2> **Into** <Номер рабочей области-2> | <Псевдоним файла-2> ... и т. д. С помощью одной команды **Set Relation** можно построить несколько отношений между одной основной и различными зависимыми таблицами;
- **In** <Номер рабочей области> — задает рабочую область основной таблицы. **In** <Псевдоним файла> задает псевдоним основной таблицы. Если аргументы опущены, основная таблица должна быть открыта в выбранной в данный момент рабочей области;
- **Additive** — сохраняет все уже существующие в текущей рабочей области отношения и создает отношение дополнительно. Если опустить ключевое слово **Additive**, все отношения в текущей рабочей области разрываются, после этого создается заданное отношение.

Рассмотрим пример:

```
Select 1 Use file_1
Select 2 Use file_2 Index field_1
Select 3 use file_3 Index field_2
Select 1
Set Relation To field_1 Into 1 field_2 Into 2 Additive
```

Здесь текущий файл (file_1) связывается с зависимыми файлами (file_2 и file_3), после этого записи зависимых файлов «берутся на буксир» и сопровождают записи текущего файла при перемещении. Предполагается, что зависимые файлы индексированы, причем индексные выражения соответствуют полям field_1 и field_2. Тогда в каждый момент времени из зависимых файлов активными будут записи, содержащие значения field_1 и field_2 текущей записи основного файла.

Выдав команду **Set Relation To** без аргументов, вы удалите все отношения, установленные в выбранной в данный момент рабочей области. Команду **Set Relation Off** можно использовать для удаления отдельных отношений типа родитель/потомок.

Создание и модификация форматов представления данных

Традиционно СУБД поддерживают две основные группы форматов данных — форматы экрана (ввод данных) и форматы отчетов (вывод данных). В первом случае речь идет о многооконной экранной форме, которая позволяет вводить, просматривать и корректировать сложный документ, а во втором — имеет место многоколонная ведомость, которая может снабжаться подсчетом итоговых, средних, максимальных и других групповых значений по полям данных.

В ранних версиях (Dbase, FoxBase) экранные формы создавались посредством группирования команд ввода-вывода в текстовый файл, который подключался при открытии файла данных для просмотра или редактирования. Вот пример подобного файла, а на рис. 5.19 приводится его интерпретация VFP:

```
* формат экрана - PRSN1T
    @ 1,15 Say [Студенческий контингент]
    @ 3,10 Say [Имя]
    @ 3,17 Get name
    @ 3,30 Say [Год рождения]
    @ 3,50 Get Year
    @ 5,10 Say [Пол]
    @ 5,15 Get sex
    @ 5,20 Say [Телефон]
    @ 5,30 Get phone
    @ 7,10 Say [Адрес]
    @ 7,20 Get adress
Read
* Конец формата PRSN1T
```

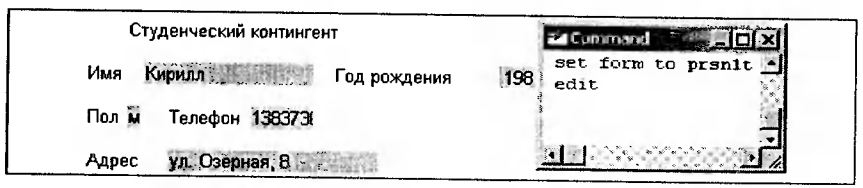


Рис. 5.19. Интерпретация текстового описания экранной формы

Здесь каждая строка (команда) задает положение на экране и формат ввода-вывода определенного данного, например `@ 1,15 Say [Студенческий контингент]` выдает текст в первой строке экрана, начиная с 15-й позиции, а `@ 3,17 Get name` образует окно для ввода/редактирования поля `name`, в 3-й строке, начиная с 17-й позиции. Длина окна соответствует по умолчанию длине поля `name` — 20 символов (см. рис. 5.19).

В системе VFP представлены два уровня средств управления представлением данных.

Это, во-первых, `Form Wizard` и `Report Wizard`, запускаемые из рубрики `Tools` главного меню и позволяющие начинающему пользователю выбрать из предлагаемого набора некоторые экранные формы или форматы отчета и построить простейшие управляющие файлы (рис. 5.20). Во-вторых, это средства `Form Designer` и `Report Designer`, которые вызываются из меню или командами **Create (Modify) Screen**, **Create (Modify) Report**.

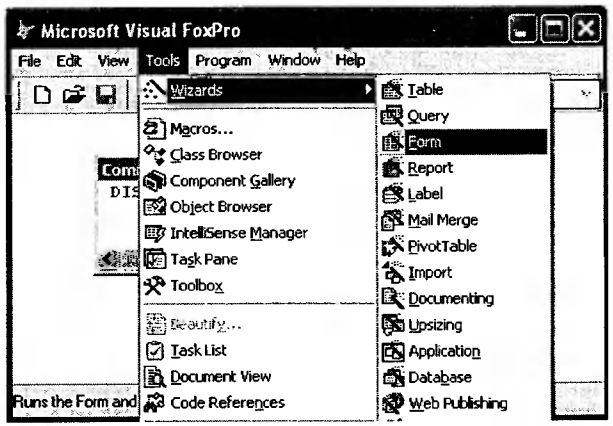
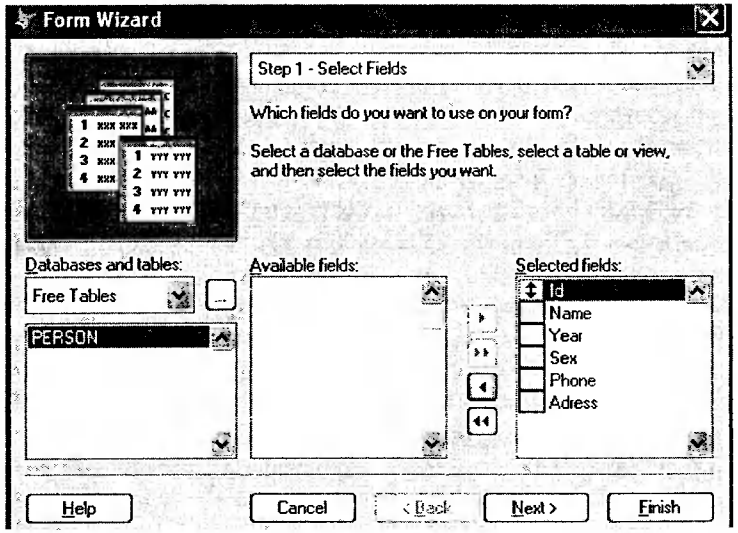


Рис. 5.20. Окно `Tools` — выбор генератора простейших экранных форм

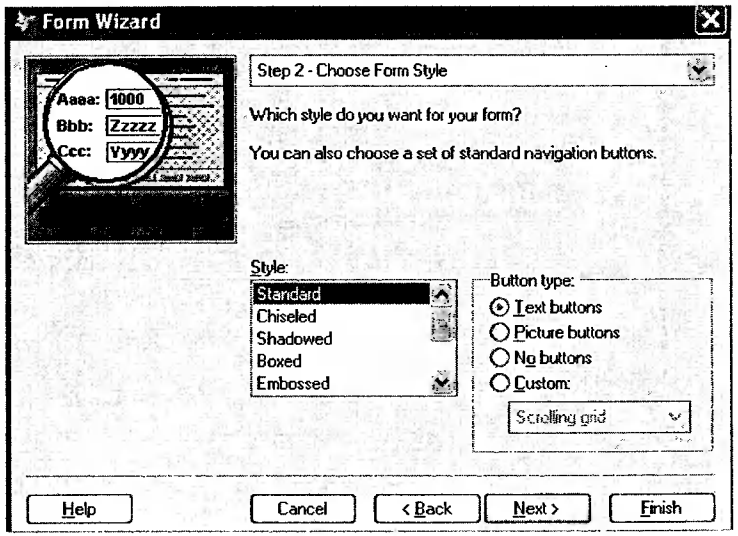
Генерация экранных форм

На рис. 5.21 представлены различные этапы генерации простых экранных форм:

- первый шаг — выбор таблиц и полей из таблиц для представления в экранной форме (рис. 5.21, *а*). Пользователю

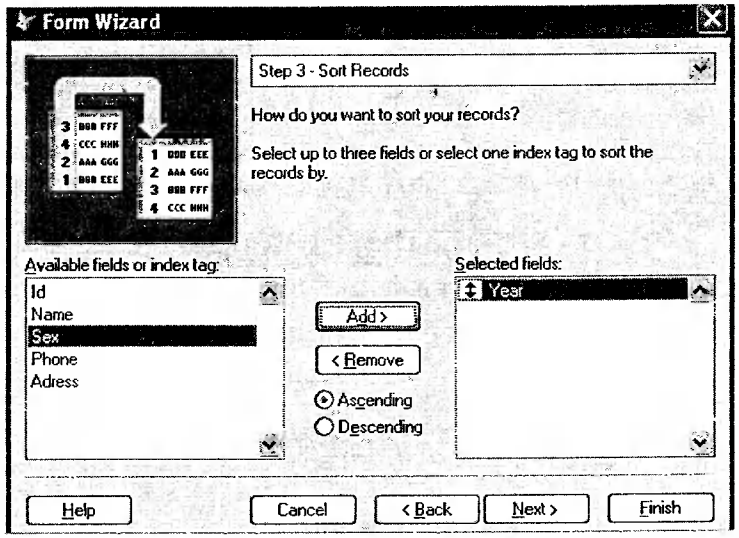


а

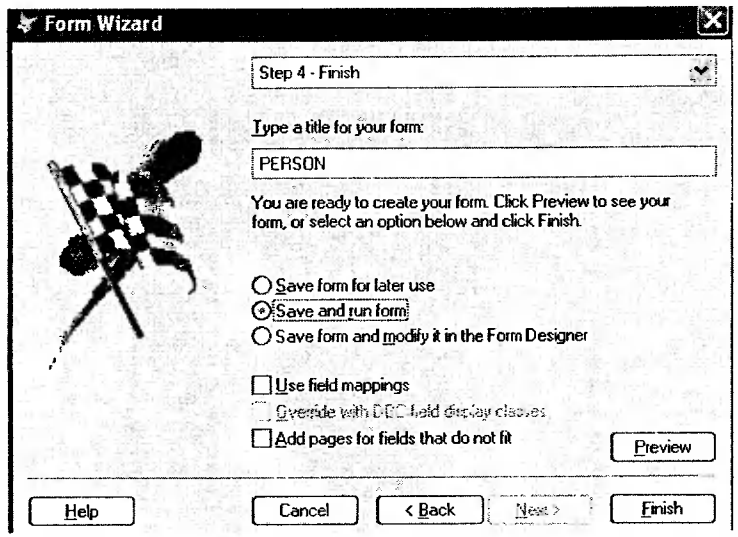


б

Рис. 5.21. Генератор экранов (начало):
а — выбор полей таблицы; *б* — выбор стиля формы; *в* — задание режима сортировки записей; *г* — завершение работы и сохранение формата (файл `person.scx`)



в

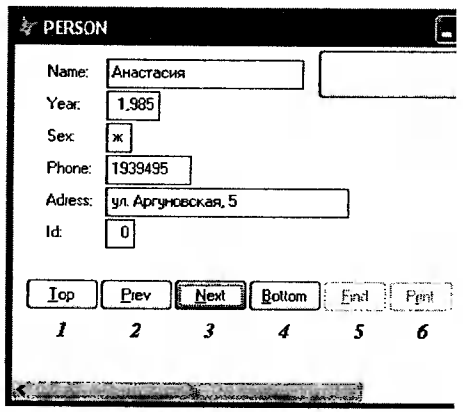


г

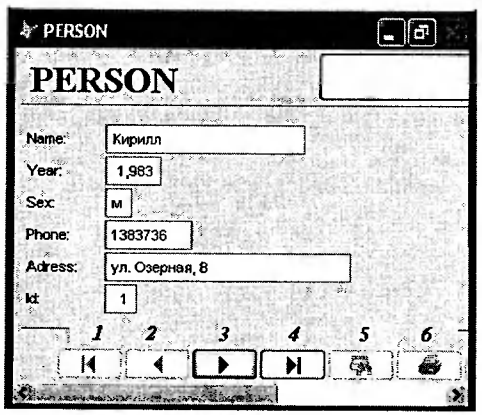
Рис. 5.21. Генератор экранов (окончание)

предлагается отобрать необходимые данные и занести их в список;

- второй шаг — выбор стиля экрана (рис. 5.21, *б*). Здесь можно выбрать оформление экранной формы из нескольких стандартных типов (в частности, различающихся шрифтами, разделителями, типом управляющих кнопок и пр.);



а



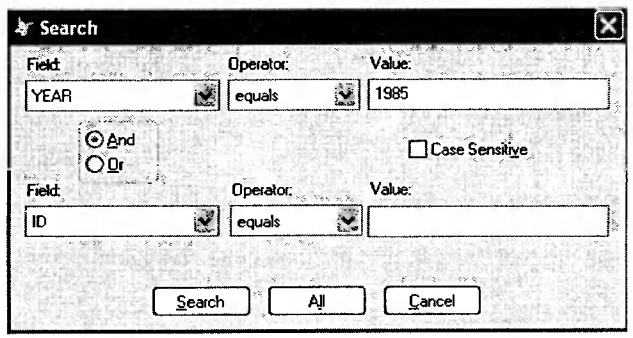
б

Рис. 5.22. Различные стили формата экрана:
а — `standard` (текстовые кнопки); *б* — `embossed` (графические кнопки); *1* — переход в начало файла; *2* — переход к предшествующей записи; *3* — переход к следующей записи; *4* — переход к концу файла; *5* — поиск по логическому критерию; *6* — распечатка

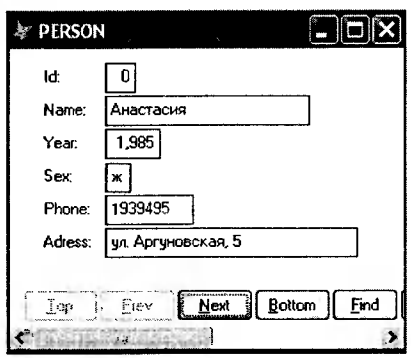
- третий шаг (рис. 5.21, *в*). Пользователь задает порядок выдачи данных из файла. Могут быть заданы до трех полей сортировки записей;
- четвертый шаг — завершение работы и сохранение форматного файла (здесь `prsn1f.scx`).

Пример управления выходным форматом приводится на рис. 5.22. При просмотре файла могут быть заданы критерии отбора релевантных записей (рис. 5.23).

Полученный на этом этапе экранный формат может быть затем откорректирован и развит средством `Form Designer` (рис. 5.24).



а



б

Рис. 5.23. Подключение режима отбора записей в формате экрана:
а — экран установки критерия отбора (кнопка `Find`); *б* — просмотр файла при включенном критерии отбора (`Year = 1985`)

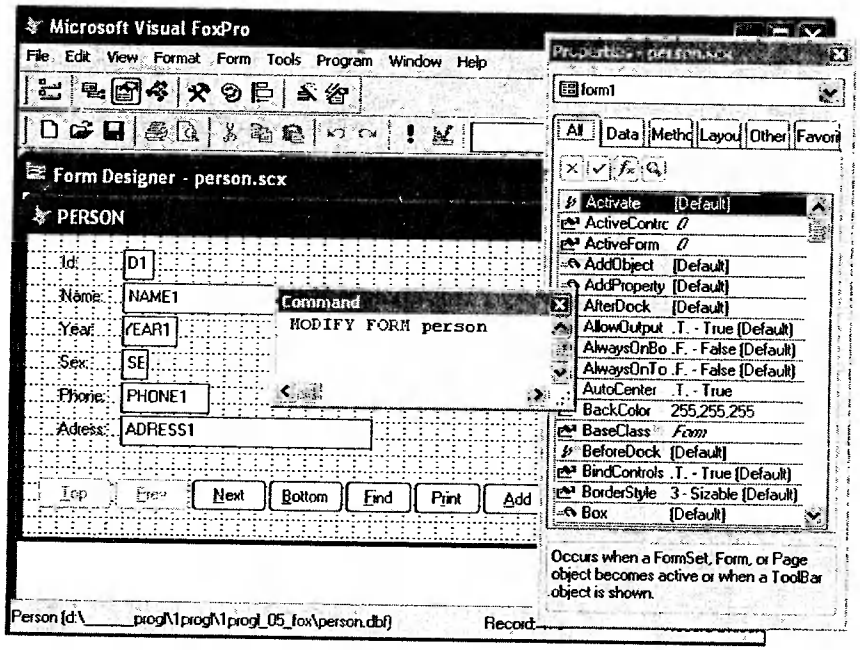
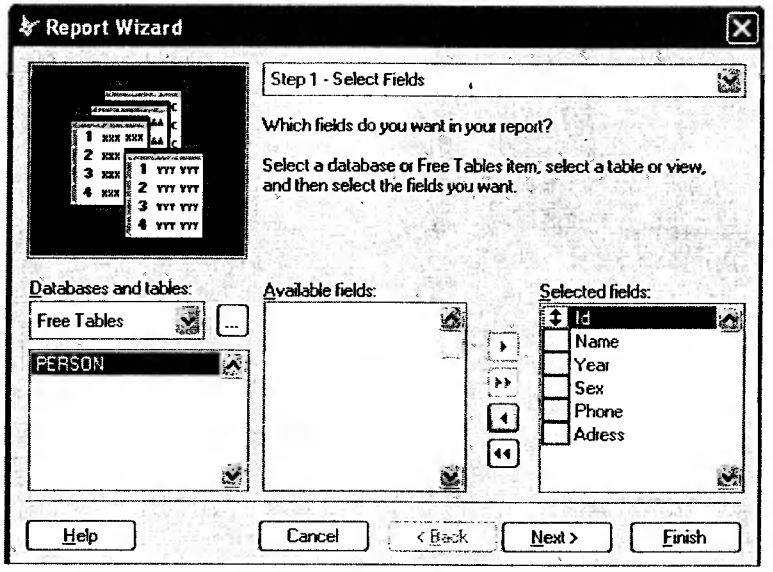


Рис. 5.24. Командой **Modify Format** вызван Form Designer для уточнения формата

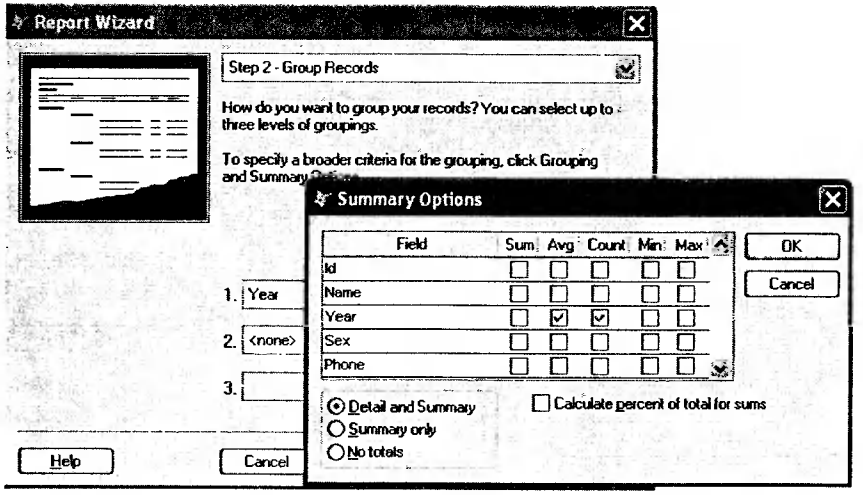
Генерация отчетов

На рис. 5.25 представлены различные этапы создания простых форматов отчетов:

- первый шаг — выбор таблиц и полей из таблиц для представления в экранной форме (рис. 5.25, *а*). Пользователю предлагается отобрать выводимые данные и занести их в список;
- второй шаг — выбор стиля экрана (рис. 5.25, *б*). Здесь можно выбрать оформление экранной формы из нескольких стандартных типов (в частности, различающихся шрифтами и оформления колонок и пр.);
- третий шаг (рис. 5.25, *в*). Пользователь задает порядок выдачи данных из файла (сортировка и группирование). Могут быть заданы до трех полей группирования записей. Здесь же можно потребовать вычисление агрегатных функций по каждой группе (сумма, среднее и пр.).

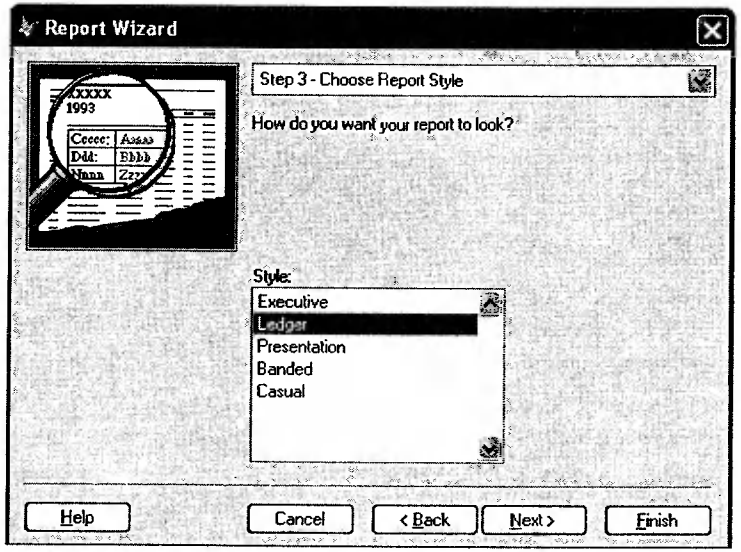


а



б

Рис. 5.25. Генератор отчетов (начало):
а — шаг выбора полей; *б* — задание группировки записей и подсчета групповых данных; *в* — выбор стиля отчета



в

Рис. 5.25. Генератор отчетов (окончание)

PERSON

04/12/07

Year	Id / Adress	Name	Sex	Phone
1983				
	1 ул. Озерная, 8	Кирилл	м	
1984				
	4 ул. Бородинская, 3	Анна	ж	3748374
1985				
	0 ул. Аргуновская, 5	Анастасия	ж	1939495
	3 ул. Летняя, 4	Ирина	ж	7947548

Command
REPORT FORM person

Рис. 5.26. Пример простейшего отчета (стиль — `Ledger`, записи сгруппированы по годам рождения)

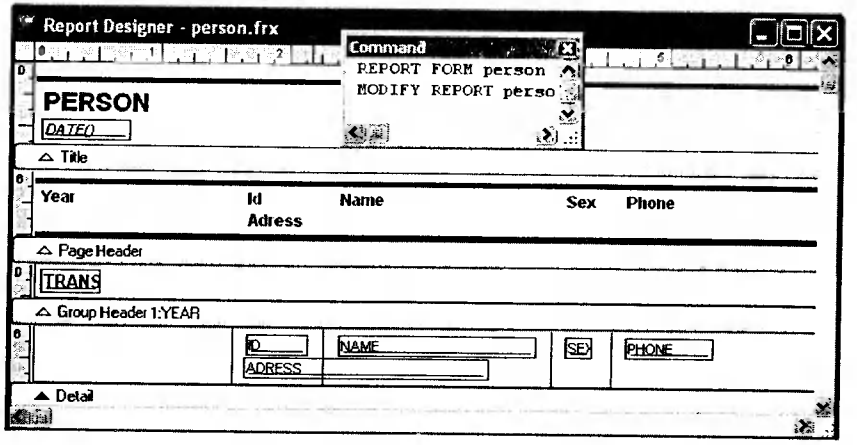


Рис. 5.27. Командой **`Modify Report`** вызван `Report Designer` для уточнения формата

Пример выходного формата отчета приводится на рис. 5.26. Затем полученный на этом этапе экранный формат может быть откорректирован и развит средством Report Designer (рис. 5.27).

Контрольные вопросы

1. Назовите основные типы файлов, образующих физическую структуру БД FoxPro.
2. Каково назначение индексного файла? Каким образом он может быть создан и активизирован?
3. Каким образом осуществляется отображение информации из БД?
4. Перечислите команды навигации в БД.
5. Каким образом можно перейти к началу файла? В конец файла?
6. Перечислите команды диалогового ввода-вывода информации.
7. Назовите основные конструкции, используемые для организации циклов.
8. В чем заключается различие операторов **`If`** и **`Do Case`**?
9. Перечислите основные понятия, характеризующие связь данных в БД.
10. Что такое экранные формы и каким образом они создаются и редактируются?
11. Что такое форматы отчетов и как они создаются и редактируются?

Заключение

В далеком прошлом люди считали на пальцах или делали насечки на ветках или костях. Древнейшим «счетным инструментом», который был представлен самой природой в распоряжение человека, была его собственная рука. На заре человеческой цивилизации были изобретены различные системы счисления, позволяющие осуществлять торговые сделки, рассчитывать астрономические циклы, проводить другие вычисления. Спустя несколько тысячелетий появились первые ручные вычислительные средства.

В наши дни сложнейшие вычислительные задачи, как и множество других операций, казалось бы, не связанных с числами, решаются именно с помощью компьютеров. Это еще один тип машин, построенный для того, чтобы увеличить эффективность и качество выполняемых работ и повысить производительность труда. Он по существу обладает единственной способностью — обрабатывать с высокой быстротой импульсы электрического поля. Истинное величие заключено в человеке, его гении, который нашел способ преобразовывать разнообразную информацию, поступающую из реального мира, в последовательность нулей и единиц двоичного кода, т. е. выразить ее на цифровом языке, идеально подходящем для электронных схем компьютера. Однако ни одна другая машина в истории не принесла в наш мир столь быстрых и глубоких изменений (посадка аппаратов на поверхность Луны и исследование планет Солнечной системы, управление медицинской аппаратурой в операционных, решение сложных экономических и управленческих задач, принятия управленческих решений, управление телефонными станциями и многое др.).

Каждому, знакомому с современными компьютерами, механические счетные машины и приборы покажутся, пожалуй, забавными и неуклюжими устройствами. Однако, ознакомившись

с историей развития счетных машин, можно поразиться изобретательности, хитроумию и настойчивости их создателей. Уместно вспомнить слова Б. Паскаля о том, что для создания «арифметической машины» ему потребовалось все ранее приобретенные знания по геометрии, физике и механике.

Закладка фундамента компьютерной революции происходила медленно и далеко не гладко. Отправной точкой этого процесса можно считать изобретение счетов (более 1500 лет назад). Они оказались очень эффективным инструментом и вскоре распространились по всему миру (в некоторых странах применяются и по сей день). В XVII в. европейские мыслители были увлечены идеей создания счетных устройств.

Работы, выполняемые в 40-х гг. XX в. по созданию вычислительных машин с программным управлением, были тесно связаны с появлением новой фундаментальной области науки — кибернетики, или науки об управлении и коммуникации. Судьба этой науки в нашей стране (в бывшем СССР) была трудной. Долгое время она считалась буржуазной «вульгарной лженаукой». Только в начале 50-х гг. прошлого века появились первые советские вычислительные машины. Несмотря на это роль отечественных ученых в области кибернетики и вычислительной техники неоценима.

Литература

1. *Алферова З. В.* Теория алгоритмов. М.: Статистика, 1973.

2. *Архангельский А. Я.* Object Pascal в Delphi. М.: БИНОМ, 2002.

3. *Вирт Н.* Алгоритмы и структуры данных. М.: Финансы и статистика, 1988.

4. *Голицына О. Л., Максимов Н. В.* Базы данных: учеб. пособие. М.: ФОРУМ: ИНФРА-М, 2003.

5. Информационные технологии: учеб. пособие / О. Л. Голицына, Н. В. Максимов, Т. Л. Партыка, И. И. Попов. М.: ФОРУМ: ИНФРА-М, 2005.

6. *Голицына О. Л., Партыка Т. Л., Попов И. И.* Программное обеспечение: учеб. пособие. М.: ФОРУМ: ИНФРА-М, 2006.

7. *Голицына О. Л., Партыка Т. Л., Попов И. И.* Системы управления базами данных: учеб. пособие. М.: ФОРУМ: ИНФРА-М, 2006.

8. *Голицына О. Л., Попов И. И.* Основы алгоритмизации и программирования: учеб. пособие. М.: ФОРУМ: ИНФРА-М, 2002.

9. *Григас Г.* Начала программирования. М.: Просвещение, 1987.

10. *Дарахвелидзе П. Г., Марков Е. П.* Программирование в Delphi 7. СПб.: БХВ-Петербург, 2003.

11. Основы информатики (учебное пособие для абитуриентов экономических вузов) / К. И. Курбаков, Т. Л. Партыка, И. И. Попов, В. П. Романов. М.: Экзамен, 2004.

12. *Михайлов В. Ю., Степанников В. М.* Современный Бейсик для IBM PC. Среда, язык, программирование. М.: Изд-во МАИ, 1993.

13. *Морозов В. П., Шураков В. В.* Основы алгоритмизации, алгоритмические языки и системы программирования: задачник. М.: Финансы и статистика, 1994.

14. *Орлов С. А.* Технологии разработки программного обеспечения: учебник. СПб.: Питер, 2002.

15. *Партыка Т. Л., Попов И. И.* Вычислительная техника: учеб. пособие.— М.: ФОРУМ: ИНФРА-М. 2006.

16. *Партыка Т. Л., Попов И. И.* Электронные вычислительные машины и системы: учеб. пособие. М.: ФОРУМ: ИНФРА-М, 2007.

17. *Партыка Т. Л., Попов И. И.* Операционные системы, среды и оболочки. М.: ФОРУМ: ИНФРА-М, 2003.

18. *Романенко А. Г., Самойлюк О. Ф.* Проектирование и эксплуатация банков данных в программных средах FoxBase и Foxpro: лабораторный практикум. М.: РГГУ, 1997.

19. Турбо Паскаль 7.0. К.: Торгово-издательское бюро BHV, 1996: 448 с., ил.

20. *Федоров А.* Создание Windows-приложений в среде Delphi. М.: Компьютер-пресс, 1995.

21. *Фридман А. Л.* Основы объектно-ориентированной разработки программных систем. М.: Финансы и статистика, 2000.

Приложение 1

ГЛОССАРИЙ ТЕРМИНОВ (АНГЛИЙСКИЙ ЯЗЫК)

Algol (ALGOrthmic Language) — алгоритмический язык, разработанный для решения научных и инженерных вычислительных задач, в котором впервые (1960 г.) были определены и стандартизованы основные понятия и объекты языков программирования.

ANSI (American National Standards Institute) — неправительственная организация, создающая и публикующая стандарты для добровольного использования в США.

API (Applications Programmer's Interface) — интерфейс прикладного программирования — спецификация набора функций, которую должны выдержать разработчики программного обеспечения для совместимости своих программ с соответствующими операционными системами.

ASCII (American Standard Code for Information Interchange) — разработанный ANSI (American National Standards Institute) стандарт представления символьной информации в ЭВМ. Символы ASCII содержат 128 символов с кодами от 0 до 127 и включают цифры, знаки пунктуации, буквы и управляющие коды, такие, как конец строки или перевод страницы.

Basic — «многоцелевой язык символических команд для начинающих». Самый простой язык высокого уровня для освоения принципов алгоритмизации и программирования. Для многих мини- и микроЭВМ предназначался в качестве единственного языка высокого уровня.

Bit — двоичная единица, базовая мера для измерения количества данных. Имеет значение «1» или « 0». Для размещения 1 байта требуется 8 бит.

Byte — восемь бит, рассматриваемые как единое целое и представляющие, например, символ кода ASCII.

CASE-средства (технологии) — программные средства, поддерживающие процессы создания и сопровождения ИС, включая

анализ и формулировку требований, проектирование прикладного ПО (приложений) и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы.

CCIA (Computer and Communications Industry Association) — ассоциация фирм-производителей компьютеров и средств коммуникации, представляющая их интересы в зарубежной и национальной торговле, а также разрабатывающая соответствующие стандарты.

Cobol (COmmon Business Oriented Language) — алгоритмический язык высокого уровня, ориентированный на обработку данных при решении экономических и управленческих задач.

CP-866 — стандарт IBM для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для русской кириллицы.

CPU (Central Processing Unit) — центральный процессор.

Flops (FLoating point operation per second) — быстродействие в количестве операций с плавающей точкой, выполняемых в секунду. Производные единицы — Mflops (10^6 оп/с), Gflops (10^9 оп/с), Tflops (10^{12} оп/с). Употребляются также транслитерации: флопс, Мфлопс, Гфлопс, Тфлопс.

Fortran — первый из известных машиннонезависимых языков программирования, ориентированный на научные и инженерные расчеты.

FoxBase — система программирования с развитыми элементами СУБД для персональных ЭВМ, являющаяся развитием систем dBase, FoxBase, Clipper. Язык программирования, объединяющий системы dBase, FoxBase, FoxPro, Clipper, включает как операторы алгоритмических (процедурных) языков, так и команды языков запросов (непроцедурных).

GB (GigaByte) — гигабайт, единица измерения, содержащая 1000 Мбайт. В качестве альтернативной IEC предложила в 1998 г. GiB (GibiByte) — 1024 MiB (MibiByte).

IDEF (Integrated DEFinition) — семейство стандартов, определяющее взаимную совокупность методик и моделей концептуального проектирования. Разработано в США по программе Integrated Computer-Aided Manufacturing.

IEC (International Electrotechnical Commission) — международная организация по стандартизации в области электротехники. В частности, в 1998 г. IEC предприняла попытку устранить разночтения в трактовке 1 Мбайта как 1000 байт (десятичный мегабайт) и 1024 байт (бинарный мегабайт), предложив обозначения 1 MegaByte и 1 MibiByte, соответственно. Аналогично были определены 1 KibiByte = 1024 байт, 1 GibiByte = 1024 MibiByte и 1 TibiByte = 1024 GibiByte.

IEEE (Institute of Electrical and Electronics Engineers) — общественная организация, которая объединяет специалистов, ученых, студентов и др. лиц, заинтересованных в электронике и смежных областях. Известна более как разработчик и популяризатор стандартов в вычислительной технике и связи, например — IEEE 802 — стандарт для локальных сетей.

IP (Instruction Pointer) — указатель команды (регистр в МП, хранящий адрес выполняемой команды).

ISO (International Standards Organization) — Международная организация по стандартизации (МОС) — международный орган, ответственный за создание и контроль деятельности различных комитетов по стандартизации и рабочих групп, работающих над стандартами обработки данных (например, сжатия изображений и пр.).

KB (KiloByte, Кбайт) — килобайт, мера количества информации в 1000 байт. Альтернативой является предложенная IEC единица 1 KiB (KibiByte) = 2^{10} = 1024 байт.

Kbit (Kilobit, Кбит) — килобит, единица измерения объема данных, равная 1000 бит, часто употребляемая для измерения скорости передачи данных (кбит/с).

Latin-1 — международный стандарт (ISO-8859-1) для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для латиницы.

Latin-C — международный стандарт (ISO-8859) для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для кириллицы.

MB, MegaByte, Мбайт (мегабайт) — единица измерения количества информации, равная 1000 Кбайт. Альтернативой является предложенная IEC единица 1 MiB (MibiByte) = 1024 KibiByte.

Microcode (микрокод, микрокоманда) — система команд нижнего (аппаратурного) уровня, которая напрямую управляет микропроцессором. Отдельная машинная команда (инструкция) обычно переводится на несколько микрокоманд. В современных ПК микрокоманды встроены в ПЗУ и не могут модифицироваться.

Object Pascal — объектно-ориентированное расширение языка программирования Pascal. Базовый язык программирования в интегрированной среде разработки Delphi.

Pascal — универсальный язык, являющийся развитием ЯП Algol, ориентированный на широкий класс задач обработки данных и организацию вычислений (научные, инженерные, экономические, управленческие и пр.).

Rational Rose — средство моделирования объектно-ориентированных информационных систем, базирующееся на языке моделирования UML.

SQL — стандартизованный язык запросов к реляционной (табличной) базе данных, обеспечивающий реляционно полный набор операций над данными; входит в состав окружения dBase-4, Oracle, FoxPro и других систем.

TB (Terabyte, терабайт) — единица измерения емкости памяти, содержащая 1000 Гбайт. В качестве альтернативной IEC предложила в 1998 г. 1 TiB (TibiByte) = 1024 GiB (GibiByte).

Volume — том, или логический носитель данных, содержащий множество файлов. В случае НЖМД том обычно размещается в разделе диска, форматируется для поддержки файловой системы (FAT, NTFS или др.) и ему назначается идентификатор (буква — `C:`, `D:` и пр.). Как правило, на одном физическом диске размещают несколько логических томов, однако том может занимать и несколько дисков. В соответствии со стандартом ISO 9660 «том» — это единичный диск CD-ROM.

WIMPD (Windows, Menu, Pointing Device) — группа пользовательских интерфейсов, использующих понятия окна, пиктограммы, меню и указывающее устройство (например, мышь).

WINDOWS — семейство оболочек MS DOS и операционных систем для IBM PC-совместимых ПЭВМ, реализующих дружественный пользовательский интерфейс.

Приложение 2

ГЛОССАРИЙ ТЕРМИНОВ (РУССКИЙ ЯЗЫК)

Агрегат данных — именованная совокупность элементов данных, представленных простой (векторной) или иерархической (группы или повторяющиеся группы) структурой. Примеры — массивы, записи, комплексные числа и пр.

Администратор системы (system administrator) — лицо (группа, подразделение) ответственное за поддержание системы (операционной, информационной, вычислительной) в работоспособном состоянии.

Адресация — способ размещения и выборки данных в памяти.

Алгебра логики — раздел математики, изучающий высказывания, рассматриваемые со стороны их логических значений (истинности или ложности) и логических операций над ними.

Алгоритм — понятное и точное предписание (указание) исполнителю совершить определенную последовательность действий, направленных на достижение указанной цели или решение поставленной задачи (приводящую от исходных данных к искомому результату).

Алфавит — фиксированный для данного языка набор основных символов, т. е. «букв алфавита», из которых должен состоять любой текст на этом языке. Никакие другие символы в тексте не допускаются.

Арифметико-логическое устройство (АЛУ) — часть процессора, которая производит выполнение операций, предусмотренных данным компьютером.

Арифметические операции — четыре действия арифметики (+, –, *, /) и операция получения остатка от деления (%) образуют группу арифметических операций. Их выполнение не имеет каких-либо особенностей, кроме как преобразование типов переменных при их несовпадении. Если в одном выражении встречаются переменные разных типов, как правило, осуществляется преобразование (приведение) типов.

Архитектура фон Неймана — архитектура компьютера, имеющего одно арифметико-логическое устройство, через которое проходит поток данных, и одно устройство управления, через которое проходит поток команд.

Архитектура ЭВМ — общее описание структуры и функции ЭВМ на уровне, достаточном для понимания принципов работы и системы команд ЭВМ. Архитектура не включает в себя описание деталей технического и физического устройства компьютера. Основные компоненты архитектуры ЭВМ: процессор, внутренняя (основная) память, внешняя память, устройства ввода, устройства вывода.

Атрибут — поле данных, содержащее информацию об объекте.

База данных (БД) — именованная совокупность взаимосвязанных данных, отображающая состояние объектов и их отношений в некоторой предметной области, используемых несколькими пользователями и хранящимися с минимальной избыточностью.

Байт — машинное слово минимальной размерности, адресуемое в процессе обработки данных. Размерность байта — 8 бит — принята не только для представления данных в большинстве компьютеров, но и в качестве стандарта для хранения данных на внешних носителях, для передачи данных по каналам связи, для представления текстовой информации. Кроме того, размерность всех форм представления данных устанавливается кратной байту. При этом машинное слово считается разбитым на байты, которые нумеруются, начиная с младших разрядов.

Библиотека (library) — набор функций, в том числе из стандартных библиотек, предопределенных переменных и констант, которые могут быть использованы в программе и хранятся в откомпилированном виде.

Бит (от *англ.* **BI**nary digi**T** — двоичная единица) — единица измерения количества информации, равная количеству информации, содержашемуся в опыте, имеющем два равновероятных исхода. Это наименьшая единица информации в цифровом компьютере, принимающая значения «0» или «1».

Блок — составной оператор, включающий описания переменных в начале. Это собственные переменные блока, действие которых

не распространяется за его пределы, а время существования совпадает с временем его выполнения.

Блок-схема — 1. Графическое представление алгоритма, повышающее его наглядность. Составление блок-схем особенно полезно начинающим программистам; 2. Графическое представление состава технических средств, или структуры системы.

Булевская (булева) алгебра — раздел математической логики, изучающий высказывания и операции над ними. Частный случай алгебры логики. Под высказываниями понимается любое утверждение, которое бывает либо истинным, либо ложным. Над высказываниями возможны операции: И (конъюнкция &, ∧); ИЛИ (дизъюнкция, ∨); «если ..., то» (импликация, →); двусторонняя импликация (эквивалентность, ~); НЕ (отрицание, ¬). Введено понятие функций, которые могут задаваться таблицами (таблицы истинности). Логические операции подчиняются законам: коммутативности, ассоциативности, поглощения, дистрибутивности, противоречия и исключенного третьего.

Валидация — автоматическая проверка распознанных данных на соответствие заданным правилам. Например, проверка на попадание численных данных в определенный интервал, проверка совпадение сумм, указанных цифрами и прописью, проверка на соответствие формату или заданному значению.

Величина аналоговая — величина, у которой значения изменяются непрерывно и ее конкретное значение зависит только от точности прибора, производящего измерение. Например, температура воздуха.

Величина дискретная — величина, значения которой изменяются скачкообразно. Например, величина, характеризующая наличие или отсутствие тока в электрической цепи, является дискретной и может принимать значения «да» или «нет» («0» или «1»).

Вещественное число — тип данных, содержащий числа, записанные с десятичной точкой и (или) с десятичным порядком.

Внешняя ссылка — обращение к переменной или вызов функции во внутреннем представлении модуля, которые определены в другом модуле и отсутствуют в текущем.

Восьмеричная система счисления — позиционная система счисления с основанием 8. Для записи чисел используются цифры 0, 1, 2, 3, 4, 5, 6, 7. Например, 123 в восьмеричной системе равно числу $1 \cdot 8^2 + 2 \cdot 8^1 + 3 \cdot 8^0 = 64 + 16 + 3 = 83$ в десятичной системе.

Время (`time`) — тип данных, предназначенный для отображения моментов событий, предполагает наличие встроенных или эмулируемых команд специальной арифметики.

Время выполнения (run time) — период, во время которого происходит выполнение программы.

Время компиляции (compiler time) — период, во время которого происходит компиляция программы. Во время компиляции обнаруживаются синтаксические ошибки.

Вызов подпрограммы. Обращение к подпрограмме реализуется при поступлении в процессор специальной команды `Call` (в некоторых процессорах эта команда имеет мнемоническое обозначение `Jsr — Jump to SubRoutine`), которая указывает адрес первой команды вызываемой подпрограммы.

Вызов функции (call interface) — выполнение ее тела с заданными значениями формальных параметров.

Выражение — множество взаимосвязанных операций над переменными и константами и скобок «(» и «)», в котором результат одной операции является операндом другой.

Высказывание — понятие математической логики, определяемое как повествовательное предложение, которое может быть истинным или ложным, но не может быть истинным и ложным одновременно. Над высказываниями возможно производить логические операции.

Генерация кода — преобразование элементарных действий, полученных в результате лексического, синтаксического и семантического анализа программы, в некоторое внутреннее представление. Это могут быть коды команд, адреса и содержимое памяти данных, либо текст программы на языке Ассемблера, либо стандартизованный промежуточный код. В процессе генерации кода производится его оптимизация.

Гигабайт (Гбайт) — гигабайт, единица измерения, содержащая 1000 Мбайт. В качестве альтернативной IEC предложила в 1998 г. 1 GiB (GibiByte) = 1024 MiB (MibiByte).

Главная, (внутренняя, оперативная) память компьютера — упорядоченная последовательность байтов или машинных слов (ячеек

памяти), проще говоря — массив. Номер байта или слова памяти, через который оно доступно как из команд компьютера, так и во всех других случаях, называется адресом. Если в команде непосредственно содержится адрес памяти, то такой доступ этому слову памяти называется прямой адресацией.

Глобальные переменные — определены вне тела функции. Представляют собой общие данные программы, которыми могут пользоваться все функции, следующие по тексту кода программы.

Данные — информация, обработанная и представленная в формализованном виде для дальнейшей обработки.

Дата (`date`) — тип данных, предназначенный для обработки и отображения дат событий и другой аналогичной информации, предполагает наличие встроенных или эмулируемых команд специальной арифметики.

Двоичная система счисления — позиционная система счисления с основанием 2. Для записи чисел используются двоичные цифры 0 и 1. Например, 101101 в двоичной системе равно числу: $1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 45$ в десятичной системе. Является основной в вычислительной технике, так как приборы, имеющие два устойчивых состояния, проще, чем приборы с любым другим числом состояний. Используются производные системы счисления (степени 2) — восьмеричная и шестнадцатеричная.

Двойная точность (`double`) — числовое данное (с фиксированной или плавающей точкой), размещенное в двух машинных словах, требует наличия операций специальной арифметики.

Двойное слово — машинное слово двойной длины, используется для увеличения диапазона представления целых чисел. Двойные слова обрабатываются либо отдельными командами процессора, либо программно (эмуляция).

Диск — носитель данных в форме круглой пластины, на которую осуществляется запись разными способами. Часто под диском подразумеваются несколько дисков, объединенных в пакет. Устройство, которое записывает (читает) данные на/с диск/а, называется накопителем данных. Диски различаются по способу записи/чтения данных, возможности их замены, плотности записи. По способу записи/чтения диски делятся на магнитные, лазерные (оптические), магнитооптические.

Длинное поле (`long`) — данное, занимающее неопределенное или переменное количества машинных слов, обычно предназначенное для размещения текстовой или графической информации.

Дополнительный код — беззнаковая форма представления чисел со знаком. В двоичной системе счисления дополнение каждой цифры выглядит как инвертирование двоичного разряда, т. е. замена 0 на 1 и наоборот. Если же знак числа представляется старшим разрядом машинного слова, то получается простой способ представления отрицательного числа: взять абсолютное значение числа в двоичной системе; инвертировать все разряды, включая знаковый; добавить 1.

Заголовок функции. Описывает «интерфейс» функции. Содержит имя функции, по которому она известна далее в программе.

Задача — одна или несколько программ, связанных общим назначением, ресурсами.

Запись (record) — агрегат данных, составляющий элемент базы данных (файла), содержащий разнотипную информацию, описывающую некоторый объект (сущность, экземпляр и пр.).

Запись логическая — идентифицируемая (именованная) совокупность элементов или агрегатов данных, воспринимаемая прикладной программой как единое целое при обмене информацией с внешней памятью. Запись — это упорядоченная в соответствии с характером взаимосвязей совокупность полей (элементов) данных, размещаемых в памяти в соответствии с их типом.

Запись физическая — совокупность данных, которая может быть считана или записана как единое целое одной командой ввода-вывода.

Защита памяти осуществляется путем блокировки доступа к памяти других процессов, а также блокировки доступа к памяти ядра. Один из способов — вся память делится на страницы, и у каждой есть замок — 4-битовый признак, который можно установить только привилегированной командой.

Идентификатор — имя переменной, состоящее из больших и маленьких латинских букв, цифр и знака «_» (подчеркивание) и начинающееся с буквы.

Импорт (загрузка, download) — утилита (функция, команда) СУБД, служащая для чтения файлов операционной системы, которые

содержат данные из базы данных, представленные в некотором коммуникативном формате.

Имя файла — последовательность от одного до (`FilenameMax`) символов, использующаяся для именования обычных файлов, каталогов или специальных файлов. В имени файла допустимы любые символы кода ASCII за исключением управляющего кода 0 (ПУС) и символа «/». Не рекомендуется использовать в именах файлов символы, имеющие специальное значение для языков управления заданиями (типа «*», «?»).

Индекс (index) — таблица (файл), устанавливающая соответствие содержания записи (поля, документа) и физического адреса на устройстве (или в информационной сети), используемая для индексного поиска.

Инициализация — установка начальных значений во время трансляции.

Инкапсуляция) — свойство объектно-ориентированного программирования, состоящее в сокрытии информации об объекте и комбинировании в объекте данных и функций.

Интерпретатор — разновидность транслятора, осуществляет покомандную расшифровку программы и выполнение инструкций входного языка (в среде конкретной системы программирования).

Интерфейс (interface) — совокупность технических и программных средств визуализации информации и ввода данных, обеспечивающая интерактивное взаимодействия пользователя с системой.

Исключения (exception) — нештатные ситуации (ошибки), возникающие при работе процессора. При выявлении таких ошибок соответствующие блоки, контролирующие работу процессора, вырабатывают внутренние сигналы запроса, обеспечивающие вызов необходимой подпрограммы обслуживания.

Исполняемый модуль — модуль, содержащий готовую к выполнению программу; может быть двух видов: точный образ памяти программы с привязкой к абсолютным адресам (в MS-DOS — формат файла *.COM); перемещаемый исполняемый формат.

Исходный код программы — код, написанный на языке программирования. Может включать модули на ЯВУ и модули с подпрограммами на языке ассемблера.

Итерационный цикл — вид цикла, для которого число повторений операторов тела цикла заранее неизвестно. На каждом шаге вычислений происходит последовательное приближение и проверка условия достижения искомого результата. Выход из цикла осуществляется в случае выполнения заданного условия.

Каталог — специальный тип файла (иногда именуется директорией или папкой), содержащий информацию о файлах, которые могут адресоваться из данного каталога без указания полного имени (т. е. по имени файла).

Килобайт (Кбайт) — килобайт, единица измерения количества данных или объема памяти, равная $10^3 = 1000$ байт. Альтернативой является предложенная IEC бинарная единица 1 KiB (KibiByte) = $= 2^{10} = 1024$ байт.

Ключевая директива включает или отключает определенные директивой возможности компилятора.

Код ASCII (от *англ.* American Standard Code for Information Interchange — Американский стандартный код для обмена информацией) — стандарт кодирования символов латинского алфавита, цифр и вспомогательных символов или действий в виде однобайтового двоичного кода (1 байт = 8 бит). Первоначально стандарт определял только 128 символов, используя 7 битов (от 0 до 127). Использование всех восьми битов позволяет кодировать еще 128 символов. В этом случае говорят о расширенном ASCII-коде. Дополнительные символы могут быть любыми, им отводятся коды от 128 до 255. Русские символы кодируются именно в этой части ASCII-кода. Например, служебное действие «ввод» (клавиша <Enter>) имеет код 13, символ «1» имеет код 49, символ «W» — 87, символ «w» — 119, символы кириллицы «Б» и «б» — соответственно, 129 и 161 (при альтернативной кодировке). Другие обозначения — IA-5, ANSI X.34, ISO-7 (код ISO-7 отличается 10 кодовыми комбинациями, зарезервированными для национальных применений).

Код EBCDIC (Extended Binary Coded Decimal Interchange Code) — внутренний 8-битовый код хранения символьной информации в больших машинах (*mainframes*) фирмы IBM и ряда других.

Код Unicode — стандарт для представления символов с использованием 16-разрядных кодов (2 байта). Допускает 65 536 символов. Стандарт должен в перспективе заменить ASCII, так как удоб-

нее пользоваться одним кодом для разных языков, чем менять перекодировочные таблицы в ASCII-коде.

Код Бодо (Baudot code) — международный телеграфный код IA-1 (International Aphabet 1), предшественник IA-2 (M-2, MKKTT-2, CCITT-2).

Код MKKIT-2 (CCITT-2 code) — телеграфный код, предложенный Международным Консультативным комитетом по телефонии и телеграфии (MKKTT), бинарный 5-разрядный, трехрегистровый; он же IA-2 (International Alphabet 2).

Код Холлерита (Hollerith code) — код, используемый для представления информации на 80-колонных перфокартах, 12-разрядный, избыточный.

Кодирование (coding) — установление согласованного (узаконенного) соответствия между набором символов и сигналами или битовыми комбинациями, представляющими каждый символ для целей передачи, хранения или обработки данных.

Кодовая таблица (code page) — таблица, устанавливающая стандартизованное соответствие графических символов и бинарных кодов, определяемое применением (алфавит, программы, устройства ЭВМ).

Команда представляет собой многоразрядное двоичное число, которое состоит из двух частей (полей) — кода операции (КОП) и адресной части (АДЧ). Код операции КОП задает вид операции, выполняемой данной командой, а АДЧ определяет выбор операндов (способ адресации), над которыми производится заданная операция. В зависимости от типа микропроцессора команда может содержать различное число разрядов (байтов). Например, команды процессоров Pentium содержат от 1 до 15 байтов, а большинство процессоров с RISC-архитектурой используют фиксированный 4-байтный формат для любых команд.

Команды арифметических операций. Основными в этой группе являются команды сложения, вычитания, умножения и деления, которые имеют ряд вариантов. Команды сложения ADD и вычитания SUB выполняют соответствующие операции с содержимым двух регистров, регистра и ячейки памяти или с использованием непосредственного операнда. Команды ADC, SBB производят сложение и вычитание с учетом значения признака C, устанавливаемого при формировании переноса или заема в процессе выполнения предыдущей операции. Команда NEG изменяет знак операнда, переводя его в дополнительный код. Операции умножения и деления могут выполняться над числами со знаком (команды IMUL, IDIV) или без знака (команды MUL, DIV). Один из операндов всегда размещается в регистре, второй может находиться в регистре, ячейке памяти или быть непосредственным операндом. Результат операции располагается в регистре. При умножении (команды MUL, IMUL) получается результат удвоенной разрядности, для размещения которого используется два регистра. При делении (команды DIV, IDIV) в качестве делимого используется операнд удвоенной разрядности, размещаемый в двух регистрах, а в качестве результата в два регистра записывается частное и остаток.

Команды битовых операций. Эти команды производят установку значения признака C в регистре состояний в соответствии со значением тестируемого бита b_n в адресуемом операнде. В некоторых микропроцессорах по результату тестирования бита производится установка признака *Z*. Номер тестируемого бита *n* задается либо содержимым указанного в команде регистра, либо непосредственным операндом.

Команды логических операций . Практически все процессоры производят логические операции И, ИЛИ, Исключающее ИЛИ, которые выполняются над одноименными разрядами операндов с помощью команд AND, OR, XOR. Операции выполняются над содержимым двух регистров, регистра и ячейки памяти или с использованием непосредственного операнда. Команда NOT инвертирует значение каждого разряда операнда.

Команды организации программных циклов осуществляют условный переход в зависимости от значения содержимого заданного регистра, который используется как счетчик циклов. Например, в процессорах Pentium для организации циклов используется команда LOOP и регистр ECX. Команда LOOP уменьшает содержимое ECX на 1 (декремент) и проверяет полученное значение. Если содержимое ECX ≠ 0, то выполняется переход к команде, адрес которой определяется с помощью относительной адресации (смещение задано в команде LOOP). Если ECX=0, то выполняется следующая команда программы.

Команды пересылки. Основной командой этой группы является команда MOV, которая обеспечивает пересылку данных между дву-

мя регистрами или между регистром и ячейкой памяти. В некоторых процессорах реализуется пересылка между двумя ячейками памяти, а также групповая пересылка содержимого нескольких регистров в память или их загрузка из памяти.

Команды прерываний INT обеспечивают переход к одной из программ обслуживания исключений и прерываний (обработчику прерываний 0). При этом текущее содержимое PC и регистра состояния заносится в стек. Каждая из программ обработки соответствует определенному типу исключения или прерывания. Например, в процессорах Pentium выбор программы обработки определяется 8-разрядным операндом, задаваемым во втором байте команды INT. Вызов соответствующей программы обслуживания производится с помощью таблицы, в которой содержатся векторы исключений (прерываний) — адреса первых команд программ обслуживания.

Команды расширения SIMD («Single Instruction-Multiple Data» — «Одна команда — Множество данных»). Такие операции широко используются для обработки изображений, звуковых сигналов и в других приложениях. Для выполнения этих операций в состав процессоров введены специальные блоки, реализующие соответствующие наборы команд, которые в различных типах процессоров (Pentium, Athlon) получили название MMX («Milti-Media Extension») — Мультимедийное расширение; SSE («Streaming SIMD Extension») — Потоковое SIMD-расширение; «3D — Extension» — Трехмерное расширение.

Команды сравнения и тестирования. Сравнение операндов обычно осуществляется с помощью команды CMP, которая производит вычитание операндов с установкой значений признаков N (знака), Z (ноля), V (переполнения), C (переноса) в регистре состояния в соответствии с полученным результатом. При этом результат вычитания не сохраняется и значения операндов не изменяются. Последующий анализ полученных значений признаков позволяет определить относительное значение (>, <, =) операндов со знаком или без знака. Использование различных способов адресации позволяет производить сравнение содержимого двух регистров, регистра и ячейки памяти, непосредственно заданного операнда с содержимым регистра или ячейки памяти.

Команды условных переходов (ветвлений программы) производят загрузку в СчАК нового содержимого, если выполняются определенные условия, которые обычно задаются в соответствии с текущим значением различных признаков в регистре состояния. Если условие не реализуется, то выполняется следующая команда программы.

Компилятор — разновидность транслятора, осуществляет подготовку результирующего (исполнительного) модуля, который может выполняться на ЭВМ практически независимо от среды.

Компоновщик (редактор связей, линкер, linker, linkage editor) — программа, осуществляющая процесс сборки программы из объектных модулей, в котором производится их объединение в исполняемую программу и связывание вызовов внешних функций и их внутреннего представления (кодов), расположенных в различных объектных модулях. Эта программа собирает откомпилированный текст программы и функции из стандартных библиотек языка в одну выполняемую программу.

Косвенная адресация — случай, когда машинное слово содержит адрес другого машинного слова. Тогда доступ к данным во втором машинном слове через первое называется косвенной адресацией. Команды косвенной адресации имеются в любом компьютере и являются основой любого регулярного процесса обработки данных.

Лексика языка программирования — правила «правописания слов» программы, таких как идентификаторы, константы, служебные слова, комментарии. Лексический анализ разбивает текст программы на указанные элементы. Особенность любой лексики — ее элементы представляют собой регулярные линейные последовательности символов. Например, идентификатор — это произвольная последовательность букв, цифр и символа «_», начинающаяся с буквы или «_». Лексика ЯП анализируется и интерпретируется на фазе лексического анализа при трансляции.

Логические операции — Над значениями условных выражений можно выполнить логические операции И (&, AND), ИЛИ (|, OR) и НЕ (!, NOT), которые объединяют по правилам логики несколько условий в одно. Благодаря тому, что любая логическая операция может быть представлена с помощью трех основных логических операций, их набора в принципе достаточно для построения любого устройства процессора компьютера, а также для описания любых алгоритмов.

Логическое высказывание — любое предложение, в отношении которого можно однозначно сказать, истинно оно или ложно.

Логическое данное (`logical`) — тип данных, предназначенный для составления логических выражений и управления вычислительным процессом, типу соответствуют определенные операции и функции (логические).

Локальные переменные — переменные, которые «известны» только данной функции (действительны в данном блоке) и являются ее «собственностью». Создаются в памяти при входе в тело функции и аннулируются при выходе. Локальными переменными функция пользуется при необходимости иметь собственные данные.

Массив — упорядоченная последовательность переменных одного и того же типа, имеющая общее имя. Номер элемента в последовательности называется индексом. Количество элементов в массиве не может быть изменено в процессе выполнения программы. Элементы массива размещаются в памяти последовательно и нумеруются от 1 до *n*, где *n* — их количество в массиве.

Машинное слово — упорядоченное множество двоичных разрядов, используемое для хранения команд программы и обрабатываемых данных. Каждый разряд, называемый битом — это двоичное число, принимающее значения только 0 или 1. Разряды в слове обычно нумеруются, справа налево, начиная с 0. Количество разрядов в слове называется размерностью машинного слова или его разрядностью.

Машинный язык — совокупность машинных команд компьютера, характеризующаяся количеством адресов в команде, назначением информации, задаваемой в адресах, набором операций, которые может выполнить машина, и др.

Мегабайт (Мбайт) — единица измерения количества информации, равная 1000 Кбайт. Альтернативой является предложенная IEC бинарная единица 1 MiB (MibiByte) = 1024 KibiByte.

Микрокоманда — элементарное действие, обеспечивающее выполнение заданной операции, УУ процессора генерирует последовательность микрокоманд в соответствии с кодом поступившей команды. Каждая микрокоманда выполняется в течение одного машинного такта — периода тактовых импульсов, задающих рабочую частоту всех внутренних узлов и блоков микропроцессора. Таким образом, тактовая частота микропроцессора определяет время выполнения отдельных микрокоманд, последовательность которых обеспечивает получение необходимого результата операции (поступившей команды).

Многозадачность — режим одновременного решения нескольких задач на компьютере. Под задачей в данном случае понимается часть работы, выполняемой процессором.

Модульное программирование — разбиение программы на подпрограммы по специфике обрабатываемых данных. Для этой цели в ЯВУ используются функции и процедуры.

Наследование (ООП). Новый, или производный класс может быть определен на основе уже имеющегося, или базового. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса. Иначе говоря, новый класс наследует как данные старого класса, так и методы их обработки.

Независимость данных логическая (физическая) — свойство системы, обеспечивающее возможность изменять логическую (физическую) структуры данных без изменения физической (логической).

Объект (object) — совокупность данных и методов работы с ними, некоторые из которых могут использоваться другим приложением. Объектно-ориентированные технологии позволяют создателю объекта определить интерфейсы к возможностям объекта, скрыв при этом особенности его реализации. Это делает возможным использование объекта многими непосредственно не относящимися к нему приложениями. Несмотря на то, что этот термин широко используется в Windows, в большинстве случаев он применяется в значении «данные» или «нечто». Слово «объект» — это, пожалуй, самый заметный кандидат на звание наиболее перегруженного термина в области программного обеспечения.

Объектно-ориентированное программирование (ООП). Технология ООП прежде всего накладывает ограничения на способы представления данных в программе. Любая программа отражает в них состояние физических предметов либо абстрактных поня-

тий (объекты программирования), для работы с которыми она предназначена. В традиционной технологии варианты представления данных могут быть разными. В худшем случае программист может «равномерно размазать» данные о некотором объекте программирования по всей программе. В противоположность этому все данные об объекте программирования и его связях с другими объектами можно объединить в одну структурированную переменную. В первом приближении ее можно назвать объектом. Кроме того, с объектом связывается набор действий, иначе называемых методами. С точки зрения языка программирования это функции, получающие в качестве обязательного параметра указатель на объект. Технология ООП запрещает работать с объектом иначе, чем через методы, т. е. внутренняя структура объекта скрыта от внешнего пользователя. Описание множества однотипных объектов называется классом. Традиционная технология программирования «от функции к функции» определяет первичность алгоритма (процедур, функций) по отношению к структурам данных. Технология ООП определяет первичность данных (объектов) по отношению к алгоритмам их обработки (методам).

Объектный Модуль — файл данных, содержащий оттранслированные во внутренее представление собственные функции и переменные, а также информацию о внешних ссылках и точках входа модуля, информацию для редактора связей и может также содержать отладочную информацию (debug info).

Операнд — переменная, константа, выражение, участвующие в операции. Унарная операция — операция с одним операндом. Бинарная — операция с двумя операндами.

Оперативное запоминающее устройство (ОЗУ) служит для хранения выполняемой программы (или ее фрагментов) и данных, подлежащих обработке. В простейших микропроцессорных системах объем ОЗУ составляет десятки и сотни байт, а в современных персональных компьютерах, серверах и рабочих станциях он достигает сотен мегабайт и более. Так как обращение к ОЗУ по системной шине требует значительных затрат времени, в большинстве современных высокопроизводительных микропроцессоров дополнительно вводится быстродействующая промежуточная память (кэш-память) относительно ограниченного объема (от десятков до сотен килобайт).

Оператор — синтаксическая единица программы, которая отражает логику ее работы (последовательная, ветвящаяся, повторяющаяся). Для операторов характерен принцип вложенности: составными частями оператора могут быть любые другие операторы, и сам он, в свою очередь, может входить составной частью в оператор более высокого уровня.

Оператор цикла обеспечивает повторяющееся выполнение следующего за ним оператора (или блока) — тела цикла. Шаг цикла (итерация цикла) — конкретный факт выполнения тела цикла.

Операции сравнения («=» — равно, «!=» — не равно, а также «>», «<», «>=», «<=») дают в качестве результата значения «истина» (`true`) или «ложь» (`false`). Выражения с такими значениями называются условными, поскольку обозначают выполнение или, наоборот, невыполнение некоторого условия в программе. Они используются в условном операторе (`if`) и в операторах цикла (`do ... while`, `for`).

Операции управления программой. Для управления программой используется большое количество команд, среди которых можно выделить: команды безусловной передачи управления; команды условных переходов; команды организации программных циклов; команды прерывания; команды изменения признаков.

Операционная система обеспечивает следующие функции: управление процессором путем передачи управления программам, обработку прерываний, синхронизацию доступа к ресурсам, управление памятью, управление устройствами ввода-вывода, управление инициализацией программ, межпрограммные связи, управление данными на долговременных носителях путем поддержки файловой системы, связь с внешней средой.

Определение переменной «создает» переменную, выделяя под нее память, задавая имя, тип и, возможно, начальное значение.

Основание системы счисления — количество различных цифр, используемых для изображения чисел в данной системе счисления.

Открытость — свойство информационных технологий и систем, предполагающее способность объединять разные информационные системы, аппаратуру и программные продукты различных производителей, что делает возможным обмен между

ними данными, распределенный доступ к информационным ресурсам.

Отношение (relation) — агрегат данных, хранящийся в одной из таблиц (строка таблицы) табличной, реляционной БД или создаваемый виртуально в процессе выполнения операция над базой данных при выполнении запросов к данным.

Переменная — именованная область памяти программы, в которой размещены данные с определенной формой представления (типом). Для того чтобы воспользоваться переменной, необходимо произвести некоторые предварительные действия — выполнить определение переменной. Только при наличии такого определения транслятор будет знать ее имя, тип данных и, возможно, начальные значения.

Подпрограмма (процедура, функция) — средство, позволяющее многократно использовать в разных местах основной программы один раз описанный фрагмент алгоритма.

Поле (field) — однородный элемент данных определенного типа, описывающий аспект объекта или соответствующий фрагменту документа.

Полиморфизм (ООП) основывается на возможности включения в данные объекта также и информации о методах их обработки (в виде указателей на функции). Принципиально важно, что такой объект становится «самодостаточным». Будучи доступным в некоторой точке программы, даже при отсутствии полной информации о его типе, он всегда может корректно вызвать свойственные ему методы. Полиморфной называется функция, независимо определенная в каждом из группы производных классов и имеющая в них общее имя. Полиморфная функция обладает тем свойством, что при отсутствии полной информации о том, объект какого из производных классов в данный момент обрабатывается, она тем не менее корректно вызывается в том виде, к каком она была определена для данного конкретного класса. Практический смысл полиморфизма заключается в том, что программист может сделать регулярным процесс обработки несовместимых объектов различных типов при наличии у них такого полиморфного метода.

Пользователь (user) — физическое или юридическое лицо, непосредственно применяющее информационный ресурс, систему, технологию для решения задач.

Препроцессинг — предварительная фаза трансляции на уровне преобразования исходного текста программы с использованием директив препроцессора.

Препроцессор — предварительная фаза трансляции на уровне преобразования исходного текста программы с использованием директив препроцессора.

Прерывания (interruption) — ситуации, возникающие при поступлении соответствующих команд (программные прерывания) или сигналов от внешних устройств (аппаратные прерывания). При этом процессор останавливает всякую другую деятельность и вызывает программу-обработчик прерывания. По окончании ее работы он продолжает прерванную работу с того места, где она остановилась. Иногда аппаратные прерывания генерируются устройством в случае некорректной работы программы, например деление на 0. Программные прерывания генерируются программой для вызова различных подпрограмм из ОЗУ и ПЗУ.

Приведение типов. В выражениях в качестве операндов могут присутствовать переменные и константы разных типов. Результат каждой операции также имеет свой определенный тип, который зависит от типов операндов. Если в бинарных операциях типы данных обоих операндов совпадают, то результат будет иметь тот же самый тип. Если нет, то транслятор должен включить в код программы неявные операции, которые преобразуют тип операндов, т. е. выполнить приведение типов. Преобразование типов может включать в себя следующие действия: увеличение или уменьшение разрядности машинного слова, т. е. «усечение» или «растягивание» целой переменной; преобразование целой переменной в переменную с плавающей точкой, и наоборот; преобразование знаковой формы представления целого в беззнаковую и наоборот.

Прикладная программа (application) — программное средство, предназначенное для решения определенного класса задач и поддерживающее некоторый класс технологий. Прикладная программа, подготовленная на языке программирования, обычно называется исходной программой (source program). Двоичная версия программы, выходящая из компилятора и размещаемая в оперативной памяти для последующего выполнения, называется исполнительным модулем (object module) в машинном представлении. Она представляет собой последовательность

машинных инструкций (команд), которые конкретно указывают компьютеру, что надо делать.

Принстонская архитектура (часто называется архитектурой фон Неймана) характеризуется использованием общей оперативной памяти для хранения программ, данных, а также для организации стека. Для обращения к этой памяти используется общая системная шина, по которой в процессор поступают и команды, и данные.

Принципы фон Неймана включают в себя: принцип программного управления — программа состоит из набора команд, которые выполняются процессором автоматически друг за другом в определенной последовательности; принцип адресности — основная память состоит из перенумерованных ячеек и процессору в любой момент времени доступна любая ячейка; принцип однородности памяти — программы и данные хранятся в одной и той же памяти. Поэтому компьютер не различает, что хранится в данной ячейке памяти — число, текст или команда. Над командами можно выполнять такие же действия, как и над данными.

Присваивание — операция, которая значение выражения, стоящее справа от символа «=» запоминает в переменной или элементе массива, стоящем слева. При присваивании происходит преобразование форм представления (типов), если они не совпадают.

Программа — одна или несколько последовательностей связанных команд (инструкций), которые, будучи выполнены компьютером, реализуют определенную функцию или операцию. Примерами таких функций или операций могут быть: математические вычисления, поиск или сортировка списка объектов, сравнение и отбор объектов по какому-либо критерию, кодирование или декодирование данных, перемещение слов или чисел в закодированной форме в память компьютера, вывод результатов для печати или отображения.

Программист (programmer) — разработчик прикладных или системных программ, использующий системы программирования.

Программное средство — программа, предназначенная для многократного применения на различных объектах и массивах данных пользователя любым способом и снабженная комплектом программных документов.

Программный продукт — набор компьютерных программ, процедур и связанная с ними документация и данные.

Процедурно-ориентированный язык программирования (операторный язык программирования; императивный язык программирования) — язык программирования высокого уровня, в основу которого положен принцип описания (последовательности) действий, позволяющей решить поставленную задачу. Обычно процедурно-ориентированные языки описывают программы как совокупности процедур или подпрограмм.

Процессор — центральное устройство компьютера. Назначение процессора: управлять работой ЭВМ по заданной программе; выполнять операции обработки информации. Возможности компьютера как универсального исполнителя по работе с информацией определяются системой команд процессора. Эта система команд представляет собой язык машинных команд (ЯМК). В современных ПК процессор представлен одной СБИС, содержащей миллионы транзисторов (например, Pentium IV Prescott — 150 млн).

Пустой оператор, не производящий никаких действий, обычно обозначается символом «;», который встречается в программе «сам по себе». Пустой оператор используется там, где по синтаксису требуется наличие оператора, но никаких действий производить не нужно.

Регистр — специальная запоминающая ячейка, выполняющая функции кратковременного хранения числа или команды и выполнения над ними некоторых операций. Отличается от ячейки памяти тем, что может не только хранить двоичный код, но и преобразовывать его.

Регистр команд — регистр УУ для хранения кода команды на период времени, необходимый для ее выполнения.

Регистр состояния (SR, State Register, в микропроцессорах Pentium именуется `EFLAGS`) — определяет текущее состояние процессора при выполнении программы. Регистр содержит биты управления, задающие режим работы процессора, и биты признаков (флаги), указывающие характеристики результата выполненной операции: `N` — признак знака / старший бит результата (`N = 0` при положительном результате, `N = 1` при отрицательном); `C` — признак переноса (`C = 1`, если при выполнении операции выработан перенос из старшего разряда результата); `V` — при-

знак переполнения (V = 1, если при выполнении операций над числами со знаком произошло переполнение разрядной сетки процессора); Z — признак нуля (Z = 1, если результат операции равен нулю). Некоторые микропроцессоры фиксируют также другие виды признаков.

Редактирование связей — разрешение внешних ссылок и создание исполняемого модуля из совокупности объектных. Основные функции редактора связей — распределение памяти, разрешение внешних ссылок.

Результат функции — выходная переменная, которую формирует функция и значение которой используется затем в том месте программы, где она вызывается. Результат, как любая другая переменная, должен быть определенного типа.

Семантика языка программирования — это смысл, который закладывается в каждую конструкцию языка. Семантический анализ — это проверка смысловой правильности конструкции. Например, если мы в выражении используем переменную, то она должна быть определена ранее по тексту программы, а из этого определения может быть получен ее тип. Исходя из типа переменной, можно говорит о допустимости операции с данной переменной.

Символьное данное (character) — тип данных, предназначенный для ввода и отображения алфавитной, цифровой и спецсимвольной информации; типу соответствуют определенные операции над переменными и функции (строчные).

Синтаксис языка программирования — правила составления предложений языка из отдельных слов. Такими предложениями являются операции, операторы, определения функций и переменных. Особенностью синтаксиса является принцип вложенности (рекурсивность) правил построения предложений. Это значит, что элемент синтаксиса языка в своем определении прямо или косвенно в одной из его частей содержит сам себя. Например, в определении оператора цикла телом цикла является оператор, частным случаем которого является все тот же оператор цикла.

Система команд. Процессоры выполняют набор команд, которые реализуют следующие основные группы операций: операции пересылки; арифметические операции; логические операции; операции сдвига; операции сравнения и тестирования; битовые операции; операции управления программой; операции управления процессором.

Система операционная (operating system) — программная среда, расширяющая эксплуатационные возможности ЭВМ и предназначенная для управления данными, программами и связи с оператором (внешней средой).

Система программирования (programming system) — программная среда разработки приложений с использованием языка программирования, библиотеки функций, компилятора или интерпретатора ЯП.

Система счисления — совокупность правил наименования и изображения чисел с помощью набора символов, называемых цифрами. Системы счисления делятся на позиционные и непозиционные. Пример непозиционной системы счисления — римская, к позиционным системам счисления относятся двоичная, десятичная, восьмеричная, шестнадцатеричная. Здесь любое число записывается последовательностью цифр соответствующего алфавита, причем значение каждой цифры зависит от места (позиции), которое она занимает в этой последовательности. В непозиционных системах счислениях не представляются дробные и отрицательные числа.

Система управления базами данных (СУБД) — совокупность языковых и программных средств, предназначенных для создания, ведения и совместного использования БД многими пользователями. Систему управления базой данных можно также определить как специальный пакет программ, посредством которого реализуется централизованное управление базой данных и обеспечивается доступ к данным.

Системное программное обеспечение обеспечивает интерфейс между программистом или пользователем и аппаратной частью ЭВМ (операционная система, программы-оболочки) и выполняет вспомогательные функции (программы-утилиты).

Составной оператор — последовательность операторов, заключенная в операторные скобки ({}, **Begin End**), образует составной оператор (или блок) и входит в охватывающую его конструкцию как одно целое, т. е. становится с точки зрения транслятора одним оператором.

Средство автоматизации разработки программ **(CASE-средство)** — программное средство, поддерживающее все процессы жизненного цикла программного обеспечения: анализ и формулировку требований, проектирование, генерацию кода, тестирова-

ние, документирование, обеспечение качества, конфигурационное управление и управление проектом.

Стандартное машинное слово — машинное слово, размерность которого совпадает с разрядностью процессора. Большинство команд процессора использует для обработки данных стандартное машинное слово.

Статическое выделение памяти — выделение памяти под данные внутри сегмента данных программы. Такие данные лоступны на протяжении работы программы вплоть до ее завершения.

Стек — среда для размещения данных для возврата из подпрограмм, а также их аргументы и автоматические данные. Все это может потребовать достаточно большого размера стека. Как правило, программист может определять размер стека в программе.

Структура данных — способ отображения значений в памяти — размер области и порядок ее выделения (который определит характер процедуры адресации-выборки).

Структурное программирование — метод программирования, базирующийся на использовании процедурного стиля программирования и последовательной декомпозиции алгоритма решения задачи сверху вниз.

СУБД (DBMS, Data Base Management System) — программная среда, обычно предназначенная для манипулирования табличными (реляционными) базами данных.

Схема алгоритма (блок-схема) — графическое представление алгоритма в виде последовательности блоков, соединенных стрелками.

Таблица истинности — табличное представление логической схемы (операции), в котором перечислены все возможные сочетания значений истинности входных сигналов (операндов) вместе со значением истинности выходного сигнала (результата операции) для каждого из этих сочетаний.

Тип данных — форма представления данных, которая характеризуется: способом организации данных в памяти; множеством допустимых значений; набором разрешенных операций. Иначе говоря, тип данных — это схема определенного вида переменных, заложенная в транслятор. Сама переменная — это не что иное, как область памяти программы, в которой размещены данные в соответствующей форме представления, т. е. определенного

типа. Область памяти всегда ассоциируется в трансляторе с именем переменной. Тип данных `Integer`, (`Int`, `Fixed` и пр.) задает свойства целой переменной: она может принимать только целые значения в определенном диапазоне, зависящем от разрядности процессора, например, `-32 767...+32 767`. Все арифметические операции над целыми числами не выходят за рамки этого представления, т. е. дают также целый результат. Тип данных `Real` (`Double`, `Float` и пр.) задает свойства переменной с плавающей точкой. Число с плавающей точкой (или вещественное, действительное число) может принимать значения десятичной дроби (например, 287,3) и имеет ограничения на количество значащих цифр (точность представления). Все операции над переменными данного типа не выходят за рамки этой формы представления. Если же в операции присутствуют переменные обоих типов, то первый тип преобразуется во второй (целое число в вещественное).

Точка входа — адрес переменной или функции во внутреннем представлении модуля, к которым возможно обращение из других модулей.

Транслятор (от *англ.* translator — переводчик) — программа-переводчик. Преобразует текст программы, написанной на одном из языков высокого уровня, в программу, состоящую из машинных команд.

Трансляция — получение объектного кода из исходного.

Указатель — переменная, содержимым которой является адрес другой переменной. Соответственно, основная операция для указателя — это косвенное обращение по нему к той переменной, адрес которой он содержит. Указатель, который содержит адрес переменной, *ссылается* на эту переменную или *назначен* на нее; наоборот, переменная, адрес которой содержится в указателе, называется *указуемой* переменной.

Управляющие структуры (структурные операторы). Для того чтобы установить порядок выполнения отдельных операторов, в программах используются структурные операторы. К структурным операторам относятся составной оператор, условный оператор и цикл. Каждый из структурных операторов, в свою очередь состоит из элементарных или других структурных операторов. Управляющие структуры — составной оператор, условный оператор и цикл — равноправны в том смысле, что любая из них

может входить в состав другой. В программах встречаются группы операторов, объединенных в составной оператор, который является элементом цикла или условного оператора, и наоборот, циклы и условные операторы могут входить в составные операторы или другие циклы. Использование управляющих структур позволяет создавать разнообразные и довольно сложные программы.

Условный оператор if используется в программе, когда нужно выполнить одну или другую последовательность действий в зависимости от выполнения некоторого условия.

Файл (file) — именованный организованный набор данных определенного типа и назначения, находящийся под управлением операционной системы. Это однородная по своему составу и назначению совокупность информации, хранящаяся на носителе информации и имеющая имя. Правила образования имен файлов и объединения файлов в файловые системы зависят от конкретной операционной системы. Например, в операционной системы MS-DOS 6.0 имя файла состоит из двух частей: собственно имени и расширения имени (т. е. типа файла). Собственно имя файла состоит не более чем из восьми символов, исключая знаки арифметических операций, пробелов, отношений, пунктуации. В качестве имен файлов запрещены имена, являющиеся в MS-DOS именами устройств, например `con`, `lpt1`, `lpt2`. Расширение имени может состоять не более чем из трех символов, в том числе может отсутствовать. Если расширение есть, то от основного имени оно отделяется точкой, например `pict.bmp`, `lett.txt`, `doc.doc`. По имени файла можно судить о его назначении, так как для расширений установились некоторые соглашения, фиксирующие для ОС тип обработки файлов. Расширение `com` или `exe` имеют файлы программ, предназначенные для немедленного исполнения; `doc` — файлы с документами, подготовленные в текстовом редакторе Microsoft World; `bak` — резервные копии и т. д.

Файл ASCII (ASCII-file) — файл, содержащий символьную информацию в коде Latin-1 и символьную разметку.

Файл бинарный (binary file) — файл, содержащий произвольную двоичную информацию (текст с бинарной разметкой, программа, графика, архивный файл)

Файл графический (image file) — бинарный файл, содержащий данные, обычно полученные с помощью растрового сканера и соответствующие двумерному изображению объекта.

Файл табличный (table of data file) — файл, содержащий организованные данные, соответствующие строкам и столбцам некоторой таблицы и являющийся объектом или продуктом табличного процессора или СУБД.

Файл текстовый (text file) — файл, содержащий символьную информацию в одном из соответствующих кодов и коды, управляющие режимом отображения символов на печать и экранные устройства.

Файловая система (file management system) — динамически поддерживаемая информационная структура на устройствах прямого доступа (диски) и обеспечивающая функцию управления данными ОС путем указания связи «имя-адрес».

Фактические параметры — значения формальных параметров, с которыми будет выполняться тело функции при вызове, определяются списком фактических параметров, которые следуют в вызове функции вслед за ее именем, разделенные запятыми и заключенные в скобки. Перед вызовом функции фактические параметры, согласно списку, копируются в соответствующие формальные параметры. Таким образом, функция получает на вход данные через фактические параметры, которые она «видит» как соответствующие им формальные.

Фиксированная точка (`fixed`) — простейший тип числовых данных, когда число размещено в машинном слове, и диапазон значений зависит только от разрядности слова.

Фокус — понятие, относящееся к экранным компонентам. Находящийся «в фокусе» компонент является активным с точки зрения пользователя (например, располагается поверх других компонент и готов ко вводу данных).

Формальные параметры. После имени функции в круглых скобках идет список формальных параметров, разделенных запятыми. Как и все другие переменные, они имеют тип и могут быть обычными переменными и массивами. Формальные параметры содержат входные данные функции, передаваемые при ее вызове, но могут использоваться только самой функцией в процессе ее работы.

Формат данных (data format) — стандартизованное представление порции данных для хранения, передачи, отображения.

Функция — выполняет некоторое законченное действие и имеет «стандартный интерфейс» с набором параметров, через который она вызывается из других функций.

Цикл Пока (While). Проверка условия продолжения проводится до начала выполнения тела цикла, и если при первой проверке условие выхода из цикла выполняется, то тело цикла не выполняется ни разу.

Цикл До (For) применяется при необходимости произвести какие-либо вычисления несколько раз до выполнения некоторого условия. Особенностью этого цикла является то, что он выполняется хотя бы один раз;

Число с плавающей точкой (`float`) — числовое данное, размещенное в машинном слове в форме мантиссы и порядка, что позволяет представлять широкий диапазон значений; предполагает наличие встроенной или эмулируемой арифметики (операции с плавающей точкой). Для использования чисел с дробной частью, а также для расширения диапазона используемых числовых данных вводится форма представления вещественных чисел или чисел с плавающей точкой: $X = m \times 10^p$, например $25{,}4 = 0{,}254 \times 10^2$, где $0{,}1 < m < 1$ — значащая часть числа, приведенная к интервалу 0,1...1, называемая мантиссой, а p — целое число, называемое порядком. Аналогично, если взять 2 в качестве основания степени, то получим $X = m \times 2^p$, где $0{,}5 < m < 1$ — мантисса, а p — двоичный порядок.

Числовое данное (`numeric`) — тип данных, предназначенный для вычислений и составления арифметических выражений, которому соответствуют определенные операции и функции (арифметические).

Элемент данных (элементарное данное) — неделимое именованное данное, характеризующееся типом (напр., символьный, числовой, логический, и пр.), длиной (в байтах) и обычно рассчитанное на размещение в одном машинном слове соответствующей разрядности. Это минимальная адресуемая (идентифицируемая) часть памяти — единица данных, на которую можно ссылаться при обращении к данным. Ранние языки программирования (Алгол, Фортран) были рассчитаны на обработку элементарных данных или их простейших агрегатов — массивов (матрицы, векторы). С появлением ЯП Кобол появляется возможность представления и обработки агрегатов разнотипных данных (записей).

Язык ассемблера — система обозначений, используемая для представления в удобочитаемой форме программ, записанных в машинном коде. Перевод программы с языка ассемблера на машинный язык осуществляется специальной программой, которая называется *ассемблером*, и является по сути, простейшим транслятором.

Язык командный (command language) — система формализованных запросов оператора и ответов операционной системы (или прикладной программы), обеспечивающая функцию связи с оператором (для управления задачами и данными).

Язык программирования (programming language) — совокупность средств, предназначенных для описания алгоритмов, реализуемых в программах ЭВМ.

Оглавление

Голицына Ольга Леонидовна
Партыка Татьяна Леонидовна
Попов Игорь Иванович

Языки программирования

Учебное пособие

Редактор *А. В. Волковицкая*
Корректор *А. В. Алешина*
Компьютерная верстка *И. В. Кондратьевой*
Оформление серии *А. Никулина*
Рисунок на обложке *А. Скотаренко*

Сдано в набор 15.06.2007. Подписано в печать 27.08.2007. Формат 60×90/16.
Печать офсетная. Гарнитура «Таймс». Усл. печ. л. 25,0. Уч.-изд. л. 25,6.
Бумага офсетная. Тираж 3000 экз. Заказ № 6518.

Издательство «ФОРУМ»
101000, Москва — Центр, Колпачный пер., д. 9а
Тел./факс: (495) 625-32-07, 625-52-43
E-mail: mail@forum-books.ru

ЛР № 070824 от 21.01.93
Издательский Дом «ИНФРА-М»
127282, Москва, ул. Полярная, д. 31в
Тел.: (495) 380-05-40
Факс: (495) 363-92-12
E-mail: books@infra-m.ru
Http://www.infra-m.ru

По вопросам приобретения книг обращайтесь:

Отдел продаж издательства «ФОРУМ»
101000, Москва — Центр, Колпачный пер., д. 9а
Тел./факс: (495) 625-52-43
E-mail: natali.forum@mail.ru

Отдел продаж «ИНФРА-М»
127282, Москва, ул. Полярная, д. 31в
Тел.: (495) 363-42-60
Факс: (495) 363-92-12
E-mail: books@infra-m.ru

Центр комплектования библиотек
119019, Москва, ул. Моховая, д. 16
(Российская государственная библиотека, кор. К)
Тел.: (495) 202-93-15

Магазин «Библиосфера» (розничная продажа)
109147, Москва, ул. Марксистская, д. 9
Тел.: (495) 670-52-18, (495) 670-52-19

Отпечатано с предоставленных диапозитивов
в ОАО «Тульская типография». 300600, г. Тула, пр. Ленина, 109.